Doctoral Dissertation

A Sharding-based Hierarchical Blockchain for Large-Scale Transaction Processing

Yongrae Jo (조 용 래)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2025

대규모 트랜잭션 처리를 위한 샤딩 기반의 계층적 블록체인

A Sharding-based Hierarchical Blockchain for Large-Scale Transaction Processing

A Sharding-based Hierarchical Blockchain for Large-Scale Transaction Processing

by

Yongrae Jo
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science and Engineering

Pohang, Korea
12. 10. 2024
Approved by
Chanik Park (Signature)
Academic advisor

A Sharding-based Hierarchical Blockchain for Large-Scale Transaction Processing

Yongrae Jo

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

12. 10. 2024

Committee Chair	Chanik Park	(Seal)
Member	Hwangjun Song	(Seal)
Member	Gwangsun Kim	(Seal)
Member	Jisung Park	(Seal)
Member	Minseok Song	(Seal)

DCSE 조 용 래 Yongrae Jo
20172938
A Sharding-based Hierarchical Blockchain for Large-Scale Transaction Processing.
대규모 트랜잭션 처리를 위한 샤딩 기반의 계층적 블록체인.
Department of Computer Science and Engineering, 2025, 103p
Advisor : Chanik Park.
Text in English.

ABSTRACT

Blockchain is a distributed system capable of reaching consensus among mutually distrusting parties, providing decentralization, availability, fault tolerance, integrity, and transparency. Thanks to these features, blockchain-based services are gaining importance in the new paradigm of next-generation IT services such as Web3.0, Decentralized Finance (DeFi), and others. However, despite its benefits, blockchain has not been widely adopted in real-world applications due to its limited performance scalability. Consequently, research efforts are underway to improve performance at various levels of the blockchain architecture. For example, consensus algorithms, transaction processing methods, database, network protocols, storage systems, cryptography, trusted hardware, off-chain techqniues, and application-layer optimizations have been proposed to improve blockchain performance scalability.

Among these techniques, consensus and transaction processing are foundational blockchain core technologies with the highest potential impact on performance improvement because those technologies directly enhance the blockchain itself. Especially, sharding is considered one of the most effective ways to enhance blockchain performance with consideration of commonly referred to as the blockchain trilemma. Sharding is a method widely used in traditional databases for horizontal scalability by dividing the entire network into smaller groups called shards, where each shard group processes independent data and transactions. This approach achieves a high degree of parallelism as the number of shard groups in the system increases. In the context of blockchain, sharding divides the blockchain network into multiple subnetworks, allowing each to perform consensus only within its shard group, handling disjoint transactions independently. As a result, a sharding-based blockchain significantly reduces the overall cost of computing, storage, and network, achieving horizontal scalability.

However, designing a blockchain platform based on sharding is not straightforward and presents several challenges. First, overcoming limited availability; most existing blockchain sharding protocols follow performance-oriented sharding (also called complete or full sharding), where each member manages a completely disjoint set of shards, resulting in the shrunken size of each shard group. This isolated sharding approach dramatically reduces availability and can lead to catastrophic consequences (e.g., a complete data loss due to region-wide earthquakes). On the other extreme, availability-oriented sharding allows all members to hold all shards, operating parallel blockchain instances within each member while providing high availability. However,

as the number of members and shards increases, the computing overhead for each member grows significantly. Second, efficiently handling cross-shard transactions is also challenging; it requires a cross-shard protocol that atomically commits the transactions by coordinating multiple shards. The atomic commit protocol is known to be notoriously complex and inefficient. For example, the typical two-phase commit (2PC)-based cross-shard protocol requires the coordinator to implement a complex locking mechanism across the involved shards. Third, supporting dynamic locality; most existing blockchain sharding protocols do not fully consider dynamic locality—a workload characteristic with high locality that changes over time (e.g., mobile edge computing). Even if the initially sharded data is well-partitioned, locality can change over time (e.g., user mobility). If not handled in a timely manner, this can lead to an accumulative increase in the number of cross-shard transactions, ultimately degrading system performance.

In this thesis, we address the challenges that arise from introducing sharding into blockchain by proposing a sharding-based hierarchical blockchain. This approach involves a main chain at the higher level and multiple local chains at the lower level, where the former consists of members with a full copy of shards, and the latter comprises members with a single copy of a shard, respectively. We refer to our approach as balanced sharding and handle the challenges as follows.

For the first challenge of overcoming limited availability, the sharding-based hierarchical blockchain achieves this by replicating the blocks of lower-level chains onto the higher-level main chain. This approach combines the benefits of both performance-

oriented and availability-oriented sharding. Specifically, we organize the network into hierarchically sharded zones, where each zone leader manages the higher-level main chain, while other local members within each zone manage the lower-level local chains. This structure ensures availability by enabling recovery through the replicated local chains on the main chain in the event of a shard group failure, while still maintaining parallelism with multiple local chains.

For the second challenge of efficient handling of cross-shard transactions, we exploit the collocated sharding of the main chain, enabling these transactions to be processed simply and efficiently without further coordinations. For this, we propose the order-execute-order-validate (O-X-O-V) transaction processing model for a hierarchically sharded blockchain. This model operates by executing local transactions in parallel on local chains and integrating cross-shard transactions on the main chain, enabling a unified transaction processing platform in a simple and efficient manner.

For the third challenge of supporting dynamic locality, we design a state reshard protocol that safely transfers a user's state between local chains. The state reshard protocol uses the main chain as a trustworthy data source and transfers a user's state from the source local chain to their destination local chain via user-assisted reshard transactions. By doing so, the protocol effectively reduces the number of cross-shard transactions and improves the overall performance of the sharding-based hierarchical blockchain.

To realize our approach, we develop two blockchain systems: DyloChain and PyloChain, based on their fundamental network assumptions for the main chain. In a

synchronous network, DyloChain introduces a sharding-based hierarchical blockchain that supports dynamic locality by extending the order-execute-order-validate (O-X-O-V) transaction processing model to accommodate hierarchically sharded blockchains. In a partially synchronous network, PyloChain builds upon DyloChain by redesigning the main chain and introducing practical improvements, enhancing the scalability of the main chain and cross-shard transactions through a directed acyclic graph (DAG)-based consensus and scheduling technique.

Contents

List of Tables

List of Figures

I. Introduction

Blockchain is a distributed network system capable of processing transactions via reaching consensus among mutually distrusting members. The blockchain provides several desirable security properties. Decentralization: Blockchain processes transactions by reaching a consensus among participating members through message exchanges, determining how and in what order transactions should be processed. Tamper Resistance: Each participant organizes transactions into blocks, and these blocks are cryptographically secured and linked together by referencing the previous block using a cryptographic hash function, making undetected tampering virtually impossible. Availability: All blockchain data is replicated across all members, ensuring that even if a participant loses their data, it can be easily recovered from others. Fault Tolerance: The blockchain protects data integrity even in the presence of malicious actions, such as data tampering or deletion by some members. Transparency: Blockchain data and operations (e.g., smart contracts) are inherently open to all members, enabling transparency and verifiability. Therefore, compared to the traditional centralized system, where a trusted entity exists to process transactions solely on its own and is exposed to various single points of failure, the blockchain system is considered a better solution in terms of secure computing infrastructure.

Originally, blockchain was first introduced by the pseudonymous Satoshi Nakamoto [1] in 2008 to support cryptocurrency. Since its invention, Bitcoin has demonstrated stable operation over a long duration in real-world deployments. Following the remarkable success of Bitcoin, many communities have come to seriously recognize the technical value of blockchain and its immense potential impact on various aspects of human society. Due to this widespread interest, numerous blockchain platforms have been proposed, such as Ethereum [2], Hyperledger Fabric [3], Sui [4], Aptos [5], Solana [6], Ripple [7], Kaia [8], EOS [9], IOTA [10], and Tendermint [11]. These

platforms have proliferated rapidly to support a wide range of applications seeking to harness the benefits of blockchain technology. Examples include web 3.0 [12], decentralized finance (DeFi) [13], voting [14], gaming [15], supply chain [16], healthcare [17], Internet of Things (IoT) [10], metaverse [18], digital assets [19], certification [20], autonomous vehicles [21], lottery [22], and more.

However, blockchain-based applications have yet to achieve widespread adoption. This is primarily because it remains highly challenging for blockchain to deliver practical performance scalability while maintaining its strong security properties without compromise. Consequently, extensive research efforts are underway to enhance performance across various levels of the blockchain architecture. For instance, the consensus layer focuses on reducing the overhead incurred during the determination of transaction processing order among members, and it includes mechanisms such as proof-of-work (PoW) [23], proof-of-stake (PoS) [24], and Byzantine fault-tolerant (BFT) [25] algorithms. The execution layer addresses how transactions are executed in terms of concurrency and paralleism, encompassing models such as the traditional order-execute model and the execute-order-validate model. Beyond these, improvements are being made in the network layer, which optimizes gossip protocols and communication topologies; the storage layer, which aims to efficiently store and query large-scale blockchain data; and the cryptography layer, which optimizes signatures and privacy-related computations. Additionally, the hardware layer leverages specialized hardware, such as trusted execution environments (TEEs), to optimize specific blockchain operations. Off-chain methods, which process certain operations outside the blockchain, and the application layer, which optimizes high-level application logic and data structures, have also been proposed. These collective efforts aim to significantly enhance blockchain's performance scalability.

In particular, consensus layer and execution layer are core technologies of blockchain, possessing high potential for performance improvement as they directly impact the core operations of blockchain system. In this thesis, we focus on these two layers in designing large-scale blockchain systems, with sharding being considered one of the

most effective methods to enhance blockchain performance while addressing the so-called blockchain trilemma. Sharding is a method widely used in traditional databases for horizontal scalability by dividing the entire network into smaller groups called shards, where each shard group processes independent data and transactions. This approach achieves a high degree of parallelism as the number of shard groups in the system increases. In the context of blockchain, sharding divides the blockchain network into multiple subnetworks, allowing each to perform consensus only within its shard group, handling disjoint transactions independently. As a result, a sharding-based blockchain significantly reduces the overall cost of computing, storage, and network, achieving horizontal scalability.

However, designing a blockchain platform based on sharding is not straightforward and presents several challenges. First, overcoming limited availability; most existing blockchain sharding protocols follow performance-oriented sharding [26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39] (also called complete or full sharding), where each member manages a completely disjoint set of shards, resulting in the shrunken size of each shard group. This isolated sharding approach dramatically reduces availability and can lead to catastrophic consequences. For example, recent incidents such as the crash of cloud machines caused by bad software updates [40], the complete power outage of machines in a data center due to a fire [41], or region-wide earthquakes causing numerous base stations to go offline [42] demonstrate potential disasters. Such catastrophic events can paralyze essential IT infrastructure and services in society. The sharding-based systems, where data is distributed among smaller portions, can become even more vulnerable to these limited availability issues under such circumstances. On the other extreme, availability-oriented sharding [43, 44, 45, 46] replicates a full copy of shards (i.e., collocated) across every server and each shard performs consensus in parallel with a corresponding shard across other servers. This scheme supports data availability even when a certain group of servers fails. However, as the number of members and shards increases, the computing overhead for each member grows significantly. Thus, it is desirable to motivate the need for a balanced

approach to find a sweet spot between availability and performance. We depict those sharding schemes in Fig. 1.1

Second, efficiently handling cross-shard transactions is also challenging; Unlike local transactions, where their atomicity is simply guaranteed by submitting them to the corresponding shards that fully own the transaction's accessed states, cross-shard transactions require a cross-shard atomic commit protocol to atomically commit the transactions by coordinating multiple shards. The atomic commit protocol is known to be notoriously complex and inefficient. For example, the typical two-phase commit (2PC)-based cross-shard protocol requires the coordinator (e.g., client [30]) to implement a complex locking mechanism across the involved shards. However, the 2PC-based protocol not only performs poorly due to multiple rounds of message communication and locking, which block subsequent dependent transactions, but also faces a significant correctness issue due to its reliance on a specific coordinator that could fail. Moreover, in the decentralized flatten approach [28], which eliminates reliance on a coordinator, all members of all shards involved in a cross-shard transaction directly communicate with each other to process the transactions. However, this approach incurs very high communication costs, making it difficult to scale effectively. Additionally, there is an approach [47, 32] that processes a cross-shard transaction by splitting it into multiple sub-transactions, such as input transactions, output transactions, and relay transactions, as in [47]. While this method can handle cross-shard transactions, it suffers from rapidly increasing overhead as the proportion of cross-shard transactions grows.

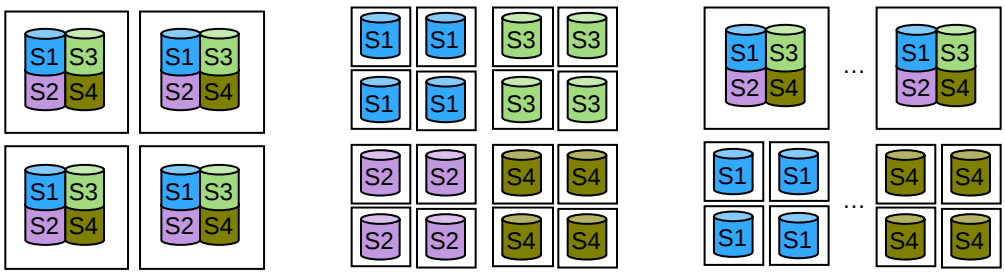
Third, supporting dynamic locality; most existing blockchain sharding protocols do not fully consider dynamic locality—a workload characteristic with high locality that changes over time (e.g., mobile edge computing). Even if the initially sharded data is well-partitioned to minimize the number of cross-shard transactions, locality can change over time (e.g., user mobility). If not handled in a timely manner, this can lead to an accumulative increase in the number of cross-shard transactions, ultimately degrading system performance. Therefore, a state resharding protocol is required to

appropriately move states to different shards when locality changes occur. However, transferring data between chains in a blockchain is far from simple. First, for correctness, a state must belong to only one shard at a time, and the state being transferred must always be the latest and correct version. Additionally, state resharding requires the user's consent, meaning the system cannot arbitrarily relocate states solely to improve efficiency. Existing methods for dynamic locality include approaches such as relocating related states to a single shard for processing cross-shard transactions [31] or applying complex mathematical analyses of historical patterns to minimize the probability of future cross-shard transactions [33, 48, 49, 50, 51]. However, these methods are unrealistic as they do not consider the user's consent. Furthermore, in performance-oriented sharding [26], the resharded state from the source local chain may become temporarily unavailable, as discussed in the first challenge. Therefore, state resharding requires a more trustworthy data source to ensure reliability and correctness.

In this thesis, we address the challenges that arise from introducing sharding into blockchain by proposing a sharding-based hierarchical blockchain. This approach involves a main chain at the higher level and multiple local chains at the lower level, where the former consists of members with a full copy of shards, and the latter comprises members with a single copy of a shard, respectively. We refer to our approach as balanced sharding scheme and handle the challenges as follows.

For the first challenge of overcoming limited availability, the sharding-based hierarchical blockchain achieves this by replicating the blocks of lower-level chains onto the higher-level main chain. This approach combines the benefits of both performance-oriented and availability-oriented sharding. Specifically, we organize the network into hierarchically sharded zones, where each zone leader manages the higher-level main chain, while other local members within each zone manage the lower-level local chains. This structure ensures availability by enabling recovery through the replicated local chains on the main chain in the event of a shard group failure, while still maintaining parallelism with multiple local chains.

For the second challenge of efficient handling of cross-shard transactions, we ex-



(a) Availability Sharding (b) Performance Sharding (c) Balanced Sharding

Figure 1.1: Blockchain Sharding Schemes

ploit the collocated sharding of the main chain, enabling these transactions to be processed simply and efficiently without further coordinations. For this, we propose the order-execute-order-validate (O-X-O-V) transaction processing model for a sharding-based hierarchical blockchain. This model operates by executing local transactions in parallel on local chains and integrating cross-shard transactions on the main chain, enabling a unified transaction processing platform in a simple and efficient manner.

For the third challenge of supporting dynamic locality, we design a state reshard protocol that safely transfers a user's state between local chains. The state reshard protocol uses the main chain as a trustworthy data source and transfers a user's state from the source local chain to their destination local chain via user-assisted reshard transactions. The protocol follows a main chain-driven 2PC protocol to process the reshard transactions. By doing so, the protocol safely transfers the user's state by atomically committing the reshard transactions and effectively reduces the number of cross-shard transactions, improving the overall performance of the system.

To realize our approach, we develop two blockchain systems: DyloChain and PyloChain, based on their fundamental network assumptions for the main chain. First, in a synchronous network, DyloChain introduces a sharding-based hierarchical blockchain that supports dynamic locality by extending the order-execute-order-validate (O-X-O-V) transaction processing model to accommodate hierarchically sharded blockchains. In DyloChain's hierarchical structure, higher-level servers (called M-nodes) handle

cross-zone communication with a main chain to ensure global consistency and availability, while lower-level servers (called L-nodes) validate local blocks within a local chain. For main chain, DyloChain adopts a simple method over a synchronous network that incorporates concurrently produced local blocks from each zone into the main chain in a throughput-oriented manner. For large-scale transaction processing on locality workload in a sharding-based hierarchical blockchain, DyloChain proposes order-execute-order-validate (O-X-O-V) model. The O-X-O-V model extends the existing execute-order-validate (X-O-V) [3] model to accommodate the hierarchical sharded blockchain. Specifically, the O-X-O-V model optimizes efficiency through speculative local updates on each local chain (O-X) for the early removal of potential conflicts within a zone and enables high parallelism across zones. Subsequently, the model aggregates these local blocks onto the main chain (O-V), ensuring a globally consistent total order across zones. This is facilitated by an extended validation procedure that detects interferences between concurrently dependent transactions across zones. DyloChain resolves interferences using a state synchronization protocol that synchronizes the corresponding main states of the interfered local states into the involved local chains. DyloChain improves locality by employing a state reshard protocol that transfers a user's state ownership between source and destination local chains by atomically committing user-issued reshard requests to the involved chains.

Despite DyloChain's novel O-X-O-V transaction model designed for large-scale transaction processing, a critical improvement are required for its practical adoption. Designing a main chain over a synchronous network presents a significant challenge. In a distributed network, the synchronous network assumes a maximum delay bound Δ_{sync} for message delivery and relies on it for the system's safety and liveness. For example, DyloChain's main chain consensus allows the creation of main blocks composed of a fixed number of local blocks with specific indices from each local chain in every synchronous round. However, practically, precisely predicting the timeout value for message transmission can be challenging. Additionally, the synchronous network can be problematic in terms of performance bottlenecks, particularly when

slower nodes or local chains with varying block propagation speeds are present. The strict rule that all local chain blocks must be included in a main block before its generation introduces substantial delays.

Building upon DyloChain, we propose PyloChain, which operates in a partially synchronous network with a redesigned main chain. In the partially synchronous network assumption, it is stipulated that synchronous and asynchronous periods alternate. Therefore, the maximum message delay is Δ_{sync} during the synchronous period, and $\Delta_{sync} + \Delta_{GST}$ during the asynchronous period, where Δ_{GST} represents the global stabilization time (GST) with an unknown message delay bound. Because the asynchronous period only partially persists, the partially synchronous network guarantees that messages sent will eventually be delivered to the receiver.

Based on the partially synchronous network assumption, PyloChain employs a DAG-based mempool [52, 53] to append highly concurrent local blocks from multiple sharding zones simultaneously in a leaderless manner, guaranteeing local block availability and higher throughput over partially synchronous network. Based on the DAG mempool, an efficient local BFT consensus produces a larger main block that contains totally orderd local blocks, by periodically interpreting local view of the DAG mempool.

For transaction processing, PyloChain adopts and enhances the order-execute-order-validate (O-X-O-V) transaction processing model [54] with a simple scheduling technique. As menteiond, this model allows speculative parallel execution of local transactions across zones, significantly reducing the burden on main block processing. However, the model is not resilient to cross-shard transactions because as they increase, local transactions are increasingly aborted, adding synchronization overhead across the hierarchy. PyloChain enhances cross-shard resilience with a simple scheduling technique in which cross-shard transactions are batch-processed at the end of each main block processing cycle, effectively improving cross-shard transaction scalability. Furthermore, this scheduling accelerates main block processing by allowing parallel processing of local blocks from independent local chains, achieving greater scalability

through the use of multiple processing threads.

Also, in sharded blockchain networks, each shard typically has a specific node called a zone leader that representatively communicates with zone leaders of other shards for cross-zone communication. Therefore, auditing the trustworthiness of this node is crucial. We demonstrate that DyloChain simply achieves both safety and liveness under the assumption of a synchronous network. For PyloChain, we design a fine-grained auditing mechanism that externalizes the zone leader's critical actions onto the main chain, allowing local members to audit them. To achieve this, PyloChain divides the operational semantics of the DAG-based main chain protocol into fine-grained phases, enabling asynchronous audits to verify the zone leader's behavior at each phase.

We implement our systems in GoLang, using Hyperledger Fabric [55] as the baseline local chain operating PBFT-based consensus [56] within each sharding zone. For PyloChain's main chain, we utilize Narwhal/Bullshark [57] as DAG-based mempool with BFT consensus.

For the feasibility study, we evaluate our systems on a cluster of multiple machines. Our comprehensive experiments of DyloChain and PyloChain demonstrate their performance and overhead under a range of configurations with SmallBank, a simple banking application [58]. For DyloChain, we show that it effectively handles large-scale transaction processing thanks to its O-X-O-V model and experimentally validate the effectiveness of its state resharding protocol. For For PyloChain, we observed that PyloChain demonstrates better performance scalability compared to DyloChain. For example, under 20% global transactions and 12 zones, PyloChain achieved 1.49x higher throughput and 2.63x faster latency. Also, by comparing PyloChain with other sharding schemes, including availability sharding and performance sharding, we show that PyloChain effectively balances performance and availability.

In summary, the main contributions of this thesis are as follows.

- We propose DyloChain, a hierarchically sharded blockchain that supports dy-

namic locality with an O-X-O-V transaction processing model and state reshard protocol.

- We propose PyloChain that practically improves DyloChain with DAG-based consensus and scheduling technique.
- We analyze DyloChain and PyloChain in terms of correctness and security properties.
- We implement and evaluate DyloChain and PyloChain to demonstrate its performance scalability under various configurations.

The rest of this thesis is organized as follows: Chapter II describes the background and related works. Chapter III presents DyloChain, a sharding-based hierarchical blockchain with the O-X-O-V transaction processing model and state reshard protocol, in synchronous network. Chapter IV introduces PyloChain, a practical improvement of DyloChain with a DAG-based consensus and scheduling technique, in partially synchronous network. Finally, Chapter V discusses our systems with future works. Chapter VI concludes the thesis.

II. Background and Related Works

2.1 Transaction Processing Methods

A transaction consists of a series of instructions that read state data, perform computations on the read data, and write back the computation results. In blockchain, transactions can be processed in various forms. They range from simple payment transactions, considering only the sender and receiver in cryptocurrency-only blockchains, to smart contract-based transactions that execute general programs autonomously. Transaction processing plays a crucial role in improving blockchain scalability.

Traditionally, blockchains like Bitcoin [1], Ethereum [2] process transactions using the order-execute (O-X) model. In this model, the consensus protocol outputs a total order of blocks, which are then sequentially processed by each node, resulting in poor performance.

One of the pouplar alternatives for the O-X model is execute-order-validate (X-O-V) model, which is designed to support highly concurrent transaction executions. It was firstly proposed by Hyperledger Fabric [3] (HLF), a production-grade permissioned blockchain platform primarily used by a consortium of enterprises. Due to the practical performance of X-O-V, HLF has found many real-world applications in various industries, such as supply chain [59], finance [60], land registry [61], insurance [62], and healthcare [63].

The X-O-V process comprises three steps, as illustrated in Fig. 2.1.

First, peers receive a transaction proposal issued by a client application, which contains the smart contract ID and its arguments. Each peer locally manages a state database and holds the smart contracts required to execute the received transaction proposal. Based on this, the peers execute the transaction proposal and compute the execution results, including a read/write set, obtained by invoking the smart contract

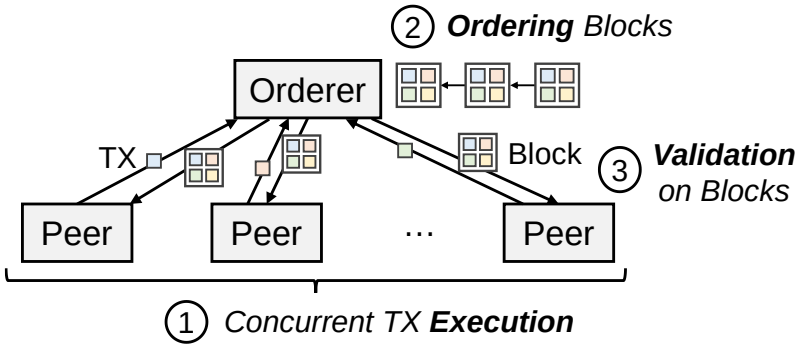


Figure 2.1: Execute-Order-Validate (X-O-V) Transaction Processing Model.

against the state database. The read set is a collection of entries, with each entry comprising a key and a version, where the version indicates the block number and transaction offset within the block that last modified the key. Similarly, the write set is a collection of entries, with each entry comprising a key and a value. It is important to note that the execution phase is highly concurrent. This concurrency arises because each peer can execute their own set of transactions simultaneously in a decentralized manner. Additionally, peers do not update the local state with the execution results during this phase. Subsequently, each peer generates a transaction containing the read/write set and submits it to the ordering service. Second, the orderer collects concurrently executed transactions and organizes them into totally ordered blocks. The orderer is a fault-tolerant state machine replication system that implements atomic broadcast. These blocks are then distributed to the peers. Third, peers validate the blocks using multi-version concurrency control [64]. During this step, each peer sequentially processes the transactions in the block and verifies that, for all transactions containing a read set, the versions of the entries in the read set match the current state in the local database. If the versions are up-to-date, the transaction is successfully committed; otherwise, it is aborted. Thereafter, only the successfully validated transactions are applied to the blockchain ledger along with their write sets.

Recent research shows that the performance of the X-O-V is heavily affected by

workload characteristics, particularly in terms of the degree of contention [65, 66, 67, 68]. The X-O-V exhibits superior performance under conditions of low contention; however, it suffers considerable performance degradation when subjected to highly conflicting workloads, primarily owing to frequent transaction aborts during the validation step.

2.2 Dynamic Locality

The dynamic locality is a common real-world workload pattern commonly observed especially in geo-distributed databases, and presents both opportunities and challenges for building high-performance system designs. Dynamic locality is where a certain set of data is frequently accessed within a particular zone for an extended period. This set gradually changes over time, for instance, due to the mobility of devices. To illustrate the concept of dynamic locality, consider the example shown in Fig. 2.2, where data is partitioned across zones, and users primarily transact with other users in the same zone. With optimal data placement before time t, only local transactions (TXs) are issued. However, after time t, some users, such as user C, move to another zone. Consequently, all transactions involving remote access to user C's data are issued as expensive cross-zone transactions that typically require a complex distributed coordination protocol such as two-phase commit (2PC).

Many studies have been conducted to effectively support dynamic locality, particularly in distributed edge database literature [69, 70, 71, 72, 73], assuming crash fault tolerance (CFT). In those environments, a variety of mission critical applications are deployed such as smart city [74], industrial internet-of-things [75], autonomous vehicles [76], and point-of-sales [77], which require practical performance and fault-tolerance. Therefore, dynamic locality motivates the development of a new system for permissioned blockchains in such an environment, which can leverage the properties of a dynamic locality workload to enhance system efficiency.

We now argue the limitations of the traditional X-O-V blockchain [3] in support-

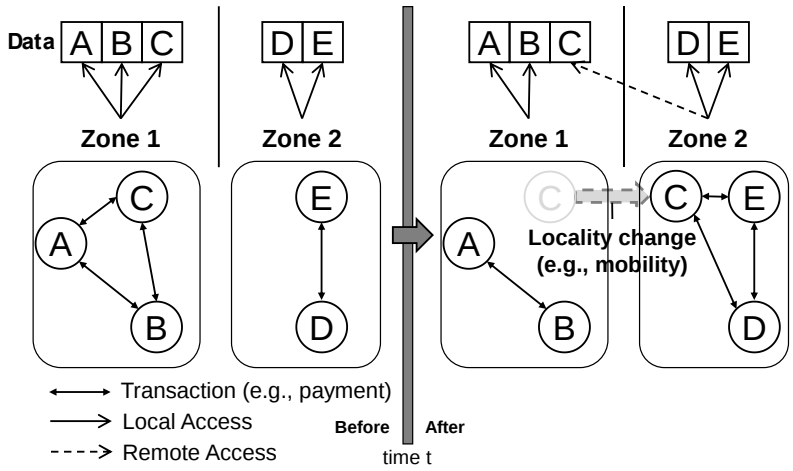


Figure 2.2: An Example of Dynamic Locality.

ing dynamic locality. First, as dynamic locality commonly is found in geo-distributed networks that provide hierarchical infrastructure across geo-distributed zones (with heterogeneous computing servers), deploying a globally single X-O-V based blockchain over those zones could lead to inefficiencies such as increased communication latency across those zones. This motivates a hierarchical blockchain architecture where each layer is tasked with different distributed processing using different computing resources and is capable of handling transactions from multiple different zones in a hierarchical fashion for efficiency.

Second, the traditional X-O-V itself is not designed to support sharding across zones, and accordingly, it does not support cross-shard transactions or dynamically changing locality (e.g., mobility of user devices) across zones. Third, although X-O-V efficiently processes workloads with a high number of mutually independent transactions, it generates numerous aborted transactions for workloads with high dependencies, resulting in reduced system performance [65, 66, 67]. This implies that X-O-V may perform poorly under dynamic locality, which can involve highly conflicting transactions within a zone. Therefore, while X-O-V preserves the strength of handling predominantly independent transactions between zones, it needs to be enhanced for

better performance under such workloads.

2.3 Sharding Schemes

Sharding is regarded as a promising solution to enhance blockchain scalability. For instance, one approach to sharding divides the entire blockchain network into multiple subnetworks, with each subnetwork managing only a portion of the complete blockchain ledger, referred to as a "shard." This division distributes the workload across multiple subnetworks, ultimately reducing consensus, network communication, and storage overhead in each smaller network with fewer members. However, introducing sharding could reduce reliability, as each shard is managed by a smaller number of members. For example, in a performance sharding setup, a single blockchain with a typical BFT protocol [25] that follows the $3f + 1$ model, where f is the maximum number of tolerable Byzantine members out of $3f + 1$ members in total, can tolerate up to 33 Byzantine members when managed collectively by 100 members. If this system is divided into 10 subnetworks with an equal number of members, each sharded network can tolerate up to only 3 Byzantine members, making it much more vulnerable to worst-case disastrous scenarios (e.g., region-wide earthquakes or data center fires) or security attacks. In such cases, the failure of a single shard group could lead to complete unavailability of that shard, which could cause severe damage to the system. Consequently, blindly pursuing high scalability with performance sharding may not be a viable option for real-world applications that handle highly security-sensitive data (e.g., supply chain on consortium blockchain).

Therefore, we argue that building a practical sharded blockchain requires balancing both performance and availability. To achieve this, it is necessary to explore trade-offs between traditional performance sharding and availability sharding. As a result, we propose a sharding-based hierarchical blockchain that divides the network into a two-level structure: the higher level maintains a full copy of all shards, while the lower level holds only a single copy of each shard. This hierarchical design provides

excellent opportunities to effectively explore a balanced approach.

We provide existing solutions based on the three sharding schemes to clearly compare our approach.

2.3.1 Performance Sharding

Performance sharding is a traditional approach to achieving blockchain scalability due to its maximal parallelism and has therefore been widely adopted in various blockchain systems [28, 26, 27, 47, 33, 30, 32, 29, 34, 35, 36, 37, 38]. A key challenge in performance sharding lies in the cross-shard protocol, which must ensure atomic commitment of cross-shard transactions across involved shards. For example, traditional coordinator-based two-phase commit (2PC) protocols [30, 28, 26, 27, 33, 34, 39], methods that split transactions into multiple sub-transactions [32, 47], techniques that move related states to a single shard [32], or a decentralized protocol where all participants communicate directly without a coordinator [28] have been developed to address this complexity. Those cross-shard protocol is known to be notoriously complex and inefficient. Interestingly, some methods [26, 78] introduce a hierarchy within performance sharding to further enhance scalability and simplify cross-shard protocols. In these blockchains, lower-level members store only a single shard of their ledger, while higher-level members either maintain a summarized view of their child ledgers or order cross-shard transactions [26, 78].

This performance sharding approach, due to the shrinking size of each shard group, involves a significant trade-off in availability, despite achieving higher scalability. In contrast, PyloChain adopts a balanced approach: it uses a hierarchical structure in which lower level members can benefit from performance sharding in terms of reduced storage/networking, and parallel executions, while higher level members maintain a full copy of all shards, ensuring availability in the event of sharding zone failures, and simplifying the cross-shard protocol as well.

2.3.2 Availability Sharding

Availability sharding allows each member to maintain a full copy of all shards, thereby preventing total data loss in the event of a single shard group failure [43, 44, 46]. Moreover, it offers the advantage of leveraging collocated sharding within each member for cross-shard transactions, which helps avoid the complex coordination required for isolated shards as in performance sharding. In Meepo [43, 45], a sharded consortium blockchain, all members maintain a full copy of the shards, enabling higher cross-shard efficiency by internalizing cross-shard transaction processing within a single member while ensuring shard availability. In PShard [44], every member participates in all shards, which consist of a root chain and multiple child chains, and executes cross-shard transactions using a root chain-driven 2PC-style commit protocol. Similarly, OHIE [46] achieves excellent throughput and availability by allowing each member to maintain up to k parallel chains. In synchronous network settings, each member in OHIE periodically derives a sequence of confirmed blocks (SCB) to establish a global total order across chains for their local view of the parallel chains.

However, availability sharding systems impose a significant burden on each consortium member, as they require every member to hold a full copy of all shards. This becomes especially demanding for less-capable members as the number of members and shards grows.

2.3.3 Balanced Sharding

Finally, there exists a balanced sharding scheme harmonizing the above two approaches, with some members maintaining a full copy of shards and others a single copy. Pyramid [79] uses layered sharding with i-shards and b-shards, which hold a single shard and all shards, respectively. To handle cross-shard transactions, Pyramid employs a b-shard-driven 2PC protocol: the b-shard proposes a cross-shard block to i-shards, which accept or reject based on local transaction conflicts. The b-shard then commits or aborts based on the responses. In contrast, PyloChain streamlines this by

executing cross-shard transactions on the main chain based on total order and notifying local chains in a single phase. Unlike Pyramid, which supports cryptocurrency applications in a public blockchain, PyloChain is designed for KV-based general workloads [27, 3] in a permissioned consortium blockchain.

2.4 Other Scaling Solutions

Another approach to improving blockchain sharding systems involves resharding techniques [33, 49], which aim to reduce the number of costly cross-shard TXs. This is achieved by observing the historical transaction pattern over a system-defined fixed interval and performing a locality analysis to appropriately place blockchain states (account/balance or UTXO) across shards, thereby minimizing cross-shard TXs in the future. For example, BrokerChain [33] analyzes Ethereum workloads to model a state graph based on TX history between users and applies a mathematical graph partitioning algorithm to optimally divide disjoint subgraphs, minimizing the number of edges between them. However, these kinds of techniques focus on applying complex mathematical methods to find an optimal state placement based on relatively long-term history, which is not necessary in our dynamic locality workload where only the most recent changes (e.g., user movement between zones) in data locality are sufficient to consider.

DAG-based blockchain systems [80] redefine the traditional linearly growing blockchain as a DAG, enabling enhanced concurrency and parallelism for scalability. Unlike the traditional linear blockchain, which accepts only one of the multiple competing blocks for the same slot (or in a round), DAG-based blockchains can simultaneously accept several blocks in the form of a DAG within a slot, resulting in higher throughput. Caper [81] is a TX-based, permissioned DAG ledger with parallel chains operating over a disjoint set of servers (i.e., sharding zone). In Caper's hierarchical protocol [80], classic BFT protocols are first applied to separated zones, then an upper-layer protocol achieves the final consensus to establish a total order of all

cross-application TXs across zones.

DAG BFT [82, 52, 53] is a block-based, permissioned DAG ledger. As we leverage DAG BFT at a higher-level main chain for PyloChain, we provide more detailed description of DAG BFT. DAG BFT addresses problems in the Byzantine Atomic Broadcast (BAB) [82], which guarantees consensus among correct servers with the properties, including *agreement* for a consistent message delivery in a round, *integrity* for an exactly-once delivery of a message in a round, *validity* for eventual delivery of a message in a round, and *total order* for the same delivery order for messages. The DAG BFT proceeds in sequential rounds and supports parallel block proposals by all servers in each round, which are then to be accepted in subsequent rounds with DAG mempool. It consists of two steps. First, the servers disseminate transaction data through reliable broadcast, which guarantees block availability and helps build a local view of the DAG mempool. Second, the servers periodically interpret their local view of DAG mempool to achieve consensus, i.e., total order of blocks from DAG mempool, using some deterministic ordering logic (e.g., depth first search).

We summarized its operation as in [52]. At the beginning of each round, servers propose a block that contains transactions received from clients and each block corresponds to a vertex in the DAG. By using reliable broadcast, they exchange their block proposals with other servers to achieve the integrity and availability of their proposed blocks. Specifically, servers broadcast their block proposals in each round, then they reply with an acknowledgment if valid. Once $2f + 1$ distinct acknowledgments are collected for the round, each server combines them to create a certificate of block availability and also broadcasts it to other servers. This is then included in the block of the next round, resulting in the block having references, i.e., edges, to previous blocks in the previous rounds. Finally, by periodically and deterministically interpreting their local view of DAG structure, they achieve consensus by locally calculating the total order (i.e., topological sorting) of the blocks with a randomly selected (or predefined) anchor vertex in the local view of DAG structure.

Ethereum scaling solutions that aim to alleviate the load on the Ethereum Main-

net through off-chain methods include several approaches: Side-chain [83] utilizes an entirely independent blockchain with its own consensus mechanism, supporting asset transfers to and from the Ethereum mainnet via a two-way bridge protocol. Roll-up [84] processes multiple TXs off-chain, bundling the resulting state changes into a single TX for the Mainnet. Danksharding [85] enhances the efficiency of Layer-2 solutions (Roll-up), by re-engineering the Mainnet architecture (e.g., Blob-carrying transactions). Plasma [86] manages multiple child chains in a hierarchical manner. However, these solutions do not address the challenges of dynamic locality and assume a completely different setting from ours with regard to their permissionless nature and transaction processing model.

III. DyloChain: A Sharding-based Hierarchical Blockchain over Synchronous Network

In this chapter, we propose DyloChain, a hierarchically sharded blockchain that supports dynamic locality by extending the X-O-V transaction processing model. In DyloChain, multiple independent zones run lower-level local chains in parallel with a higher-level main chain for global consistency across zones. A local chain within a zone contains a disjoint set of sharded states among the entire states in the main chain. Each local chain independently performs consensus on transactions occurring within its zone, generating local blocks in parallel. Then, these local blocks are concurrently proposed across zones and aggregated at the higher-level main chain, where a total order across zones is established. DyloChain considers two different types of nodes: higher-level servers, called M-nodes, which have relatively powerful computing resources and are responsible for cross-zone communications and maintaining the main chain; and lower-level servers, called L-nodes, which are less capable and are responsible for auditing the operations of M-nodes and maintaining a local chain within a zone.

Based on the hierarchical structure, DyloChain introduces a novel hierarchically fashioned transaction processing model called order-execute-order-validate (O-X-O-V). By combining the order-execute (O-X) within a zone and the execute-order-validate (X-O-V) across zones, it takes advantage of the strengths of each approach. The O-X within a zone optimizes transaction processing by serializing predominant local transactions (TXs) with speculative local updates. This helps to exploit the high degree of locality with early removal of potential conflicts between the local TXs due to frequent access to certain states within a zone. The X-O-V establishes a global total order of all TXs with an extended validation procedure to validate both local TXs and global TXs

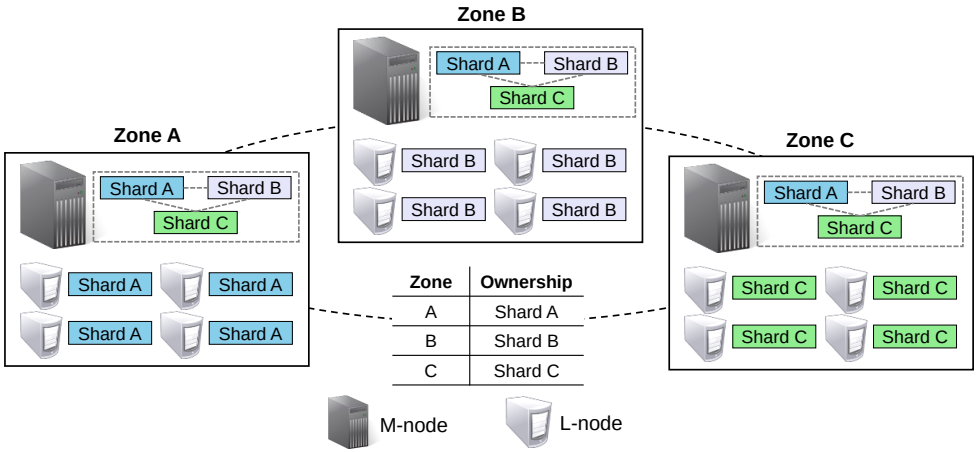


Figure 3.1: System Overview with an Example Deployment

based on multi-version concurrency control (MVCC). Note that most TXs are mutually independent of each zone to ensure fewer conflicts across all zones, so the X-O-V based main chain can maximize its success rate in this locality workload. DyloChain, with the O-X-O-V model, provides a state synchronization protocol responsible for addressing interfered states identified from the main chain validation by synchronizing the corresponding main chain's states into local chains. Additionally, DyloChain enhances locality by employing a state reshard protocol that transfers a user's state ownership between zones by atomically committing the user's reshard request issued when they move to their destination zone.

3.1 Model and Assumptions

We assume that identities of participants are known in advance, and Byzantine attacks are infrequent as is typical in consortium blockchains. Each zone has an M-node, a higher-level large-scale data center that can offer sufficient networking/computing power for cross-zone communications, and multiple smaller lower-level servers, called L-nodes, within a zone. There exist f_L and f_M Byzantine failures out of $3f_L + 1$ L-

nodes within each zone and $2f_M + 1$ across f_M zones, respectively, implying $3f_L + 2$ servers in a zone. However, there could be additional machines for those servers to be recruited upon detecting the failure in a zone. If an M-node fails in a zone, we assume that another nearby M-node in the same zone is recruited on-demand to resume normal operations following a view change protocol (e.g., PBFT [25]). We assume servers within the same zone are significantly more proximate to each other compared to servers from remote zones, so in case of an M-node failure within a zone, any newly recruited M-node in the same zone can replace the failed one. An entire zone failure is possible, yet the maximum number of concurrent zone failures is limited to f_M and the probability of the zone failure is mutually independent between zones. DyloChain differentiates between two network types: *partially synchronous* network within a zone, with an unknown but finite bound Δ_L, and a *synchronous* network across zones with a known fixed upper bound Δ_M.

All servers are authenticated offline (e.g., by a certificate authority) and only those servers are granted permission to participate in the system. Each server is equipped with a private/public key pair and can be identified by its public key and a corresponding certificate. All communications between servers occur via point-to-point channels. Communications across zones take place with possibly higher latency, while communications within a zone happen with relatively low latency. We assume that standard cryptography cannot be compromised.

We assume a workload that exhibits dynamic locality where most transactions only require local data access, and such locality can gradually change over time. Applicable scenarios for this may include location-based services such as hyperlocal services, ride-sharing services, autonomous driving systems, and a consortium of local banks.

3.2 Overview

Nodes Two types of blockchain nodes exist in DyloChain: M-node and L-node. An M-node holds a main chain, i.e., full copy of the shards, and communicates with other M-nodes across zones. In contrast, an L-node only holds a local chain, i.e., single copy of a shard, for endorsing local blocks and after-the-fact auditing of relevant operations performed by the M-node in the same zone. An example of DyloChain deployment is illustrated in Fig. 3.1.

Hierarchically Sharded Blockchains There are two types of blockchains: sharded local chains within each zone and a main chain consisting of local blocks across zones. Higher-level M-nodes manage both the main chain and the local chain of their respective zone, while L-nodes in each zone manage only the local chain of their zone. For a local chain, we utilize a PBFT-based self-verifiable blockchain [87, 56], where succinct cryptographic consensus messages (denoted by π_L), signed by each L-node in the zone, are recorded in the blockchain. This allows external verifiers to examine the messages to validate remote blockchains. For a main chain, we enforce a simple rule to organize locally committed blocks from the local chains. Specifically, one main block is created in each synchronous round as the M-node in each zone concurrently broadcasts a local block (or an empty block if no user TXs are present) to the other M-nodes across zones. Afterwards, after verifying the local blocks by analyzing a local quorum π_{LQ} consisting of a series of π_L, the M-nodes construct a main block containing one local block from each zone by applying a deterministic ordering rule (e.g., lexicographical order with zone index). Note that the main chain can be considered parallel chains as in DAG-based blockchain [80].

Each L-node within a zone must be able to verify that the user TXs issued in their zone have been correctly committed to the main chain. For this purpose, we define a proof for generating a correct i-th main block (denoted by $\pi_M(i)$), which summarizes the processing results of each main block and contains the necessary information for

L-nodes in each zone to verify the main block. The proof is generated by M-nodes after creating a main block and is broadcasted to remote M-nodes. Thus, for an i-th main block to be validated, a quorum of the corresponding proofs, denoted by $\pi_{MQ}(i)$, must be identified on the local chain.

Specifically, for each main block, a $\pi_M(i)$ includes the following fields: Generator: the creator's identity and signature; M-Header: the main block number, the hash of the main block, and the hash of the previous main block; L-Headers: an ordered list of local block headers; StateHash: a hash of the main chain's state database after executing the i-th main block; Vals: an ordered list of zone-specific validation results of user TXs, which include a commit/abort flag alongside the TX's string-based ID in the i-th main block; Note that for performance reasons, Vals contains only the aborted user TXs; Entries: a set of zone-specific digests of interfered states and their corresponding main states, used by the state synchronization and state reshard protocol. Each zone's L-node examines these fields and validates the matching quorum to ensure that the local block appears correctly on the main chain and that its zone's M-node has executed the main block correctly. Additionally, it verifies whether optimistically executed TXs within a zone have been successfully committed/aborted on the main chain without/with conflicts, and checks Entries for the presence of further required operations.

Ownership Each zone is associated with ownership, system-wide metadata for tracking current location of zone-specific user state; Initially, each user's state is assigned to a specific zone and is associated with only one ownership at any given time. The ownership can be changed by reshard TXs, which are issued when the user moves to their final destination zone and are subsequently handled by the state reshard protocol based on the main chain. Note that, the main chain guarantees that the current authoritative location of each user's state can be verified. The M-nodes manage ownership for all users, while L-nodes manage ownership only for users relevant to their zone. Ownership is also used for determining TX types as follows: a TX is classified as a lo-

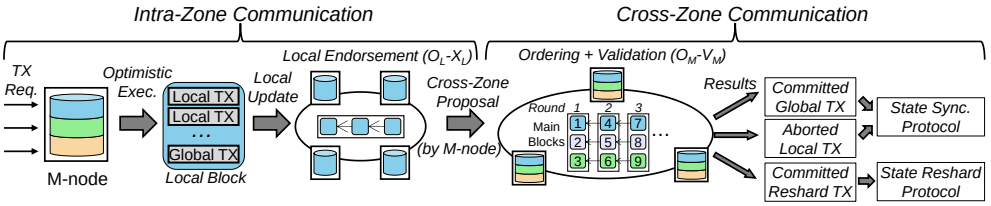


Figure 3.2: Transaction Processing Flow of DyloChain

cal TX when accessing a single shard; as a global TX (or cross-shard) when accessing multiple shards without changing ownership; and as a reshard TX when accessing two shards but changing the ownership of accessed states (e.g., mobile users).

3.3 Transaction Processing

In this section, we explain the transaction processing of DyloChain, as illustrated in Fig. 3.2. DyloChain's transaction processing includes the O-X-O-V flow and subsequent two protocols, including the state synchronization protocol and the state reshard protocol.

3.3.1 O-X-O-V Flow

Users in each zone submit their TX requests to a nearby M-node within the same zone. Initially, these requests are assigned a local sequence number, and then the TX type is determined based on the states accessed by the TX before execution. M-nodes can infer the accessed states if they are evident from the invocation arguments, such as sender/receiver in a simple payment or keys in a key/value (KV) store. If not (for instance, when accessed states are determined dynamically during execution), M-nodes can simulate the transaction or use static analysis to reveal the accessed states before execution.

Determining the TX types, M-nodes optimistically execute TX requests according to their type. For local TXs, the M-node executes the TX based on the latest

local shard, generates the read/write set, and speculatively updates the local states. For global TXs, the M-node executes the TX based on states from the main chain, generates the read/write set, but does so without any updates. For reshard TXs, the M-node executes a special smart contract and generates a TX that includes the issuing user's key, source zone, and destination zone. The user's state value for reshard TXs has not yet been finalized at this moment. Note that if any of the keys accessed by a user TX are locked by the state synchronization protocol or the state reshard protocol, which will be explained subsequently, the M-node will abort the TX and respond with a corresponding lock certificate.

After executing TXs, the M-node creates a new local block, propagates the block to L-nodes for local endorsement (i.e., obtaining π_L), and commits it to the local chain. Only the endorsed local blocks, certifying their integrity, are permitted by the M-node for cross-zone proposals for creating a new main block. After each round of synchronously broadcasting local blocks by the M-nodes, each M-node lexicographically sorts the collected local blocks based on the index of a zone, thereby deterministically generating the main block for that round.

Subsequently, M-nodes validate all user TXs in the main block to finalize the results. Assuming no conflicts exist, all user TXs are clearly committed, regardless of their types. However, when conflicts arise, the situation becomes complex. We demonstrate how O-X-O-V mechanisms handle user TXs in terms of concurrency conflicts. There are three cases to consider.

First, the all-local workload is simple to handle in the O-X-O-V because all TXs are serialized before their local execution on a local chain, and they are mutually independent across zones. Therefore, there are no conflicts to address on the main chain. Second, the all-global workload is also manageable in the O-X-O-V because all TXs are processed in the typical X-O-V flow, which is similar to HLF [3]. Lastly, the mixed workload where local TXs and global TXs coexist is crucial and requires careful consideration for correct execution. Because of independent local ordering in parallel combined with speculative local updates, several challenges for consistent validation

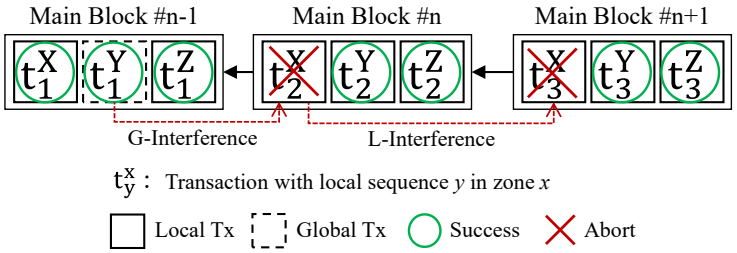


Figure 3.3: Global TX Interference with Local TX

of the two different types of TXs with mutual dependencies arise when concurrent user TXs with conflicts are serialized on the main chain, as shown in Fig. 3.3.

Consider two TXs, a global TX and a local TX, denoted by t_1^Y and t_2^X in the Fig. 3.3, respectively, that are concurrently issued and attempting to read and update the same state s. When they are serialized on the main chain in the order of t_1^Y and t_2^X, they are subject to MVCC validation, which determines whether they will be committed or aborted. Recall that local TX and global TX build their read sets from local and main states in execution step, respectively. This implies that these two TXs should have consistent version information before entering the validation step on the main chain. To address this, we define a data structure called *version map* that translates versions in each local chain to the corresponding versions in the main chain. The version map contains a set of mapping entries of (k, v_L, v_M), where k is a key, and v_L and v_M are the versions for the local chain and main chain, respectively. Note that a version of k indicates the latest TX's block height that had updated the value of key k. With the version map, every local version of the read set from local TXs is converted to the main version. This conversion enables consistent MVCC validation between local and global TXs, as well as among local TXs from different zones. The version map is managed by M-nodes and updated only by the committed user TXs on the main chain, which ensures that the entries in the version map are globally consistent.

Upon conducting the MVCC validation with the help of the version map, only t_1^Y is successfully committed, while t_2^X is aborted on the main chain. We refer to this

Algorithm 1: State Synchronization Protocol

on receiving *i-th main block* B_i^M *at* $p \in \Pi_M^z$**:**
- $s_I \leftarrow$ identify interfered states from committed global TXs and aborted local TXs from B_i^M
- $s_M \leftarrow$ retrieve corresponding main states from s_I
- $\pi_M(i) \leftarrow$ generate a main block proof from B_i^M and (s_I, s_M)
- $\pi_{MQ}(i) \leftarrow$ exchange $\pi_M(i)$ among Π_M and derive a quorum certificate for the B_i^M
- broadcast $<$ SYNC, $\pi_{MQ}(i)$, s_I^z, $s_M^z >$ to $p \in \Pi_L^z$

on receiving $<$ SYNC, $\pi_{MQ}(i)$, s_I^z, $s_M^z >$ at $p \in \Pi_L^z$**:**
- verify $\pi_{MQ}(i)$; verify $D(s_I^z)$ and $D(s_M^z)$ by comparing the corresponding ones in $\pi_{MQ}(i)$
- identify the involved states from s_I^z and commit the corresponding main states from s_M^z

type of conflict as *G-Interference*. The fact that the aborted t_2^X had already updated the local state on the local chain due to speculation leads to inconsistencies between the local states and the main states. Furthermore, if there are any ongoing local TXs that have dependencies with t_2^X in the same zone and have concurrency with global TX, t_1^Y, then the subsequent local TXs, e.g., t_3^X, which are also doomed to fail on the main chain, also induce inconsistent states, which we call *L-interference*.

3.3.2 State Synchronization Protocol

DyloChain handles these interferences with two protocols. First, a state synchronization protocol resolves existing interferences by synchronizing the correct main states to the involved local chains. Second, a state reshard protocol is responsible for eliminating the cause of these interferences, i.e., mobile users' global TXs, by relocating the user's state to their current zone with the user-issued reshard TX, resulting in their subsequently issued transactions being local TXs.

We first explain state synchronization protocol. We introduce several notations as follows: p for any server, Π_M^z for an M-node in a zone z, Π_L^z for L-nodes in a zone z, Π_M for all M-nodes across zones, and D for a digest function.

The protocol is invoked after each zone's M-node validates the i-th main block B_i^M and identifies the interfered states from committed global TXs and aborted local TXs (L1-2). Then, the main states, s_M, corresponding to the interfered states are retrieved (L3). These generated main states are consistent among the M-nodes because they are based on the same i-th main block. After that, a main block proof, $\pi_M(i)$, is generated from B_i^M, s_I, and s_M (L4). This proof is then exchanged among the M-nodes to obtain a quorum certificate, $\pi_{MQ}(i)$ (L5). The M-nodes then broadcast a SYNC message to the L-nodes in their respective zones, which contains $\pi_{MQ}(i)$ and their zone-specific interfered states s_I^z, and their corresponding main states s_M^z (L6). Upon receiving the SYNC message, the L-nodes verify whether the $\pi_{MQ}(i)$ shows that their M-node had correctly performed on the main chain. Also, L-nodes verify s_I^z and s_M^z by comparing digests of them with the corresponding ones in $\pi_{MQ}(i)$. If they are valid, L-nodes identify the states related to their zone based on their local ownership metadata from s_I^z and commit the corresponding main states from s_M^z to the local chain (L7-9).

Note that our state synchronization protocol is single-phase, where the main states are synchronized to the involved local chains with a single message transmission. This may cause temporary inconsistencies across local chains in terms of global TXs that involve multiple local chains due to communication latency. Thus, the stale main states could be used for executing user TXs in the involved local chains until the completion of the protocol. However, since all user TXs are eventually validated on the main chain, it ensures that two inconsistent transactions will never both be committed to the main chain.

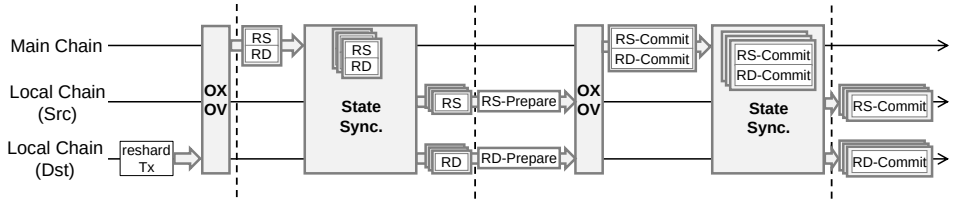


Figure 3.4: Communication Flow of State Reshard Protocol

3.3.3 State Reshard Protocol

The state reshard protocol differs from the state synchronization protocol in that it updates the ownership metadata of the issuing users from the source zone to the destination zone. This implies that the ownership metadata must be atomically committed to both the source and destination chains; otherwise, a user's ownership could simultaneously belong to two zones, violating our assumption that a user's ownership should be associated with only one zone at a time. Therefore, the state reshard protocol follows a typical two-phase commit (2PC) style of processing flow for atomic commitment.

We illustrate the submission and processing communication flow of a reshard TX in Fig. 3.4. We first note that the state reshard protocol leverages the O-X-O-V flow and the state synchronization protocol to abstract hierarchical communication operations: the former for transmitting messages required for the protocol from local chains to the main chain, and the latter for transmitting from the main chain to the involved local chains.

The state reshard protocol is invoked after the user issues a reshard TX upon moving to the destination zone and after it is confirmed on the main chain via the O-X-O-V flow. M-nodes generate the reshard-src (RS) and reshard-dst (RD) messages, lock the user's state, and send these messages to the source chain and destination chain, respectively, via the state synchronization protocol. Then, the L-nodes in the involved local chains verify these RS and RD quorum messages and lock the user's states. Proofs for these actions are included in the RS-Prepare and RD-Prepare messages, which

are sent to the main chain via the O-X-O-V flow. Finally, on the main chain, once the RS/RD-Prepare messages with their local quorums are confirmed from both the source and destination chains, the M-nodes generate the RS-Commit and RD-Commit messages to unlock the user's state and update the corresponding ownership metadata, and send them to the source and destination chains, respectively, via the state synchronization protocol. Finally, the involved L-nodes verify these commit messages, unlock the locked state, and update the ownership appropriately.

Algorithm 2: State Reshard Protocol

1 **on receiving** i-th main block B_i^M *at* $p \in \Pi_M$**:**
2 $rt \leftarrow$ identify valid reshard TX rt from B_i^M
3 $s_u \leftarrow$ get the latest main state at height i of a user $rt.u$
4 lock(s_u)
5 broadcast $<$ RS, rt, $s_u >$ and $<$ RD, rt, $s_u >$ to z_s and z_d, respectively, via state sync. protocol
6 **on receiving** $<$ RS, rt, $s_u >$ at $p \in \Pi_L^{z_s}$**:**
7 lock(s_u); send $<$ RS-Prepare, $D(rt)$ $D(s_u) >$ to $p \in \Pi_M^{z_s}$
8 **on receiving** $<$ RD, rt, $s_u >$ at $p \in \Pi_L^{z_d}$**:**
9 lock(s_u); send $<$ RD-Prepare, $D(rt)$ $D(s_u) >$ to $p \in \Pi_M^{z_d}$
10 **on receiving** i'-th main block $B_{i'}^M$ *where* $i' > i$ *at* $p \in \Pi_M$**:**
11 **if** *both quorums on* RS-Prepare *and* RD-Prepare *are identified* **then**
12 $cert_{\text{commit}} \leftarrow$ create a commit certificate that contains the both quorums
13 broadcast $<$ RS-Commit, $D(rt)$, $cert_{\text{commit}} >$ and $<$ RD-Commit, $D(rt)$, $cert_{\text{commit}} >$ to z_s and z_d, respectively, via state sync. protocol
14 metadata[u] $\leftarrow z_d$; unlock(s_u)
15 **on receiving** $<$ RS-Commit, $cert_{\text{commit}}$, $D(rt) >$ at $p \in \Pi_L^{z_s}$**:**
16 verify $cert_{\text{commit}}$; commit rt and remove u; metadata[u] $\leftarrow z_d$; unlock(s_u)
17 **on receiving** $<$ RD-Commit, $cert_{\text{commit}}$, $D(rt) >$ at $p \in \Pi_L^{z_d}$**:**
18 verify $cert_{\text{commit}}$; commit rt and add u with s_u; metadata[u] $\leftarrow z_d$; unlock(s_u)

The detailed operational sequence of the state reshard protocol is illustrated in Algorithm 2. M-nodes identify a valid reshard TX rt from the i-th main block B_i^M and retrieve the latest main state of the user who issued rt (L1-2). Note that all M-nodes

capture a consistent state based on the same height i of the main chain. M-nodes then lock s_u (L4) and send RS and RD messages, containing rt and s_u, to the source zone z_s and destination zone z_d respectively via the state synchronization protocol (L5). Once the RS/RD messages are delivered to the source and destination chains, the involved L-nodes lock s_u and respond with RS/RD-Prepare messages containing the digests of $D(rt)$ and $D(s_u)$ to their respective M-nodes in the form of transactions (L6-9). These messages can then be identified in a subsequent i'-th main block $B^M_{i'}$, where $i' > i$, following the O-X-O-V flow. If the quorums for the Prepare messages from the source and destination zones are both confirmed (L10-11), the M-nodes generate a commit certificate $cert_{\text{commit}}$ that contains both local quorums (L12). Then, RS/RD-Commit messages, along with the commit certificate and $D(rt)$, are broadcast to the source and destination zones respectively via the state synchronization protocol (L13). Next, the M-nodes update the metadata of the corresponding user to the destination zone z_d and unlock the state s_u (L14). Finally, when the L-nodes from the source and destination zones receive the RS/RD-Commit messages, they verify the commit certificate and, if valid, commit rt. This means removing the user u from the source zone and adding the user u to the destination zone, respectively, and then updating the ownership metadata accordingly to z_d and unlocking s_u (L15-17).

We briefly analyze the performance of the state reshard protocol in terms of required quorums. We consider a quorum to be a set of local or main proofs that contains a set of signatures from L-nodes or M-nodes, respectively, i.e., π_{LQ} or π_{MQ}. The Algorithm 2 requires three subsequent quorums: First, a main quorum among M-nodes is needed for RS/RD messages via the state synchronization protocol. Second, a local quorum among L-nodes at each of the involved local chains is needed for the RS/RD-Prepare messages via the O-X-O-V flow. Third, a main quorum among M-nodes is required for the RS/RD-Commit messages via the state synchronization protocol. Note that for the reshard TX to appear in the main block via the O-X-O-V flow, an additional local quorum among L-nodes at each of the involved local chains is required. Thus, the state reshard protocol requires four subsequent quorums.

3.4 Analysis

We argue the correctness of DyloChain by providing our analysis on concurrent transaction executions under interference for the O-X-O-V flow, the consistency of the state synchronization protocol, and the atomicity of the state reshard protocol. Also, we provide a security analysis of DyloChain.

Correctness Analysis

A key's version information represents the latest user TX by which the corresponding key-value in the state database is updated. For each key, there are two types of versions: local version v_L and main version v_M, which are defined for local chain and main chain, respectively.

Definition 1 (Version Map, $\mathcal{V}$ and $\overline{\mathcal{V}}$)**.** *We define two version maps, $\mathcal{V}$ and $\overline{\mathcal{V}}$ for a given key k: $\mathcal{V}$: $(k, v_L) \rightarrow v_M$ and $\overline{\mathcal{V}}$: $k \rightarrow \overline{v}_M$.*

Given a key k and its local version v_L, $\mathcal{V}$ returns the corresponding main version v_M, which points to the previously committed local TX on the main chain, and the local TX matches the given v_L on the local chain. $\overline{\mathcal{V}}$ simply returns the latest main version $\overline{v}_M$ of the given key k. Therefore, for each key in the read set of local TXs from different local chains, M-nodes can consistently validate the local TX in terms of MVCC, by comparing the corresponding main version v_M of the key with the latest main version $\overline{v}_M$. For example, for a key k, if v_M from $\mathcal{V}$ (k, v_L) is the same as $\overline{v}_M$ from $\overline{\mathcal{V}}$ (k), the local TX is to be successfully committed on the main chain. Note that these version mappings are only updated when user TXs with its write set are committed on the main chain.

Note that for all keys that had previously been updated by interfered local TXs, $\mathcal{V}$ maps those versions into an empty version (i.e., $\mathcal{V}(v_L)$ returns $\emptyset$). Also, there could exist v_M with no v_L mapped due to global TXs (i.e., writing keys for the first time).

Definition 2 (Interference Relationship)**.** *Consider two TXs, tx_g and tx_l, which are a global TX and a local TX, respectively. If tx_g is ordered before tx_l in a main chain,*

denoted by $tx_g \prec_M tx_l$, and there exist common keys between tx_g and tx_l, then we call tx_g G-interferes tx_l. Formally, the two TXs have interference relationship if the following conditions hold:

- *There exists a key k such that k appears at both tx_g's write set and tx_l's read set.*
- *$tx_g \prec_M tx_l$ (meaning $\overline{\mathcal{V}}(k)$ is to be updated by tx_g first. Then, we have $\overline{v}_M \leftarrow \overline{\mathcal{V}}(k)$)*
- *Given the v_L of k in the read set of tx_l, we have $v_M \leftarrow \mathcal{V}(k, v_L)$.*
- *Then, for v_M and $\overline{v}_M$, $v_M \prec_M \overline{v}_M$*

Theorem 1 (Correctness under Interference)**.** *For two transactions, having interference relationship, only one of these two transactions is to be committed.*

Proof. Assume that two transactions, tx_g and tx_l, are ordered in a main chain, having an interference relationship. All M-nodes maintain a version map $\mathcal{V}$ and update $\mathcal{V}$ based on valid writes of committed transactions. After tx_g updates all v_M of keys in write set of the tx_g, there are two cases in which tx_l is to be aborted due to the occurrence of an interference relationship. First, tx_l that immediately follows tx_g is destined to be aborted because there exists a version v_L of a key in the read set of tx_l, such that v_L does not have the latest version mapping in the main chain. Second, the other v_L of in-flight local TXs (i.e., L-Interference) does not have the corresponding v_M, which results in them being aborted in the main chain. □

Theorem 2 (Consistency)**.** *The state synchronization protocol eventually ensures consistency of interfered states by synchronizing the corresponding main states into the involved local chains.*

Proof. First, synchronizing interfered states from aborted local TXs is straightforward because the corresponding states belong to only a single ownership within a local chain. Second, synchronizing interfered states from valid global TXs, whose write sets involve multiple local chains, may cause temporary inconsistencies between the

involved local chains due to communication latency, as the SYNC messages might reach the local chains at different times. During this temporary inconsistency period, each local chain can execute local or global TXs that read and update the affected states. However, since these user TXs will eventually be validated on the main chain, there will be no cases where two conflicting states are committed simultaneously, and TXs executed based on stale values will not be committed on the main chain. Additionally, due to network assumptions, the SYNC messages are guaranteed to be eventually delivered to the involved local chains, ensuring that the main states are eventually synchronized with the involved local chains. Third, we explain the correctness of the state synchronization protocol under M-node failures. By assumption, DyloChain allows for up to f_M concurrent failures out of $2f_M + 1$ M-nodes in the worst-case scenario. The failure of an M-node relies on a PBFT-like view change protocol within a zone, which eventually replaces the failed M-node with a new nearby M-node in the same zone. Therefore, as long as the underlying view change protocol is not compromised, the state synchronization protocol guarantees that the corresponding main states of interfered states will eventually be delivered to the involved local chains under M-node failures.

□

Theorem 3 (Atomicity)**.** *The state reshard protocol atomically commits a reshard TX to the involved local chains.*

Proof. A user's ownership must belong to only one location at a time. We prove that the state reshard protocol preserves this assumption by atomically committing the reshard TX.

We consider the case where the RS/RD-Commit messages are delivered to the involved L-nodes at different times. In other words, after confirming that both local chains have locked the user's state from the RS/RD-Prepare messages, there may be an interval where the reshard TX is committed to only one of these local chains first due to communication latency. For example, suppose the RD-Commit message is delivered to the destination local chain, z_d, first, causing the L-nodes in z_d to update the ownership of the user to z_d, while the L-nodes in the z_s have not yet received

the RS-Commit message and still assume the user's ownership is z_s. However, since the L-nodes in z_s are still locking the user's state, executing transactions based on the user's associated ownership is unavailable. Additionally, even if they receive a new SYNC message from a subsequent main block that could affect the user's state with remote global TXs before receiving the RS-Commit, each message contains a main block number i, enabling the L-nodes to safely process the received messages in order. Similar arguments apply to the opposite case where the RS-Commit message is first delivered to z_s. Finally, because the network is assumed to be partially synchronous, the delayed message will eventually be delivered, and the L-nodes in z_s will eventually unlock the state, guaranteeing the liveness of the protocol. The correctness involved with cross-hierarchy communication and under failure scenarios relies on the correctness theorems of the O-X-O-V flow and state synchronization protocol, as shown in Theorem 1 and Theorem 2.

□

Security Analysis

We discuss security properties of DyloChain under various attack scenarios.

Malicous M-nodes

1) Omission Attack. A faulty M-node may neglect to respond to TX requests, cease the broadcasting of local blocks to other zones, or fail to deliver a main block proof π_M. To address this, DyloChain employs a timeout mechanism to detect such omission attacks. Specifically, if an M-node does not respond to TX requests, clients will redirect these requests to nearby L-nodes within the same zone. Subsequently, L-nodes initiate a timer upon forwarding these requests to the M-node. If the results of these TX requests are not delivered within a predefined timeout value, the M-node is deemed faulty. In scenarios where there is a failure to broadcast local blocks across zones or deliver π_M, L-nodes can still detect the omission because DyloChain operates under the assumption of a synchronous network across zones, and L-nodes utilize a

reliable synchronous timer to ensure the timely delivery of π_M. 2) Execution Attack. First, the M-node can maliciously misclassify user TXs such as handling global TXs as if they were local TXs, or vice versa. As these attacks involve ownerships, they can be detected by L-nodes that tracks changes in ownership in state reshard protocol. Second, by exploiting the fact that L-nodes are unable to verify global TXs at the time of proposal, a malicious M-node can include incorrect global TXs in the main chain. However, since other M-nodes can perform validations for those TXs, they can eventually report the incorrectness of the malicious M-node by including it in a π_M. 3) Order Consistency Attack. Malicious M-nodes can propose a different order of local blocks for the main chain, which invalidate local agreement. This attack can be detected by L-nodes with π_{MQ}, a quorum of main block proofs, which includes a sequence of local block headers. Note that duplicating or skipping certain local blocks on the main chain can also be detected by the L-nodes with the same manner. 4) Fake Proof Attack. Malicious M-nodes may generate fake proofs on a main block. However, L-nodes in each zone only verify the π_{MQ} that contains a quorum-based consistent result with valid signatures from other M-nodes.

Malicious L-nodes and Users

L-nodes may either crash or exhibit Byzantine faults. However, each zone contains $3f_L + 1$ L-nodes, allowing the system to tolerate up to f_L Byzantine L-nodes. A malicious user can falsely report their location (e.g., issuing reshard TXs while staying in the same zone), regardless of their actual current location. To address this, we may consider a trustworthy reporting mechanism, such as proof-of-location [88]. For example, when mobile users submit their reshard TXs, they must include endorsements from L-nodes in the destination zone, verifying their actual presence in that zone. By considering the physical distances between zones, L-nodes can measure network latencies, ensuring the user is highly likely to be in the destination zone. Once L-nodes agree on the user's current zone, they provide their endorsements to the user, who then attaches them to their reshard TXs. Additionally, malicious users may generate an ex-

cessive number of global TXs or intentionally conflicting TXs to abort other correct client TXs. In this case, we can apply economic penalties, such as imposing TX fees, or limiting the maximum issuance of TXs within a certain duration (e.g., 10 calls in a day).

3.5 Evaluation

Implementation details We implemented a prototype of DyloChain using HLF v2.1 [3], written in Go [89]. The M-node is based on the codebase of Peer and Orderer, while the L-node utilizes the Peer codebase to implement the protocol. We made further optimizations, such as using an in-memory key-value store to manage state information and indexing proofs (i.e., π_L and π_M) in blocks to reduce lookup costs. A client is integrated with the HLF client SDK. Our prototype maintains compatibility with HLF in terms of smart contracts and client SDK. We utilize our customized implementation of a PBFT-based self-verifiable consensus protocol in each zone.

3.5.1 Experimental Setup

We conducted our experiments on an in-lab cluster consisting of 24 machines with three different configurations: 6 high-end machines with AMD Ryzen CPU 3990x (2.9GHz) and 256GB RAM, 3 medium-end machines with AMD Ryzen CPU 3970x and 128GB RAM, and 12 low-end machines with AMD Ryzen CPU 5950x (3.4GHz) and 32GB RAM. All machines feature Samsung SSD 970, run Ubuntu 20.04, and are connected via a 10 Gbps Ethernet network. Initially, we carried out a series of in-depth experiments on DyloChain using fifteen machines across three zones, and then we expanded to a large-scale experiment involving twenty-four machines with up to fifteen zones. Each zone consists of one M-node, four L-nodes, and four clients. In the three-zone experiments, each local chain was deployed on five machines: one M-node on a high-end machine, four L-nodes on two low-end machines, and four clients on two low-end machines. For large-scale experiments, we deployed two zones on each

high-end machine and one zone on each medium-end machine. We will use the terms zone and shard interchangeably as needed.

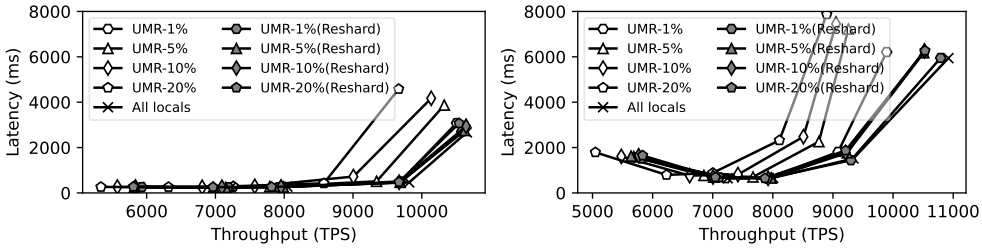
For evaluating DyloChain, we simulated a workload that exhibits dynamic locality using SmallBank [90]. Initially, we created 30,000 accounts in the SmallBank contract and distributed them equally across zones. Clients in each zone hold initial accounts of the same size and randomly invoke methods by selecting one or two users' accounts in their zone. Unless otherwise stated, we set a default value of batch size to 100. π_L has a size of 2.1 KB.

For generating global Txs, we controlled the percentage of mobile users, denoted by μ, by triggering mobility events at some time point t_M. For example, if μ is 5%, then randomly selected 5% users from each zone were moved to different random zones at the time point t_M, after which the client at a zone could possibly generate global Txs by selecting states of mobile users where their state ownership did not belong to their zone.

We demonstrate the relationship between user mobility ratio (UMR) and global Tx ratio (GTR). In SmallBank, two methods (i.e., send payment and amalgamate) access two states, while the other four methods access only one state. We used an implementation of SmallBank as in [91]. When the total number of users is n and the UMR is μ, the GTR is derived as follows.

$$\text{GTR}(\mu, n) = \frac{4}{6} \times \frac{\mu n}{\binom{n}{1}} + \frac{2}{6} \times \frac{\binom{\mu n}{2} + \binom{n}{1}\binom{\mu n}{1}}{\binom{n}{2}}$$

We have listed some of the GTR values for the UMRs used in the experiment when n is 10^3. When the μ values are 1%, 5%, 10%, and 20%, the corresponding GTR values are 1.3%, 6.5%, 13%, and 25.3%, respectively. For the next set of experiments, we triggered a globally occurring one-shot user mobility event at 20 seconds after starting Tx submissions, which lasted for 60 seconds in each experiment.



(a) Performance of Local Chain

(b) Performance of Main Chain

Figure 3.5: Overall Performance

3.5.2 Overall Performance

We first evaluated DyloChain's overall performance in terms of throughput and latency of local chain and main chain, respectively, under various UMRs with and without resharding support, as depicted in Fig. 3.5. We varied each client's send rate for issuing TXs at a rate from 600 to 1400 TXs per second with a step of 200, resulting in the system's total send rates ranging from 7200 to 16800. Then, we obtained the system's throughput and latency by calculating the average values from different total send rates, with each case running over 60-second intervals. We compared three cases in experiments: First, All-locals indicates that all user TXs are local, showing maximal performance over the other cases and used as a baseline for the other two cases. Second, UMR-μ% adds μ% mobile users to the system where μ is one of 1,5,10, and 20, without any reshard TX issued. Third, UMR-μ%(Reshard) allows users to issue reshard TX when they move to their destination zone, so the states of mobile users in each of the UMR-μ%, would be resharded appropriately.

In Fig. 3.5, it is clear that as UMR increases, the performance of both chains is noticeably saturated compared to All locals. As expected, constantly issued global TXs by mobile users degrade performance because there are many aborted TXs at the main chain due to interference effects and the state synchronization protocol. In contrast, the performance with reshard TX is much closer to those of All locals. This is attributed to the state reshard protocol that reduces interference effects, enabling

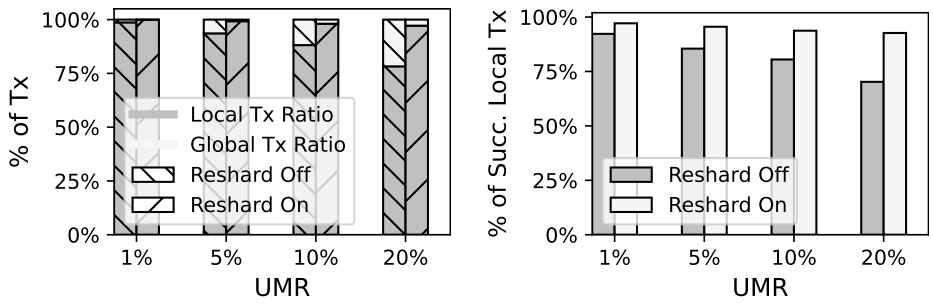
Table 3.1: Analysis of Resharding Effect on Main Chain in terms of Storage/CPU Cost

UMR	Size(Off) (avg / max / std)	Size(On) (avg / max / std)	Delay(Off) (avg / max / std)	Delay(On) (avg / max / std)
0%	2.75 / 2.75 / 0.00	2.75 / 2.75 / 0.00	3.93 / 4.43 / 0.65	3.93 / 4.43 / 0.65
1%	3.82 / 6.59 / 0.96	2.83 / 3.60 / 0.10	4.01 / 4.81 / 0.40	4.02 / 4.58 / 0.51
5%	6.41 / 18.21 / 3.81	3.54 / 20.12 / 2.76	4.68 / 5.37 / 0.76	4.11 / 4.62 / 0.76
10%	9.15 / 25.96 / 6.66	4.10 / 37.99 / 5.26	5.00 / 6.07 / 0.95	4.19 / 4.70 / 0.72
20%	15.02 / 40.77 / 11.38	5.45 / 57.90 / 10.12	5.94 / 7.53 / 1.56	4.54 / 5.22 / 0.67

mobile users to become local users by moving their states from remote source zones to local destination zones. Comparing both cases at a total send rate of 14400, the performance of UMR-20% without reshard TX decreased by 13.8%, while that with reshard TX only by 2.7%. Therefore, we can confirm that there is an improvement of 5.11x due to the effect of reshard TX. Note that, as shown in Fig. 3.5b, the latency initially decreases for the main chain when the total send rate is low. This occurs because the low volume of incoming TXs does not fulfill the generation rule for main blocks in a timely manner, resulting in new main blocks as well as their main block proof π_{MQ} taking relatively longer to be confirmed.

3.5.3 Analysis of Resharding Effect

We analyze the resharding effect to first observe how DyloChain minimizes global TXs under various UMRs. As shown in Fig. 3.6a, the percentage of local TXs in both cases decreases differently over the UMRs. Without resharding, the percentage of local TXs rapidly decreases from 98% to 78%. With resharding, however, the percentage of local TXs slowly decreases from 99.8% to 97.1%. The differences show that the resharding algorithm effectively minimizes global TXs by almost up to 7.58x for 20% mobile users. The minimized global TXs resulted in an increase in the success rate of local TXs committed to the main chain, as depicted in Fig. 3.6b.



(a) Minimizing global TX (b) Recovering Locality

Figure 3.6: Analysis of Resharding Effect

The second analysis of the resharding effect involved observing its effect on the main chain in terms of the proof's storage cost and the main blocks' processing cost. Because DyloChain uses the main chain to process global TXs and resharding, we need to observe how its overhead changes. Our measurements are shown in Table 3.1. We measured the average, maximum, and standard deviation for the size of π_M in KB. Apparently, as the ratio of UMR increases, the size of π_M also increases, regardless of resharding, because π_M carries information for processing global TXs. However, its trend differs depending on resharding. Across the range of UMRs, the max size (57.90 KB) of π_M with resharding is larger than the one (40.77 KB) without resharding because resharding burdens π_M during its execution. In comparison, the average size (5.45 KB) of π_M with resharding is three times smaller than the one (15.02 KB) without resharding. As resharding causes a temporary increase in the size of π_M, the completion of the protocol results in the creation of more local TXs that do not exert pressure on π_M. Next, we measure the processing delay of the main block by M-nodes in milliseconds (ms). The trend is the same that aforementioned. But, from the measurements, it is clear the costs are negligible, and the differences are not significant for processing the main block in terms of its time unit (ms).

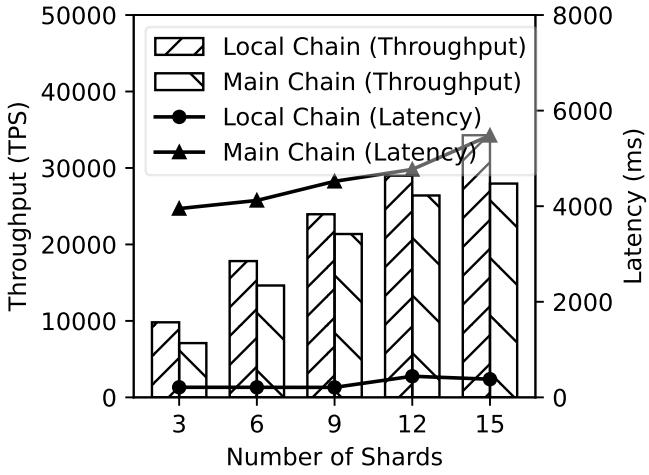


Figure 3.7: Scaling Shards

3.5.4 Scaling Shards

As DyloChain supports sharding, we experiment with it by scaling shards (i.e., zones). Fig. 3.7 shows performance scalability in the number of shards. In this experiment, we separately display the performance of the local chain and main chain to assess which component will be a bottleneck when DyloChain scales. We conduct experiments with all local TXs and depict data points right before it is saturated. Because DyloChain runs multiple independent local chains in parallel, the throughput of the local chain increases almost linearly to the number of shards, from 9.8k to 34k, with an average latency of 311ms. However, the performance of the main chain, which incorporates local blocks from zones, has a bottleneck at 12 shards with 26k of TPS and 4.7s of latency. The bottleneck may occur because the main chain has to process local blocks from all shards, and the performance of the main chain is bound by the capacity of the M-nodes responsible for processing these blocks. However, we found another factor behind this bottleneck, which we call the batch contention problem, which will be discussed in the next experiment.

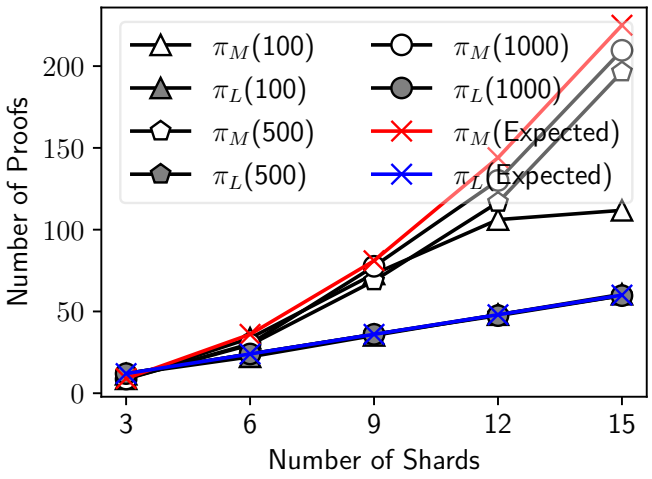
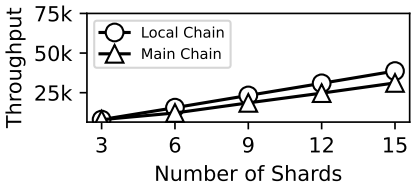


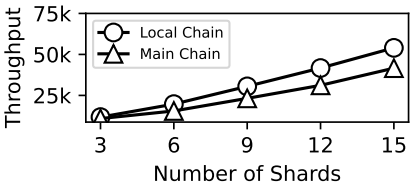
Figure 3.8: Batch Contention

3.5.5 Batch Contention

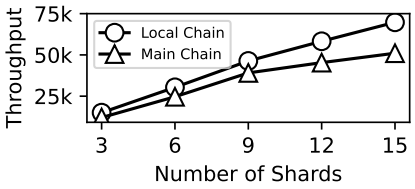
We analyze the cause for the early saturation of the main chain as in Fig. 3.8. In DyloChain with L shards, each M-node broadcasts π_M to L shards, so it is expected that there are L^2 of π_M in one main block, which is depicted in the figure as a red/blue line for π_M and π_L, respectively. However, the observed value of π_M when the batch size is 100 falls short of the expected number at 12 and 15 shards, which definitely delays gathering π_{MQ}. This is because the increased number of π_M per shard incurs high contention between a number of π_M and user TXs for being included in each local block. We call this *batch contention problem*. Meanwhile, π_L increases linearly with the number of shards due to the fixed number of L-nodes configured in each zone. If the number of L-nodes were to increase, we expect that the performance of the local chain will also be affected by the batch size. This implies that the self-verifiable features of DyloChain using the proof TXs incurs batch contention problem, which not only affects the performance of the main chain but also impacts the local chain as the system scales. Understanding and addressing the batch contention problem is



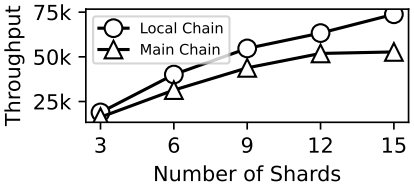
(a) batch=500, send-rate=1000



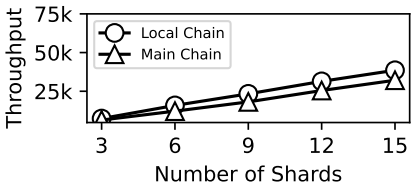
(b) batch=500, send-rate=1400



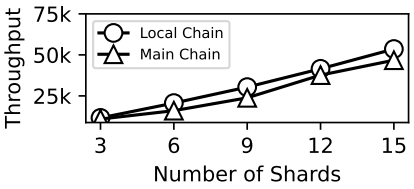
(c) batch=500, send-rate=1800



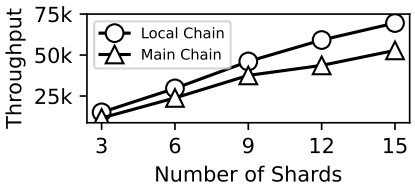
(d) batch=500, send-rate=2200



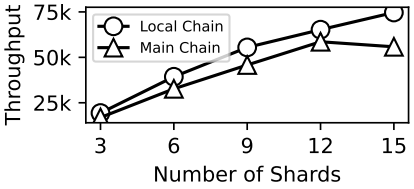
(e) batch=1000, send-rate=1000



(f) batch=1000, send-rate=1400



(g) batch=1000, send-rate=1800



(h) batch=1000, send-rate=2200

Figure 3.9: Performance Impact of Batch Size

crucial for improving the overall performance and scalability of the DyloChain system, particularly when dealing with a large number of shards and servers.

3.5.6 Impact of Batch Size

We now show the performance impact of batch size as seen in Fig. 3.9, which demonstrates throughput improvements across shards, with a batch size of 500 in Fig. 3.9(a) to Fig. 3.9(d) and 1000 in Fig. 3.9(e) to Fig. 3.9(h), for different client send-rates. Notably, our experiments with increasing the batch size show better performance than our previous one, wherein saturation was reached earlier with a batch size of 100 and a send-rate of 1000. Specifically, we measure the peak performance of the local chain and main chain as 74k and 55k, which are shown in Fig. 3.9h and Fig. 3.9g, respectively. As mentioned, this is because proofs can reach L-nodes faster owing to lowered batch contention. However, for send-rates over 1800, we observe that performance saturates with 12 shards. This is identified as a network bottleneck due to the increased size of each local block. In summary, increasing the batch size improves the performance of both the local chain and main chain by reducing the batch contention problem. However, there is still a saturation point at higher send-rates due to network limitations. To fully realize the potential of DyloChain, it is essential to find the right balance between batch size and send-rate while considering the network and processing capacities of the system.

3.5.7 Storage Overhead

Using DyloChain incurs additional storage overhead for the main chain and proofs. We measured the overhead according to the number of shards and batch size, as shown in Fig. 3.10. Apparently, the size of the main block that incorporates multiple local blocks from all shards is the largest overhead in DyloChain. Next, to quantify the storage overhead of proofs in DyloChain, we calculated the percentage of a main block occupied by a proof. It was confirmed that for a small batch size of 100, this value ex-

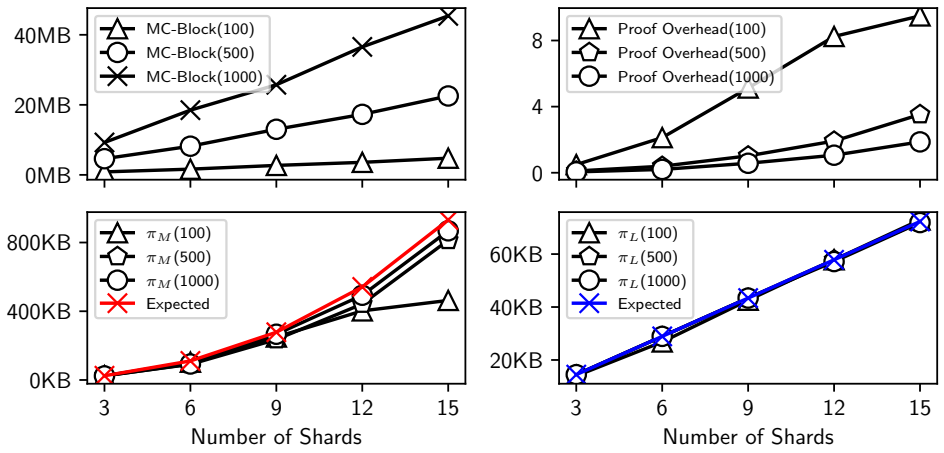


Figure 3.10: Storage Overhead of Main Chain and Proofs per Shards

ceeds 8%, but for larger batch sizes, it is below 4%. Furthermore, the size of the actual π_M and π_L approaches the expected size as the batch size increases. This observation suggests that increasing the batch size not only improves performance, as discussed earlier, but also reduces the storage overhead of proofs in DyloChain. As the batch size increases, the percentage of a main block occupied by a proof decreases, making the storage overhead more manageable.

3.5.8 Analysis of Storage Overhead

We formulate the amount of required storage overhead for maintaining a main chain, managed by M-nodes across zones. We first introduce notations as needed (summarized in Table 3.2). Without loss of generality, we make several assumptions. We assume that there is no batch contention, to ensure that π_Ms are persisted in every next block. This means that the size of the local batch only affects the number of user TXs in this analysis and each main block stores as many π_Ms as the square of the number of shards. Also, we assume that all user TXs have the same size and π_L, and the number of π_Ls in each block increases linearly with the number of L-nodes.

Since a main block is replicated across zones, Overhead(Main Block) $= h_m \times L \times$ Size(Main Block). A main block consists of its header and a set of local blocks, which is determined as $L \times$ Size(Local Block). Size(Local Block) is the sum over Overhead(UserTx), Overhead(π_M), and Overhead(π_L). The Overhead(UserTx) is Number(UserTx) X Size(UserTx), which can be written in $s_l \times s_u$. The Overhead(π_M) is Number(π_M) X Size(π_M), and Number(π_M) is L, so it can be written in $L \times s_\gamma$. The Overhead(π_L) is Number(π_L) X Size(π_L), and Number(π_L) is the same as the number of L-nodes; hence, it can be written in $R \times s_\alpha$. Combining these, we derive the Overhead(Main Block) as in Fig. 3.11. Most notably, the overhead of π_M increases proportionally to the cube of the number of shards, as it includes each local block header, is broadcasted to each zone, and the main block containing π_M is replicated across zones.

Table 3.2: Notations

Notation	Description
L	Number of shards
R	Number of L-nodes
s_h	Size of main block's header
s_l	Batch size of local block
s_u	Size of User TX
s_α	Size of π_L
s_γ	Size of π_M
h_m	Number of Main Blocks
h_l	Number of Local Blocks
Size(X)	Size of X
Number(X)	Number of X
Overhead(X)	Size(X) $\times$ Number(X)

Overhead(Main Block) $=$

$= h_m \times L \times (s_h + \text{Size(Main Body)})$

$= h_m \times L \times (s_h + L \times \text{Size(Local Block)})$

$= h_m \times L \times (s_h + L \times ((s_l s_u + L s_\gamma) + (R s_\alpha))))$

$= h_m \times (L s_h + L^2 s_l s_u + L^3 s_\gamma + L R s_\alpha)$

Figure 3.11: Analysis of Storage Overhead

3.5.9 Performance Comparison with HLF

We conducted a performance comparison of DyloChain as it scales up against Hyperledger Fabric (HLF), which adopts the X-O-V model, as shown in Fig. 3.12. We scaled DyloChain to 15 zones, and equivalently increased the number of nodes

in HLF, where L-nodes and M-nodes correspond to peers and orderers, respectively. For instance, HLF with 36 peers and 9 orderers is comparable to DyloChain with 9 zones, denoted by Local Chain-9 and Main Chain-9, with each zone containing 4 L-nodes and 1 M-node. Both DyloChain and HLF were configured with a batch size of 500, utilizing the SmallBank smart contract containing 30k accounts. The HLF orderer was deployed using Raft, the endorsement policy was set to any single peer, and each peer belonged to a different organization. DyloChain is configured for local TXs. The results show that DyloChain consistently outperforms HLF. Since HLF is non-sharded, all peers hold a full blockchain while validating all blocks from orderers, leading to higher overhead. In contrast, as the number of zones increases, DyloChain demonstrates enhanced scalability due to its sharding effect and hierarchically fashioned transaction processing.

3.5.10 Wide Area Network

We also conducted an experimental evaluation of DyloChain in a wide area network (WAN). We simulated WAN environment based on real-world cross-region latencies between Amazon Web Services (AWS) regions, as reported in [92]. We generated zone latencies accordingly and injected them into the inter-zone communication among M-nodes. We mapped each of DyloChain's 15 zones to AWS regions in the following order: Seoul, Tokyo, Hong Kong, Osaka, Singapore, Sydney, Frankfurt, London, N. California, Ireland, Mumbai, N. Virginia, Ohio, Oregon, and Stockholm. Note that cross-region latencies have an average of 140.84 ms, a minimum of 9.81 ms (Osaka to Tokyo), a maximum of 269.3 ms (Stockholm to Sydney), and a standard deviation of 70.65 ms. The results are shown in Fig. 3.12. When the number of zones is only three, the zones composed of relatively close distances (i.e., Seoul, Tokyo, and Hong Kong) exhibit minimal latency, resulting in no significant difference from the main chain performance in a local environment. However, as the number of zones increases, we observe that the main chain's performance becomes early satu-

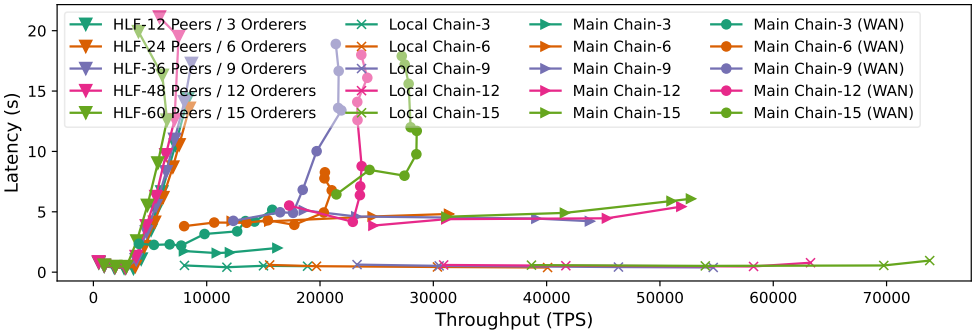


Figure 3.12: Performance Comparison of DyloChain with HLF, alongside experiments in wide area network (WAN)

rated. This occurs because the main chain requires the inclusion of local blocks from all zones in each round, and the heterogeneous latencies between zones from the WAN significantly degrade the main chain's performance. Nonetheless, it is evident that DyloChain still outperforms the traditional HLF due to its sharding and hierarchical architecture.

IV. PyloChain: A Sharding-based Hierarchical Blockchain over Partially Synchronous Network

In this chapter, we propose PyloChain, a sharding-based hierarchical blockchain that practically enhances DyloChain with DAG-based consensus and scheduling technique. Specifically, PyloChain employs a DAG-based mempool protocol [52], over partially synchronous network, to append highly concurrent local blocks from multiple sharding zones simultaneously in a leaderless manner, guaranteeing local block availability and higher throughput. Based on the DAG mempool, an efficient local BFT consensus produces a main block that contains totally orderd local blocks, by periodically interpreting local view of the DAG mempool.

Also, PyloChain enhances the order-execute-order-validate (O-X-O-V) transaction processing model with a simple scheduling technique. Although this model significantly reduces the burden of main block processing by allowing speculative parallel execution of local transactions across zones, it is not resilient to cross-shard transactions. PyloChain enhances cross-shard resilience with a simple scheduling technique in which cross-shard transactions are batch-processed at the end of each main block processing cycle, effectively improving transaction scalability. Furthermore, this scheduling accelerates main block processing by allowing parallel processing of local blocks from independent local chains, achieving greater scalability through the use of multiple processing threads.

4.1 Model and Assumptions

We assume a consortium blockchain where the identities of all members are known in advance, and clients submit transactions to a consortium member operating

a PyloChain server that they trust. We say a member is *Byzantine* if they arbitrarily deviates from the protocol, others we call them *honest*. As in the consortium blockchain, we assume that Byzantine members's attacks are infrequent. There exists f_L and f_F Byzantine failures out of $3f_L + 1$ local members within each zone and $3f_F + 1$ full members across zones, respectively, implying $3f_L + 2$ members in a zone. Note that there could be additional machines to be recruited upon detecting the failure of the full member in a zone for recovery. A zone could become completely unavailable (e.g., all the members within the zone crash due to a region-wide earthquake); yet the maximum number of availability failures is limited to f_F, and the probability of zone failure is mutually independent between zones. We assume cross-zone communication is limited to full members; local members do not communicate across zones.

We assumes a partially synchronous network where synchronous and asynchronous intervals alternate. During the asynchronous intervals, network partitioning is allowed, but this period only lasts for the global stabilization time (GST), denoted by Δ_{GST}. After this, a sufficiently long synchronous interval is maintained, ensuring that all message transmissions are eventually delivered within Δ_{Sync}. We cryptographic primitives used in PyloChain are not broken.

4.2 Overview

PyloChain consists of multiple independent zones within a hierarchical architecture. Within each zone, a lower-level sharded local chain is managed by local members, while a higher-level main chain across zones, managed by full members (also called zone leaders), as shown in Fig. 4.1.

A full member holds a full copy of all shards and maintains the higher-level DAG-based main chain. It is responsible for relaying protocol messages across the hierarchy, including bottom-up messages that relay local blocks with local certificates verifying their correctness to the main chain, and top-down messages that relay main block certificates with sync entries containing relevant states (e.g., from aborted local or

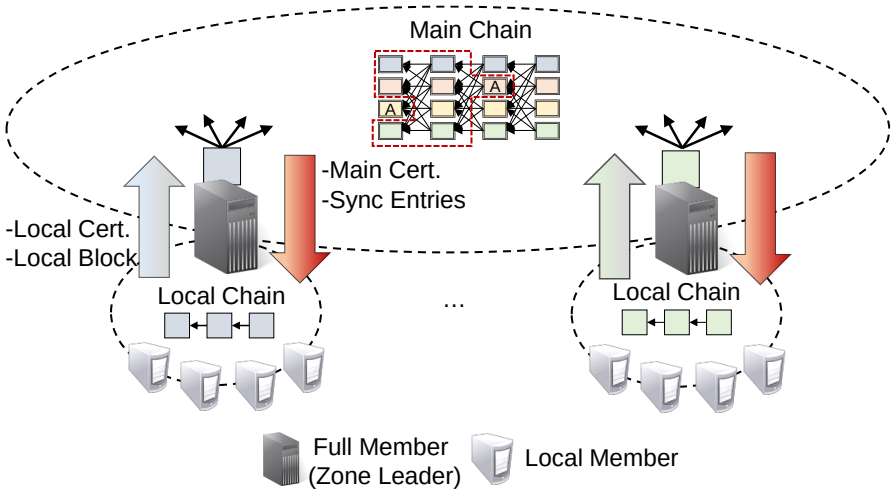


Figure 4.1: PyloChain: A sharding-based hierarchical blockchain

global transactions) to local members.

A local member maintains a single copy of a shard and, using a PBFT-like protocol with other local members in the same zone, reaches consensus on the local blocks proposed by the full member to produce a locally agreed-upon sequence. Local members can verify the state of local transactions but cannot verify global transactions. They also detect potentially faulty full members through a fine-grained auditing mechanism and, upon detecting failure, can replace the faulty full member with a newly recruited honest one.

Each user transaction includes read/write sets for a state database, where each entry contains a (key, value, version) tuple, with version defined by a block number and transaction offset [3]. Each state is assigned to a specific shard, determining the transaction type. As PyloChain uses the O-X-O-V model [54], it categorizes transactions into local (single shard) and global (multiple shards). Local transactions are executed in parallel across multiple local chains, speculatively updating local states, while global transactions are executed and validated only on the main chain. Global transactions may abort conflicting local transactions, known as interference [54]. The

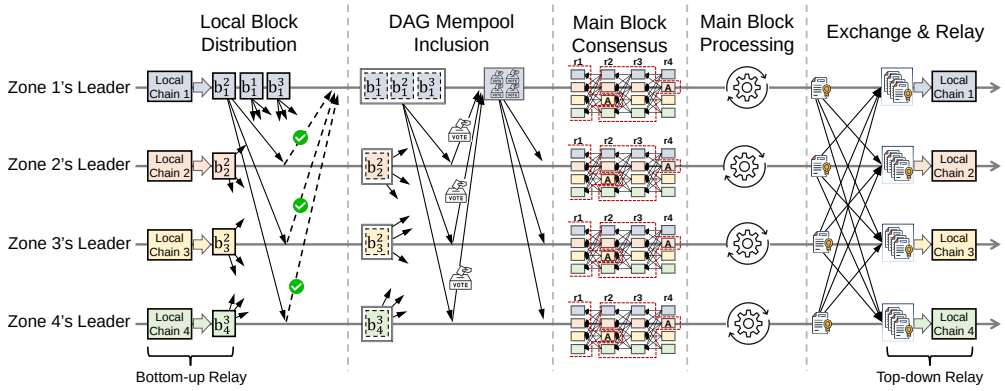


Figure 4.2: PyloChain: The Main Chain Protocol

full member includes aborted local and committed global transactions in state sync entries, relaying them with the main block certificate to the local chain, where local members commit the entries to the local chain.

4.3 Protocol Designs

In this section, we provide a detailed description of the protocol design of PyloChain.

4.3.1 Local Chain

In the same zone, the zone leader receives transaction requests from clients and categorizes each transaction by type based on the state it accesses: as a local transaction (accessing only a single local shard) or as a global transaction (accessing multiple shards or a remote shard). For simplicity, we assume that the zone leader can determine the transaction type in advance (e.g., through simulation or static analysis). For local transactions, the zone leader executes the transaction, calculates the read/write set, updates the local state speculatively, and generates a transaction to include in the next local block to be proposed. For global transactions, the zone leader simply includes them in the local block. When the zone leader broadcasts the local block to local

members in the same zone, the local members use their own local BFT consensus protocol (e.g., PBFT [25]) to verify the integrity of the zone leader's local block proposal. Specifically, they verify whether the zone leader has executed the client's transaction requests accurately and calculated a valid read/write set. If validated, each local member speculatively updates its corresponding local state and confirms the agreed-upon sequence of the proposed local block in their local chain by submitting endorsements, resulting in a local commit certificate for the local block proposal.

4.3.2 Main Chain

In Fig. 4.2, we demonstrate the main chain protocol of PyloChain.

Local Block Distribution Zone leaders collect commit certificates for their local blocks and then reliably broadcast these blocks to other zone leaders, guaranteeing local block availability by receiving $2f_F + 1$ acknowledgments from other zone leaders. Specifically, zone leaders effectively leverage parallel workers architecture [52] to distribute local blocks concurrently. This concurrent distribution of local blocks provides performance benefits by utilizing maximal network bandwidth; however, it is highly likely not to preserve the locally agreed sequence of the local blocks in the final consensus order. For example, if there are two local blocks, b_1^1 and b_1^2, from zone 1, b_1^2 might complete its broadcast before b_1^1 due to network conditions. As a result, b_1^2 can appear before b_1^1 in the final order, which violates the locally agreed sequence. We address this issue in the next DAG mempool inclusion step.

DAG Mempool Inclusion The purpose of this step is to include the digest of $2f_F+1$ availability-guaranteed local blocks, into a vertex, and incorporate it into the DAG mempool. This step consists of two main operations: First, Local order-aware vertex creation: the vertex to be proposed is created by each zone leader, in a way that it maintains the locally agreed sequence of concurrently broadcasted local blocks. For this, each zone leader proposes the local block by matching its number in one-to-

one correspondence with a monotonically increasing round number as defined by the DAG mempool. This ensures that even if local blocks are completed in different orders during the local block distribution phase, their local sequence is preserved in the DAG mempool seamlessly, with no gaps. Second, DAG mempool inclusion: The zone leader proposes the created vertex to other zone leaders and obtains $2f_F + 1$ votes to finally produce a DAG mempool inclusion certificate. This certificate is then rebroadcast so that other zone leaders incorporate the vertex into their respective local views of the DAG mempool. For details on the process, including round advance rule, voting rule and commit rule, refer to [52, 53].

Note that although proposing only one local block per round is conceptually simple, but it may lead to performance issues, as the local block corresponding to a particular round number might be completed much later, while subsequent local blocks complete much earlier. In such cases, idle time increases, and communication overhead can surge dramatically. To address this, we define the maximum number of local blocks that can be included in a single vertex, parameterized by B_p, allowing multiple local blocks to be proposed simultaneously, thereby resolving this issue. Since each vertex proposal only includes a list of small-sized block digests, the performance penalty in terms of network bandwidth is negligible.

Main Block Consensus For local blocks included in the DAG mempool, full members periodically apply a locally executed BFT algorithm, reaching consensus on a committed sub-DAG, i.e., main block. To achieve this, we follow the partially synchronous Bullshark [53, 93] protocol, summarized as follows: In every even-numbered round, each full member selects a predefined anchor vertex (e.g., in a round-robin manner) and includes in the sub-DAG all vertices within the causal history of the anchor, specifically those between the current anchor and the previous anchor vertex. The vertices included in the sub-DAG are then arranged according to some deterministic topological ordering logic (e.g., depth-first search) to establish a total order across local chains. For example, in Fig. 4.2, the anchor vertices are denoted by "A" in the

second and fourth rounds. The vertices between these two anchors form the sub-DAG, indicated by the red dotted line. The main block, which includes all vertices in the sub-DAG, outputs the corresponding local blocks' digests, the block number, its block hash, and the previous hash. In summary, by performing consensus locally and using a single certificate for the anchor block to commit multiple mempool blocks—i.e., all of its causal histories—the main chain can produce a communication-efficient and high-throughput main block.

Note that, because consensus is performed only locally by interpreting each full member's local view of the DAG mempool data structure, inconsistent sub-DAGs may be created from different DAG views. However, the main chain guarantees that all honest full members can generate a consistent main block by relying on Bullshark's anchor ordering rule with the $f_F + 1$ commit rule. The specific mechanisms for this, along with other algorithm-specific challenges such as garbage collection and fairness, are discussed in [53].

Main Block Processing After the main block consensus outputs a main block, i.e., an ordered list of local blocks from the committed sub-DAG, the full member starts to process all transactions within the main block. For processing main blocks, the zone leader maintains the main-level state DB and version map [54], which enables consistent validation across local chains. For local transactions, since they have already been executed in the local chain, they contain a read/write set. The zone leader identifies this read/write set and performs validation based on multi-version concurrency control (MVCC) [3]. If validation succeeds, the write set is applied to both the state DB and the version map. For global transactions, they are executed in the order determined by the main chain, calculating their read/write sets, which are then applied to the state DB as well as the version map. For the aborted local transactions and all global transactions, their latest main states are appended to the sync entries after processing the main block.

Local transactions that had been speculatively executed may be aborted by global

transactions on the main chain. Worse, the aborted local transactions can further cause cascading aborts of subsequent local transactions. This can introduce overhead by increasing the size of the sync entries and significantly reducing the number of meaningful committed transactions.

To address this, PyloChain applies a simple yet effective scheduling technique: The full member filters global transactions to process all local transactions first, before handling the global transactions during main block processing. This ensures that global transactions within a single main block do not interfere with local transactions in the same main block. We consider this effect quite significant, as a DAG-based main block—with its higher concurrency and throughput—can include a very large volume of transactions from multiple local chains, thereby avoiding many local transactions being aborted by global transactions.

Additionally, this scheduling technique offers an opportunity to process mutually independent local blocks from different local chains in parallel, as all global transactions are filtered to the end. This can boost the processing speed of each main block, especially as the number of zones increases. For comparison, we illustrate our processing methods in Fig. 4.3, showing the DAG-only PyloChain and the scheduling-enhanced version of PyloChain, denoted as PyloChain(DAG) and PyloChain(DAG+Sched), respectively, alongside a naive baseline approach that simply collects a fixed number of local blocks from each local chain (e.g., DyloChain [54]).

Exchange and Relay Afterward, each zone leader generates a main block certificate for the main block, summarizing the results of main block processing, and broadcasts it among themselves to collect $f_F + 1$ certificates. The main block certificate includes the main block number, main block hash, previous main block hash, the sequence of local block headers within the main block, the identity of the creator, and the digest values of the sync entries that need to be committed to each local chain. Each sync entry consists of a key, value, and transaction ID. Once $f_F + 1$ main block certificates are gathered, each zone leader propagates the main block certificates along with the

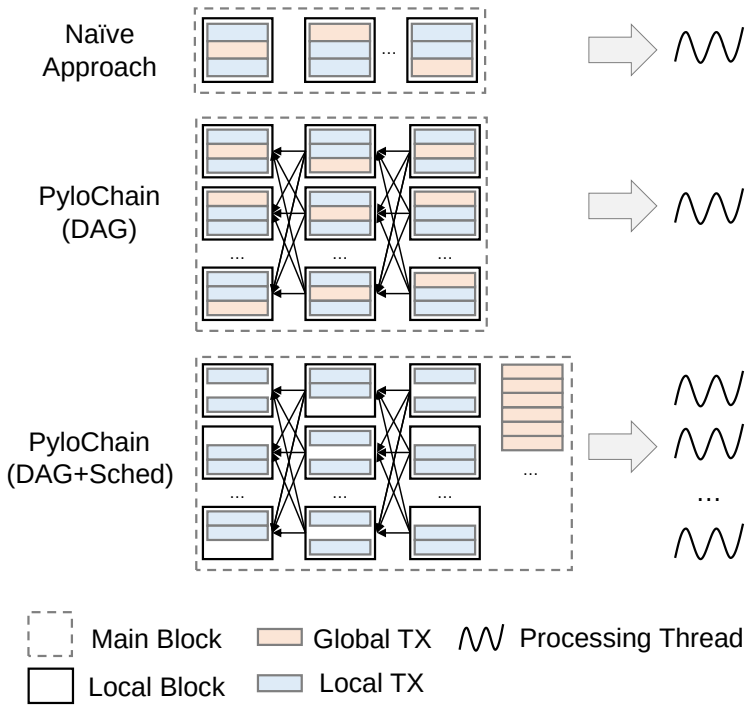


Figure 4.3: Comparisons of Main Block Processing

payloads of the sync entries, to their respective local chain. Then, the local members verify the main block certificates and sync entries, allowing them to reliably confirm that the local blocks from their local chain have been committed to the main chain. Additionally, they safely synchronize the states of aborted local transactions and global transactions to the local chain using the payloads contained in the sync entries.

4.3.3 Auditing Zone Leader's Trustworthiness

The fact that a zone leader relays important protocol messages between a lower-level local chain (i.e., within a zone) and the higher-level (i.e., across zones) main chain poses significant security challenges for PyloChain, because a zone leader could act honestly within the local chain internally while behaving maliciously in cross-zone communications externally, deceiving participants in either direction. We classify

these malicious relay attack behaviors into two directions. First, the bottom-up relay attack; a zone leader may fail to broadcast the lower-level local blocks to other zones not to guarantee their availability, or broadcast a malicious local block. Second, the top-down relay attack; a zone leader may not deliver the higher-level main chain block processing results to the local chain or may report fake results.

Among these attacks, we can simply guarantee the integrity of the data relayed by zone leaders, as the local blocks and main block processing results each contain $2f_L + 1$ signatures from local members and $2f_F + 1$ signatures from full members, respectively. Additionally, although a zone leader may include local blocks in the DAG mempool in an order that does not respect the locally agreed-upon sequence, local members can detect this by verifying the order of local block headers within the subsequently created main certificate. Note that, for simplicity, we do not consider the time cost involved in local computation in this problem. Furthermore, we assume that, once the GST (Global Stabilization Time) arrives, a sufficiently long period remains.

Timing-related attacks, such as delaying or dropping relayed messages, need to be carefully addressed. Local members, after confirming a local block commitment on their local chain, need to ensure it is committed on the main chain in a timely manner, as indicated by the main certificate delivered by the full member. The issue is that, since the main chain operates in a partially synchronous network and is collectively managed by multiple full members, local members do not have visibility into the main chain's status. As a result, when a timing-related issue arises, local members might question, "Is this delay due to the GST period, or is it the fault of our zone leader?" Therefore, the full member needs to provide proof to the local members that it is behaving correctly.

To achieve this, PyloChain exploits the fine-grained semantics of the main chain, externalizing to local members the certificates corresponding to its operations, including local block availability, DAG mempool inclusion, main block consensus, main block processing certificates. For local block availability certificate, after broadcasting the local block, the full member collects $2f_F + 1$ acknowledgments from other full

members and delivers them to the local members, a process that takes $3\Delta_{Sync}$. For DAG mempool inclusion certificate, the full member secures the round number, block digest, and $2f_F + 1$ votes from other full members through a round-based local block proposal, confirming the reliable inclusion in the DAG mempool, which also takes $3\Delta_{Sync}$. For main block consensus certificate, every two rounds, the full member creates a main block locally using the partially synchronous Bullshark protocol [53, 93], which requires $4\Delta_{Sync}$. For main block processing certificate, the full member gathers $2f_F + 1$ certificates summarizing the main block's processing results and delivers them to the local members, taking $2\Delta_{Sync}$. Since both the main block consensus and processing steps are performed locally, their results can be combined and delivered in a single batch, which is expected to take $6\Delta_{Sync}$ overall.

This allows local members to follow up on the status of their local blocks in the main chain during a normal synchronous period. If they do not receive the expected certificate within the anticipated delay, they may question, "Is this delay due to being in the GST period, or is it due to our zone leader's fault?" To address this, local members can conservatively wait up to Δ_{GST} for the message to arrive. If it still does not arrive by then, they can conclude a fault by the zone leader, or, more proactively, they may determine a fault as soon as the synchronous timer expires. In both cases, this triggers a leader fault response (e.g., a PBFT-like view change). In PyloChain, we assume that full members are reputable service providers who generally operate correctly, so we adhere to the former, more conservative approach.

We describe the behavior of PyloChain during the GST period in two cases. First, if messages are received within the GST period, the local chain records pending blocks, and the main chain processes these blocks according to the DAG BFT protocol. Once the GST period ends, all pending local blocks receive the main certificate and are recorded, after which normal operations resume. Second, if messages are not received even after the GST period ends, the local chain still records the pending blocks. However, after the GST period, a leader change is initiated. Subsequently, all pending local blocks receive the main certificate from the new leader, allowing normal operations to

resume.

4.3.4 Availability and Recoverability

Since PyloChain adopts a balanced sharding scheme, local chains are replicated across multiple zones with full members, unlike traditional complete sharding schemes. Therefore, even if one zone entirely fails, for instance due to a region-wide earthquake, clients can move to another zone and continue their business there because the full member in the new zone can still handle requests from clients in the failed zone. Note that PyloChain can leverage existing dynamic resharding states [54, 26] to improve efficiency. While this scenario can provide availability during zone failures, it may lead to a performance penalty due to a potentially heavier or imbalanced workload over time. Additionally, a malicious full member detected by local members through the fine-grained auditing mechanism needs to be recovered to ensure the liveness of locally agreed-upon client-submitted transactions.

Therefore, it is necessary to timely recover a faulty zone or faulty full member to ensure it resumes correct behavior, thereby guaranteeing the liveness of submitted transactions within the faulty zone. To recover a faulty zone or full member, PyloChain first excludes the faulty zone's full member from the main chain protocol. A new full member joining the consortium group must receive offline permission from consortium members before joining. After that, the new full member participates in the main chain protocol and restores the main chain by contacting an existing honest full member. The new local members can recover their local chain from the full member in the same zone. PyloChain can leverage dynamic BFT [94], which provides dynamic membership with a recovery protocol under partially synchronous BFT, to remove a malicious full member from the consortium group and replace it with a new honest full member. We leave the detailed mechanism for designing the recovery process as future work.

4.4 Analysis

We define one liveness and two safety properties of PyloChain in the context of a sharding-based hierarchical blockchain for proving its correctness. We assume the BFT algorithms operating within each local chain and the main chain are correct. Additionally, for simplicity in proving correctness, we address a full member failure instead of an entire zone failure. We denote the i-th agreed-upon local block from the x-th local chain as b_x^i. Then, we define the following properties that PyloChain should guarantee:

- **Validity** If a local block is agreed upon in a local chain, then that local block will eventually be confirmed on the main chain.
- **Bottom-up Consistency** If two local blocks b_x^i and b_x^j are agreed upon within the same x-th local chain with $i < j$, then b_x^i is ordered before b_x^j within the main chain.
- **Top-down Consistency** A local chain does not finalize any transactions that are inconsistent with the processing results on the main chain.

We then prove that PyloChain guarantees Validity, Bottom-up Consistency, and Top-down Consistency.

Theorem 4. *PyloChain guarantees Validity.*

Proof. Assume that a local block is agreed upon in a local chain and its full member is malicious enough not to follow the main chain protocol. Then, the malicious full member will cause a failure in at least one phase of the main chain protocol for that local block. This failure will be detected by a fine-grained auditing mechanism carried out by local members from the same local chain. Specifically, the malicious full member might fail to deliver the local block availability certificate within $3\Delta_{Sync}$ during the local block distribution phase, or fail to deliver the DAG mempool inclusion certificate within $3\Delta_{Sync}$ during the DAG mempool inclusion phase, or fail to deliver the main

block certificate (including its consensus and processing) within $6\Delta_{Sync}$. If a failure occurs in any of these phases, the local members record the local block as pending, start a GST timer, and, upon expiration, the local members recover the full member through a PBFT-like view change. Then, a new honest full member will execute the main chain protocol again on the pending local block, thus guaranteeing the validity of PyloChain. □

Theorem 5. *PyloChain guarantees Bottom-up Consistency.*

Proof. We first mention that PyloChain is resilient to a full member's fork attack—that is, generating different local blocks for the same sequence number. This is because, in the main chain protocol, full members consider only local blocks with exactly $2f_L + 1$ local certificates from other full members to be valid. As a result, maliciously forked local blocks are ignored by honest full members in the main chain protocol. Let b_x^i and b_x^j be two agreed-upon blocks from the x-th local chain, where $i < j$. These blocks can be safely submitted to the main chain, and during the local block distribution phase, they can be completed in any order. However, in the DAG mempool inclusion phase, an honest full member in the x-th local chain proposes the digest of b_x^i first, followed by b_x^j. If a malicious full member violates this order—e.g., by proposing b_x^j before b_x^i—the other honest full members can easily detect this by checking its monotonically increasing round number and its block number and will refrain from voting. As a result, the malicious full member will fail to deliver the DAG mempool inclusion certificate to its local members within the expected time. Eventually, this violation will be detected by a fine-grained auditing mechanism and, after GST, the malicious full member will be replaced. The newly recovered honest full member will then re-propose those pending local blocks, ensuring that PyloChain guarantees Bottom-up Consistency.

□

Theorem 6. *PyloChain guarantees Top-down Consistency.*

Proof. Assume that a global TX has been committed on the main chain. Then, all local TXs that were issued on the involved local chains, until the sync entry of this

global TX is committed to those chains, are subject to being aborted. There are two cases. First, there may be subsequent local TXs included in some main blocks, which are dependent on that global TX, are doomed to be aborted. Second, there may be inflight local TXs that are speculatively executed on the local chain, updating the local states and preparing to be included in a main block. Despite these potential inconsistencies, PyloChain ultimately finalizes all TXs on the main chain. Additionally, since involved local chains finalize their local TXs only if the main block processing certificate is confirmed, if a speculatively executed local TX is aborted on the main chain, it deterministically synchronizes the latest main states based on the sync entries of the aborted local states. Consequently, a local chain never finalizes a result that contradicts the main chain. If a full member is malicious and incorrectly processes a main block, this can be validated through the main block processing certificate, which contains the execution results from the other $2f_F + 1$ honest full members. Therefore, PyloChain guarantees Top-down Consistency.

□

4.5 Evaluation

To evaluate the feasibility of the PyloChain designs, we prototyped PyloChain in GoLang [89] and integrated it into Hyperledger Fabric [55] to support smart contracts, incorporating necessary modifications for the PyloChain implementation. For DAG BFT, we utilize the implementation of Narwhal/Bullshark [57] written in Rust [95], and extend it to support the main chain of PyloChain. Narwhal/Bullshark is comprised of two components: Worker and Primary. The Worker is responsible for receiving local blocks from each full member and broadcast the local block between Workers. The number of workers per full member is configurable, and the workers in each full member are assigned unique indices. Workers communicate only with those in other full members that have the same index. We modified the Worker so that it exports an event when a proof of local block availability (i.e., an $2f_F + 1$ certificates) for a new local block is collected. The Primary exists for each full member, receives

local block digests from its workers, and finalizes the order of local blocks using local consensus logic based on Bullshark. Note that we use a partially synchronous version of Bullshark [53] for the consensus logic, rather than its fully asynchronous version, to match the network assumptions of PyloChain. Each Primary then exports its DAG mempool inclusion and main block consensus results in the form of ordered digests, i.e., a topologically sorted sub-DAG, to its full member.

We also implemented other sharding schemes, i.e., availability sharding and performance sharding, within our environment for comparison with PyloChain. For availability sharding, we simply configured full members to propagate the collected local blocks from the DAG mempool to their respective local members within the same zone. For performance sharding, we removed the main chain and implemented a 2PC protocol to handle cross-shard transactions instead. In the 2PC protocol, a designated full member acts as the global coordinator. All cross-shard transactions issued within zones are forwarded to this coordinator, which then carries out the 2PC protocol, i.e., the prepare phase for locking and aggregating the involved states, and the commit phase for finalizing the transaction (i.e., commit or abort) and delivering the finalized results to the involved local chains.

4.5.1 Experimental Setup

We conducted our experiments on an in-lab cluster consisting of 24 machines with three different configurations: 6 high-end machines with AMD Ryzen CPU 3990x (2.9GHz) and 256GB RAM, 3 medium-end machines with AMD Ryzen CPU 3970x and 128GB RAM, and 12 low-end machines with AMD Ryzen CPU 5950x (3.4GHz) and 32GB RAM. All machines feature Samsung SSD 970, run Ubuntu 20.04, and are connected via a 10 Gbps Ethernet network. We evaluated PyloChain in an environment with up to 18 zones, where in each zone, we deployed one full member, four local members, and four clients, which sums up to a total of 18 full members for the main chain, and 72 local members for 18 local chains, respectively. All members in

our experiment run as Docker containers [96], orchestrated by Docker Swarm, and deployed on its overlay network.

We set up a micro-payment application using SmallBank [58] with 300k users equally distributed and statically initialized for each zone. The average size of each transaction and a local block are 2.89KB and 1.4MB, respectively. The experimental parameters of the main chain using DAG BFT [57] are as follows: each worker batch is set to include one local block. A primary can propose up to 40 local block digests per vertex, i.e., B_p is 40. The maximum delay for a primary to propose a vertex is 800ms. We measured PyloChain's throughput and latency on the client side, with each client asynchronously submitting transaction requests at a send rate ranging from 10k to 110k, in steps of 10k, evenly distributed across the clients. Latency was measured as the time between the client submitting a transaction and receiving a commit event from the local members, while throughput was calculated by summing the number of transactions committed per second as measured by all clients.

4.5.2 Overall Performance

We demonstrate PyloChain's overall performance scalability in terms of throughput and latency, based on the number of zones (rows) and the percentage of global transactions (columns), by comparing PyloChain with other approaches, such as performance sharding and availability sharding, as shown in 4.4. Additionally, we differentiate between two versions of PyloChain—one that applies only the DAG-based BFT consensus, denoted as PyloChain(DAG), and another that includes both the DAG-based BFT and the scheduling technique, denoted as PyloChain(DAG+Sched)—to clearly understand the effects of each technique. In the figure, each subplot legend is labeled with the number of zones and the global transaction percentage. For example, "12 / 40%" means that the experiment is conducted with 12 zones and 40% global transactions.

We first compare DyloChain and PyloChain, followed by a comparison of Py-

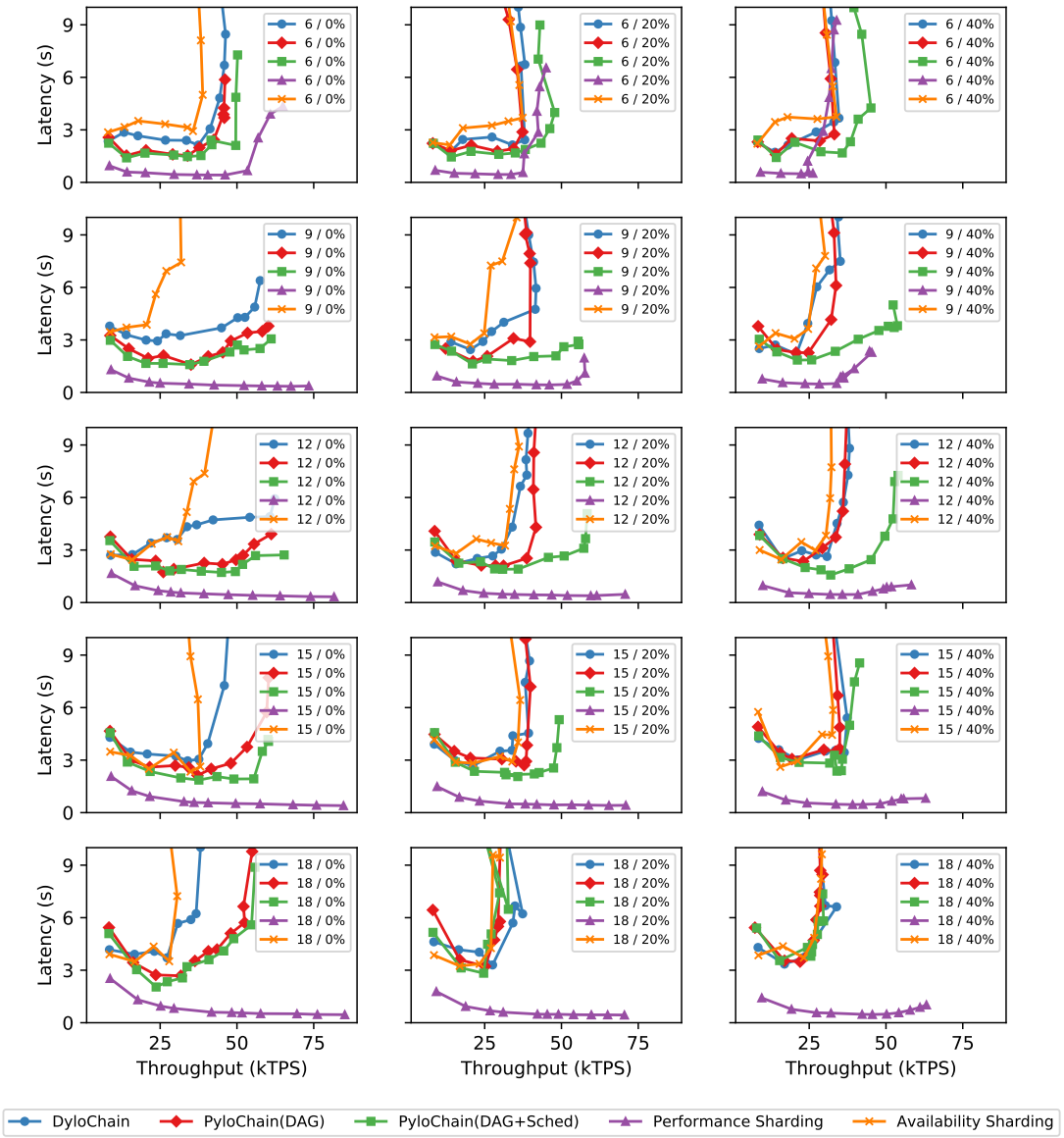


Figure 4.4: Overall Performance of PyloChain: Each graph represents the performance in terms of throughput and latency, according to the number of zones and the percentage of global transactions, denoted by #Zones / %GlobalTX

loChain with other sharding schemes.

Zone Scalability

We focus on the case of all local workloads (the first column). Notably, PyloChain(DAG) consistently exhibits better performance, especially in terms of latency, compared to DyloChain across different numbers of zones. For example, with 9 zones, PyloChain(DAG) achieves an average latency of 2.67s, which is 1.47 times faster than DyloChain's 3.92s. Meanwhile, the average throughput is only slightly better, with PyloChain(DAG) at 37,432 TPS and DyloChain at 35,148 TPS, respectively. At higher zones (18 zones), the performance gap between the two systems becomes more pronounced. PyloChain(DAG) clearly outperforms DyloChain, achieving an average TPS of 37,090 and maintaining an average latency of 4.84s, whereas DyloChain reaches only 30,672 TPS with a latency of 8.13s.

The performance difference stems from the distinct approaches to main block consensus. DyloChain's simpler main block creation rule requires a fixed number of local blocks, with the same block number from each local chain, to be synchronously included in a main block. This approach can lead to higher latency and lower throughput as the number of zones scales. In contrast, PyloChain(DAG) employs DAG-based consensus to asynchronously collect local blocks for main block creation, enabling it to flexibly include more blocks based on varying workloads, thereby improving scalability and efficiency. Additionally, as the number of zones increases, both systems initially exhibit improved performance due to greater workload distribution and parallelism, achieving peak performance at 12 zones. However, starting from 15 zones, performance begins to decline. This decline is attributed to the overhead associated with the main chain's higher volume of local block distributions, which exhausts network bandwidth resources, and the increased main block processing delay due to its larger size.

Note that PyloChain(DAG+Sched) shows similar performance to PyloChain(DAG) or only slightly better under zero global transactions. This outcome may seem coun-

terintuitive since PyloChain(DAG+Sched) includes parallel processing of local blocks with multiple threads, whereas PyloChain(DAG) uses only a single processing thread. This observation suggests that validating local transactions, which have already been executed on parallel local chains, incurs minimal overhead thanks to the O-X-O-V processing model. Also, note that at higher zones with a low send rate, latency tends to be higher. This occurs because the configured send rate is divided among all clients in the system, causing each client's send rate to be too low to fill the local block batch size (up to 500 transactions per local block) on the local chain in a timely manner.

Global Transaction Scalability

Now, we describe the scalability of PyloChain under global transaction rates of 20% and 40%, as shown in the second and third columns of Fig. **??**. Obviously, all systems experience earlier saturation compared to all-local workload scenarios because higher percentages of global transactions impose additional burdens on main block processing. These burdens include the execution of global transactions' smart contracts, interference with conflicting local transactions, and the overhead of the state synchronization protocol. We examine the scheduling effects of PyloChain(DAG+Sched), which is observed to outperform the other two systems in all cases. For example, with 12 zones and 20% global transactions, PyloChain(DAG+Sched) exhibits an average of 37,642 TPS and 2.80s latency. This represents approximately 1.17x higher throughput and 1.83x faster latency compared to PyloChain(DAG)'s 32,212 TPS and 5.13s, and 1.25x higher throughput and 1.95x faster latency compared to DyloChain's 30,169 TPS and 5.47s. At a send rate of 90,000, PyloChain(DAG+Sched) achieves 57,263 TPS with 3.10s latency, offering 1.39x and 1.49x higher throughput, and 2.76x and 2.63x faster speed compared to PyloChain(DAG) and DyloChain, respectively. The PyloChain(DAG+Sched)'s improvement is due to the simple within-a-main-block scheduling technique, which processes global transactions at the end of the main block, thereby significantly reducing interferences with local transactions within the same main block. As a result, there is less overhead in state synchronization, lead-

ing to more efficient performance.

Comparisons with Other Sharding Schemes

By comparing PyloChain, which employs balanced sharding, with other approaches such as performance sharding and availability sharding, we find that balanced sharding shows better performance than availability sharding but is less performant than performance sharding. In availability sharding, since all local members receive local blocks from their respective full members each time, we observe a considerable overhead. On the other hand, in the case of performance sharding, the absence of a main chain allows for horizontal scalability with an increasing number of shards, particularly in all-locals workloads. Even in the presence of global transactions, the 2PC protocol in performance sharding tends to incur significantly lower overhead compared to the main chain-related overhead in balanced sharding. This is because the cross-shard protocol is initiated earlier in performance sharding than in balanced sharding. Specifically, 2PC is handled immediately by a global coordinator as soon as a global transaction occurs, whereas PyloChain experiences longer latency until a global transaction appears on the main chain. Additionally, the throughput is constrained by the total network bandwidth capacity, and since PyloChain's main chain requires all-to-all broadcasting of local blocks across zones for availability, its zone scalability is shown to be lower compared to performance sharding.

4.5.3 Interference Analysis

To better understand system interference and scheduling effects, we analyzed the abort ratio of local transactions under 15 zones, 40% global transactions, and a send rate of 80k, as shown in 4.5. PyloChain(DAG) showed higher interference than DyloChain due to larger main blocks with more locally dependent transactions. However, PyloChain(DAG+Sched) significantly reduced interference through global transaction scheduling. For instance, with 300k accounts, PyloChain(DAG+Sched) achieved abort

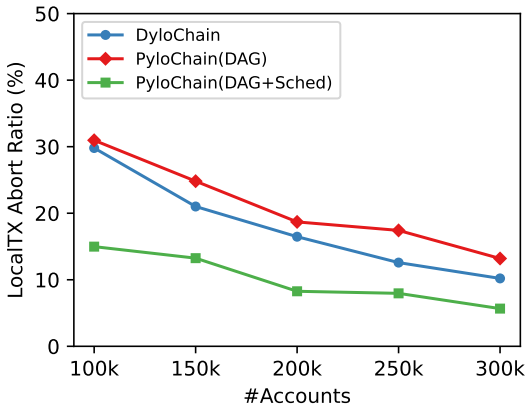


Figure 4.5: Interference Analysis

ratios of 10.20%, 13.19%, and 5.66%, 1.8x and 2.3x lower than DyloChain and PyloChain(DAG), respectively.

4.5.4 Storage Cost

In Fig. 4.6, we compare the storage consumption of the balanced approach (i.e., PyloChain) against other approaches, measured in gigabytes (GB). In this experiment, clients in each zone submitted transactions at a rate of 4000 per seconds, with all transactions being local, and the experiment lasting one minute. As expected, storage costs were highest for availability sharding (up to 457 GB), as local members, along with full members, also store local blocks across zones. Balanced sharding, which requires only full members to store local blocks across zones (up to 122 GB), was closer to the performance of performance sharding, where each member holds only their respective local chain (up to 30.7 GB).

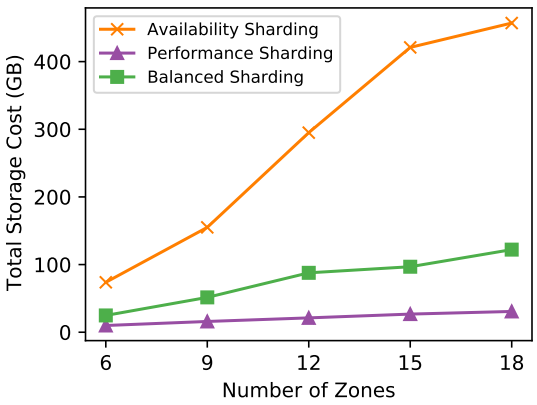


Figure 4.6: Storage Cost

V. Discussions and Future Works

Distributed Network Assumptions In a network with adversarial (i.e., Byzantine) members, achieving consensus among correct members depends on the network communication assumptions, leading to different theoretical results. As famously described in the Byzantine Generals Problem [97], where generals must consistently decide whether to attack or retreat without tampered or forged messages, network communication models are broadly classified into synchronous (or reliable) and asynchronous (or unreliable) networks. When the total number of members is N, the Byzantine resilience of consensus algorithms is $(N-1)/2$ for synchronous networks and $(N-1)/3$ for asynchronous networks. Synchronous networks provide higher resilience and allow for much simpler algorithm designs. However, a practical challenge for such algorithms is the need to precisely predict message transmission timeouts.

For a long time, synchronous algorithms were studied purely in theory. However, since around 2010, sophisticated delay prediction techniques capable of guaranteeing high accuracy in environments such as data centers have been developed [98, 99]. These advancements have brought renewed attention to the practicality of synchronous networks. Leveraging synchronous network assumptions has enabled the design of Byzantine fault-tolerant algorithms that are significantly more efficient and practical compared to traditional algorithms that assume a strong adversary fully controlling network communications [100, 101, 102, 103].

Following this trend, DyloChain exploits the advantages of synchronous networks for its main chain, allowing for simple designs and higher resilience.

However, in many real-world scenarios, leveraging sophisticated latency techniques can be challenging. For example, if the communication infrastructure used by DyloChain relies on an underlying gossip network, predicting end-to-end message delays becomes extremely difficult. Similarly, edge computers deployed at base stations

may face limitations in network resources, making communication unreliable. In such environments, asynchronous network assumptions become more relevant. Unfortunately, in asynchronous environments, consensus is theoretically impossible, as stated by the FLP impossibility theorem [104].

One way to circumvent this limitation is by introducing asynchronous periods only partially, leading to what is known as a partially synchronous network. In such a network, synchronous and asynchronous periods alternate. The asynchronous period persists for a duration known as the global stabilization time (GST) and eventually transitions into a synchronous period. In a partially synchronous network, it is guaranteed that messages sent will eventually be delivered to their recipients.

Due to its better practicality, many deployable Byzantine fault-tolerant algorithms have been developed for partially synchronous networks [25, 105, 106, 107, 108, 109]. Based on these assumptions, PyloChain redesigns its main chain, opting for a DAG-based leaderless approach among various possible designs for partially synchronous networks, thereby exhibiting practicality.

Sharding-based Hierarchical Blockchain and Edge Computing With the advancement of 5G/6G communication technologies, edge computing has become essential for mission-critical services that require high security and performance. Edge computing typically exhibits the following characteristics: (1) Multi-level hierarchical infrastructure, spanning from traditional large-scale cloud servers at the higher levels to small-scale edge servers at the lower levels, which are closer to end users for low-latency; (2) heterogeneous resource distribution, where cloud servers offer higher computing resources, while edge servers operate with limited computing power; (3) High Locality, as most transactions in edge databases (e.g., [110, 71]) occur within the same zone due to its physical proximity; (4) Mobility, with certain user devices (e.g., laptops and smartphones) moving across zones; and (5) Offloading, as resource-constrained edge servers tend to offload heavy workloads to higher-level cloud servers.

Given these properties of edge computing, our sharding-based hierarchical blockchain

is well-suited to exploit such an environment and achieve scalability. The system consists of a lower level of local chains and a higher level of a main chain, corresponding to edge servers and cloud servers, respectively. It supports the O-X-O-V processing model to efficiently manage locality workloads and introduces a state reshard protocol to handle user mobility. Additionally, it offloads the persistence of storage-heavy full shard copies to higher-level servers, ensuring availability. For these reasons, our system can be described as a blockchain platform optimized for edge computing. For future work, we plan to systematically evaluate our system in real-world edge computing environments.

Trade-off between Performance and Availability Applying sharding to a blockchain improves performance due to enhanced parallelism but proportionally limits availability, as each member holds only a single shard. In this thesis, we designed and evaluated DyloChain and PyloChain, which follow a balanced sharding scheme. Our experiments demonstrated that these systems balance performance and availability between availability-oriented sharding and performance-oriented sharding. However, there is still room for improvement in bridging the performance gap between our systems and performance-oriented sharding.

For example, even when all transactions are local, our systems exhibit significantly lower performance compared to performance-oriented sharding because transaction finality depends on the main chain. By effectively exploiting the fact that transactions are all local, the system could eliminate the need to rely on the main chain for finality. This would require techniques to detect global transactions and their state dependencies early before they are committed to the main chain. Currently, global transactions are included in local blocks and then propagated to the main chain, even though they cannot be validated on local chains. This occurs because local members (or L-nodes) maintain only a single copy of the shard and rigidly follow the O-X-O-V flow, which is not necessary to do so. This unnecessarily increases the latency of global transactions. As future work, we could explore creating global blocks at the

main chain specifically designed for global transactions. Note that this would replace PyloChain's global transaction scheduling technique.

Additionally, the primary overhead in balanced sharding arises from broadcasting local blocks across local chains to ensure availability. This inevitably consumes significant network resources, and its negative impact is amplified as the size of local blocks increases. To address this issue, we can consider various solutions for a lightweigh main chain: i)To reduce the cost of ensuring availability, compress local blocks by summarizing their content or transmitting only delta changes in terms of state updates. ii) Selectively propagate blocks based on their importance or priority. iii) Broadcast only the block headers while propagating block bodies through gossip networks or separate network layers.

Finally, in high-throughput systems, the size of the main chain can grow exponentially, making infinite storage infeasible. For real-world systems, blockchains need to periodically prune their data to manage storage efficiently. However, this comes at the cost of sacrificing historical records for storage optimization.

In conclusion, while our balanced sharding systems offer a promising trade-off between performance and availability, these enhancements would further optimize their practicality and scalability.

Byzantine Shard Resilient Cross-Shard Protocol It is well known that in sharding systems, security and performance often exhibit a trade-off relationship in the number of shards [111, 112, 113, 114]. For example, in a performance sharding, a single blockchain with a typical BFT protocol [25] that follows the $3f + 1$ model, where f is the maximum number of tolerable Byzantine members out of $3f + 1$ members in total, can tolerate up to 33 Byzantine members when managed collectively by 100 members. If this system is divided into 10 subnetworks with an equal number of members, each sharded network can tolerate up to only 3 Byzantine members. In such cases, attackers find it much easier to compromise the smaller shard groups, increasing the likelihood of a breach in the entire system's security.

DyloChain and PyloChain are designed for permissioned consortium blockchains and follow a balanced sharding approach, providing better security properties in terms of availability compared to performance sharding. Specifically, even if a specific shard group, i.e., a local chain, becomes completely unavailable, the higher-level main chain ensures availability by replicating the local chain. However, it is important to note that caution is required if the safety of a local chain is completely violated, such as when all members of the local chain behave maliciously. In our system, the higher-level main chain maintains its safety and liveness even in the presence of a Byzantine local chain, thanks to the main chain's f-tolerant consensus mechanism, although performance may be negatively affected. Nonetheless, our current system has a limitation in addressing application-level malicious attacks, such as updating a user's account with a fake value and using this fake value to corrupt other shard's state data via malicious cross-shard transactions. We leave the design of a Byzantine-resilient cross-local chain protocol to address this limitation as future work.

Note that in public blockchains [115, 30, 32, 47], the number of servers is typically very large. When applying sharding to the public blockchains, this relatively ensures that each shard has a sufficient number of servers to provide higher fault tolerance with a high probability with dynamically reconfiguring shard groups.

Parallel Global TX Executions In DyloChain and PyloChain, global transactions are processed on the main chain using the traditional order-execute transaction processing model. As the percentage of global transactions increases, the overhead of processing main blocks rises proportionally. This is especially costly because many global transactions are processed on a high-throughput, DAG-based main block as in PyloChain, as we demonstrated in our PyloChain evaluation. Fortunately, our main chain can easily accommodate the use of various parallel transaction execution algorithms for global transactions. For instance, we could construct a dependency graph for global transactions in each main block and execute non-conflicting transactions in parallel [116, 117, 118, 119, 120, 121], thereby significantly boosting performance for

global transactions. We consider integrating such algorithms into PyloChain to be a straightforward extension and leave this as future work.

Across main blocks scheduling In PyloChain, to further reduce interferences, scheduling across main blocks could be considered rather than within a single main block. However, since transaction commit events occur at the main block level, this approach could introduce fairness issues from the user's perspective (e.g., an earlier submitted transaction is committed later than a subsequently submitted transaction). Additionally, looking ahead across multiple main blocks and scheduling them could worsen commit latency. Thus, PyloChain performs scheduling only within a single main block, sidestepping fairness issues and maintaining low latency.

Member Selection In balanced sharding, where shards are replicated heterogeneously—unlike in availability and performance sharding—selecting full and local members from the entire consortium members can become a critical issue for real-world use cases. For instance, members with relatively ample budgets could join as full members by equipping high-performance machines, while others with fewer resources might participate as local members using less-capable machines. Alternatively, a blockchain-as-a-service provider, externally managed and running in a cloud environment, could be utilized to deliver full member services, collectively funded by the members. Another consideration is to avoid categorizing members strictly as local or full. Instead, shards could be configured in a more fine-grained manner, allowing for varying levels of availability and reduced storage costs according to consortium requirements. For instance, partial members might maintain different numbers of shards. However, this fine-grained replication would add considerable complexity to the system. Further research would be necessary to explore new challenges in such a setup.

VI. Conclusion

In this thesis, we present DyloChain and PyloChain, two sharding-based hierarchical blockchain models addressing scalability challenges in different network models.DyloChain with a synchronous main chain extends the X-O-V architecture to accommodate a sharding-based hierarchical blockchain with the O-X-O-V transaction processing model. DyloChain also effectively manages dynamic locality workloads with state reshard protocol. PyloChain with a partially synchronous main chain improves upon DyloChain by leveraging a DAG-based mempool with an efficient BFT consensus for scalable main chain consensus and reducing cross-shard TX interferences with a simple scheduling technique. Both systems were implemented and evaluated, demonstrating their feasibility and effectiveness, providing a strong foundation for future research in hierarchical blockchains.

요 약 문

블록체인은 상호 신뢰하지 않는 참여자들 간에 합의를 이룰 수 있는 분산 시스템으로, 탈중앙화, 가용성, 장애 내성, 무결성, 투명성을 제공한다. 이러한 특징 덕분에 블록체인 기반 서비스는 Web3.0, 탈중앙 금융(DeFi) 등 차세대 IT 서비스 패러다임에서 중요성이 커지고 있다. 하지만 이러한 이점에도 불구하고 블록체인은 제한된 성능 확장성 때문에 실제 응용 프로그램에서 널리 활용되지 못하고 있다. 따라서 블록체인 아키텍처의 다양한 레벨에서 성능을 향상시키기 위한 연구가 진행되고 있다. 예를 들어, 합의 알고리즘, 트랜잭션 처리 방식, 데이터베이스, 네트워크 프로토콜, 스토리지 시스템, 암호학, 하드웨어, 오프체인 기술, 응용 계층 최적화 등이 블록체인 성능 확장성을 개선하기 위해 제안되고 있다.

이러한 기술 중에서도 합의와 트랜잭션 처리는 블록체인의 코어 기술로, 성능 향상에 있어서 가장 큰 잠재력을 가지며 블록체인 자체를 직접적으로 개선한다. 특히, 샤딩은 흔히 블록체인 트릴레마라 불리는 문제를 고려하여 블록체인 성능을 향상시키는 가장 효과적인 방법 중 하나로 여겨진다. 샤딩은 전통적인 데이터베이스에서 네트워크를 작은 그룹(샤드)으로 나누어 수평적 확장성을 제공하는 데 널리 사용되는 방법이다. 각 샤드 그룹이 독립적인 데이터와 트랜잭션을 처리하면서 샤딩은 시스템 내 샤드 그룹의 수가 증가함에 따라 높은 수준의 병렬성을 달성한다.

블록체인에서 샤딩은 블록체인 네트워크를 여러 하위 네트워크로 나누어 각 네트워크가 자체 샤드 그룹 내에서만 합의를 수행하며, 독립적인 트랜잭션을 처리할 수 있도록 한다. 그 결과, 샤딩 기반 블록체인은 컴퓨팅, 스토리지, 네트워크 비용을

크게 줄이면서 수평적 확장성을 달성한다.

하지만, 샤딩을 기반으로 한 블록체인 플랫폼을 설계하는 것은 간단하지 않으며 여러 가지 챌린지를 제시한다. 첫째, 제한된 가용성을 극복하는 챌린지를 다뤄야 한다. 대부분의 기존 블록체인 샤딩 프로토콜은 성능 중심 샤딩(완전 또는 풀 샤딩이라고도 함)을 따르며, 각 멤버가 완전히 분리된 샤드를 관리하도록 설계되었다. 이는 각 샤드 그룹의 크기를 줄이지만, 이로 인해 가용성이 극적으로 감소하게 되어 재난 상황 (예. 지역 규모의 지진) 등으로 인한 데이터 손실 같은 치명적인 결과를 초래할 수 있다. 반대로, 가용성 중심 샤딩은 모든 멤버가 모든 샤드를 보유하도록 하여 각 멤버 내에서 병렬 블록체인 인스턴스들을 운용하면서 높은 가용성을 제공한다. 하지만 멤버와 샤드의 수가 증가함에 따라 각 멤버의 컴퓨팅 오버헤드도 급격히 증가한다.

둘째, 크로스 샤드 트랜잭션을 효율적으로 처리하는 챌린지다. 이는 여러 샤드를 조율하여 트랜잭션을 원자적으로 커밋하는 크로스 샤드 프로토콜을 필요로 한다. 원자적 커밋 프로토콜은 복잡하고 비효율적인 것으로 알려져 있다. 예를 들어, 일반적인 2단계 커밋(2PC) 기반 크로스 샤드 프로토콜은 조정자가 관련 샤드 전체에 복잡한 락킹 메커니즘을 구현해야 한다.

셋째, 동적 지역성을 지원하는 챌린지다. 대부분의 기존 블록체인 샤딩 프로토콜은 동적 지역성(시간에 따라 변하는 높은 지역성을 가진 워크로드 특성)을 충분히 고려하지 않는다. 예를 들어, 처음에 샤딩된 데이터가 잘 분할되었더라도, 지역성은 시간에 따라 변할 수 있으며(예: 사용자 이동성), 이를 적시에 처리하지 않으면 크로스 샤드 트랜잭션 수가 누적적으로 증가하여 시스템 성능이 저하될 수 있다.

본 논문에서는 샤딩을 블록체인에 도입할 때 발생하는 챌린지들을 다루기 위해

샤딩 기반 계층적 블록체인을 제안한다. 이 접근법은 상위 계층의 메인 체인과 하위 계층의 여러 로컬 체인을 포함하며, 메인 체인은 전체 샤드 복사본을 보유한 멤버들로, 로컬 체인은 단일 샤드 복사본을 보유한 멤버들로 구성된다. 이를 균형 샤딩(balanced sharding)으로 부르며, 다음과 같은 방식으로 문제를 해결한다.

첫째, 제한된 가용성 챌린지를 다루기 위해 샤딩 기반 계층적 블록체인은 하위 체인의 블록을 상위 계층 메인 체인에 복제함으로써 이를 달성한다. 이 접근법은 성능 중심 샤딩과 가용성 중심 샤딩의 이점을 결합한다. 구체적으로, 네트워크를 계층적으로 샤딩된 존으로 구성하며, 각 존 리더는 상위 계층 메인 체인을 관리하고, 각 존 내 다른 로컬 멤버는 하위 계층 로컬 체인을 관리한다. 이 구조는 샤드 그룹에 고장 발생 시 메인 체인의 복제된 로컬 체인을 통해 복구를 가능하게 하여 가용성을 보장하면서도 여러 로컬 체인을 통해 병렬성을 유지한다.

둘째, 크로스 샤드 트랜잭션을 효율적으로 처리하기 위해 메인 체인의 각 구성 멤버는 모든 보유한 샤드들을 (collocated shards) 활용하여 이러한 트랜잭션을 추가적인 조정 없이 간단하고 효율적으로 처리할 수 있다. 이를 위해 계층적으로 샤드화된 블록체인에 대한 O-X-O-V(Order-Execute-Order-Validate) 트랜잭션 처리 모델을 제안한다. 이 모델은 로컬 체인에서 로컬 트랜잭션을 병렬로 실행하고, 메인 체인에서 크로스 샤드 트랜잭션을 통합함으로써 단순하고 효율적인 통합 트랜잭션 처리 플랫폼을 제공한다.

셋째, 동적 지역성을 지원하기 위해 사용자의 상태를 안전하게 로컬 체인 간에 전송할 수 있는 상태 리샤드 프로토콜(state reshard protocol)을 설계한다. 이 프로토콜은 메인 체인을 신뢰할 수 있는 데이터 소스로 사용하여, 사용자가 제출하는 리샤드 트랜잭션을 통해 출발 로컬 체인에서 목적지 로컬 체인으로 사용자의 상태

를 전송한다. 이를 통해 크로스 샤드 트랜잭션 수를 효과적으로 줄이고, 샤딩 기반 계층적 블록체인의 전반적인 성능을 향상시킨다.

제안한 접근법을 실현하기 위해 DyloChain과 PyloChain이라는 두 가지 블록체인 시스템을 개발하였다. 이 두 시스템은 메인 체인의 네트워크 가정에 따라 서로 다른 방식으로 설계되었다. DyloChain은 동기 네트워크 환경에서 계층적으로 샤딩된 트랜잭션을 효과적으로 처리하기 위해 O-X-O-V 트랜잭션 처리 모델과 동적 지역성을 지원하는 메인 체인 기반의 상태 리샤드 프로토콜을 제안한다. 한편, PyloChain은 DyloChain을 기반으로 일부 비동기 구간이 존재하는 네트워크를 고려하여 메인 체인을 재설계하였다. PyloChain은 방향 비순환 그래프(Directed Acyclic Graph, DAG) 기반의 고성능 비잔틴 합의 알고리즘을 도입하고, 크로스 샤드 트랜잭션 처리 효율성을 높이기 위한 스케줄링 기술을 적용함으로써 메인 체인 및 크로스 샤드 트랜잭션의 확장성을 강화한다.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

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

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Sui delivers the benefits of web3 with the ease of web2, 2024.

[5] Aptos: The world's most production-ready blockchain, 2024.

[6] Solana: Web3 infrastructure for everyone, 2024.

[7] Ripple: Global payments & financial solutions for businesses, 2024.

[8] Kaia is an evm layer 1 public blockchain designed to bring web3 to millions of users across asia, 2024.

[9] Eos network - high-performance blockchain for scalable dapps, 2024.

[10] An open, feeless data and value transfer protocol, 2024.

[11] Building the most powerful tools for distributed networks, 2024.

[12] What is web3?, 2024.

[13] Decentralized finance, 2024.

[14] Delivering transparency and security in every vote, 2024.

[15] What are blockchain games?, 2024.

[16] Blockchain in supply chain management, 2024.

[17] Abid Haleem, Mohd Javaid, Ravi Pratap Singh, Rajiv Suman, and Shanay Rab. Blockchain technology applications in healthcare: An overview. *International Journal of Intelligent Networks*, 2:130–139, 2021.

[18] Changlin Yang, Ying Liu, Kwan-Wu Chin, Jiguang Wang, Huawei Huang, and Zibin Zheng. A Novel Two-Layer DAG-Based Reactive Protocol for IoT Data Reliability in Metaverse . In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 25–36, Los Alamitos, CA, USA, July 2023. IEEE Computer Society.

[19] Sangwon Hong, Yoongdoo Noh, Jeyoung Hwang, and Chanik Park. Fabasset: Unique digital asset management system for hyperledger fabric. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1269–1274, 2020.

[20] Haya R. Hasan, Khaled Salah, Raja Jayaraman, Junaid Arshad, Ibrar Yaqoob, Mohammed Omar, and Samer Ellahham. Blockchain-based solution for covid-19 digital medical passports and immunity certificates. *IEEE Access*, 8:222093–222108, 2020.

[21] Yihao Guo, Zhiguo Wan, Hui Cui, Xiuzhen Cheng, and Falko Dressler. Vehicloak: A blockchain-enabled privacy-preserving payment scheme for location-based vehicular services. *IEEE Transactions on Mobile Computing*, 22(11):6830–6842, 2023.

[22] Yongrae Jo and Chanik Park. Blocklot: Blockchain based verifiable lottery. *arXiv preprint arXiv:1912.00642*, 2019.

[23] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *J. ACM*, 71(4), August 2024.

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

[25] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[26] Mohammad Javad Amiri, Ziliang Lai, Liana Patel, Boon Thau Loo, Eric Lo, and Wenchao Zhou. Saguaro: An edge computing-enabled hierarchical permissioned blockchain. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 259–272, 2023.

[27] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 123–140. ACM, 2019.

[28] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 76–88, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Feng Cheng, Jiang Xiao, Cunyang Liu, Shijie Zhang, Yifan Zhou, Bo Li, Baochun Li, and Hai Jin. Shardag: Scaling dag-based blockchains via adap-

tive sharding. *2024 40th IEEE International Conference on Data Engineering (ICDE)*, 2024.

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

[31] Mohammad Javad Amiri, Daniel Shu, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. Ziziphus: Scalable data management across byzantine edge servers. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 490–502, 2023.

[32] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 931–948, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Huawei Huang, Xiaowen Peng, Jianzhou Zhan, Shenyang Zhang, Yue Lin, Zibin Zheng, and Song Guo. Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1968–1977, 2022.

[34] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[35] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. *Proc. VLDB Endow.*, 14(11):2314–2326, jul 2021.

[36] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. Gridb: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *Proc. VLDB Endow.*, 16(7):1685–1698, mar 2023.

[37] Mingzhe Li, You Lin, Jin Zhang, and Wei Wang. Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 133–143, 2022.

[38] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. Qanaat: a scalable multi-enterprise permissioned blockchain system with confidentiality guarantees. *Proc. VLDB Endow.*, 15(11):2839–2852, jul 2022.

[39] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[40] GT. Cybersecurity update causes worldwide microsoft outages, Jan 2024.

[41] SeoulShinmoon. Pangyo data center fire, Jan 2022.

[42] Paul Lipscombe. More than 170 base stations offline in taiwan following earthquake, Jan 2024.

[43] Peilin Zheng, Quanqing Xu, Zibin Zheng, Zhiyuan Zhou, Ying Yan, and Hui Zhang. Meepo: Sharded consortium blockchain. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1847–1852, 2021.

[44] Jianbo Gao, Jiashuo Zhang, Yue Li, Jiakun Hao, Ke Wang, Zhi Guan, and Zhong Chen. Pshard: A practical sharding protocol for enterprise blockchain. In

Proceedings of the 2022 5th International Conference on Blockchain Technology and Applications, ICBTA '22, page 110–116, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Peilin Zheng, Quanqing Xu, Zibin Zheng, Zhiyuan Zhou, Ying Yan, and Hui Zhang. Meepo: Multiple execution environments per organization in sharded consortium blockchain. *IEEE Journal on Selected Areas in Communications*, 40(12):3562–3574, 2022.

[46] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105, 2020.

[47] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.

[48] Y. Zhang, S. Pan, and J. Yu. Txallo: Dynamic transaction allocation in sharded blockchain systems. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 721–733, Los Alamitos, CA, USA, apr 2023. IEEE Computer Society.

[49] L. N. Nguyen, T. T. Nguyen, T. N. Dinh, and M. T. Thai. Optchain: Optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535, Los Alamitos, CA, USA, jul 2019. IEEE Computer Society.

[50] Michał Król, Onur Ascigil, Sergi Rene, Alberto Sonnino, Mustafa Al-Bassam, and Etienne Rivière. *Shard Scheduler: Object Placement and Migration in Sharded Account-Based Blockchains*, page 43–56. Association for Computing Machinery, New York, NY, USA, 2021.

[51] Liuyang Ren, Paul A. S. Ward, and Bernard Wong. Toward reducing cross-shard transaction overhead in sharded blockchains. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 43–54, New York, NY, USA, 2022. Association for Computing Machinery.

[52] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery.

[54] Yongrae Jo and Chanik Park. A hierarchical blockchain supporting dynamic locality by extending execute-order-validate architecture. *Distrib. Ledger Technol.*, aug 2024. Just Accepted.

[55] Hyperledger fabric v2.1.0 branch, 2020.

[56] Jeonghyeon Ma, Yongrae Jo, and Chanik Park. Peerbft: Making hyperledger fabric's ordering service withstand byzantine faults. *IEEE Access*, 8:217255–217267, 2020.

[57] narwhal, 2021.

[58] Bronw Univ. Smallbank benchmark, December 2019.

[59] www.hyperledger.org. Fujitsu and botanical water technologies create the world's first global water trading platform using hyperledger fabric., December 2023.

[60] www.hyperledger.org. Gsbn: A new global trade operating system, December 2022.

[61] www.hyperledger.org. How tech mahindra deployed hyperledger fabric for the digital transformation of abu dhabi's land registry., December 2023.

[62] openidl. openid: The first blockchain network connecting data across the insurance industry., December 2023.

[63] www.hyperledger.org. Hyperledger-powered healthcare solutions in action., August 2021.

[64] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[65] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: The case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 105–122, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery.

[67] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. Why do my blockchain transactions fail? a study of hyperledger fabric. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 221–234, New York, NY, USA, 2021. Association for Computing Machinery.

[68] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 110–122, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223, jan 2020.

[70] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1221–1236, New York, NY, USA, 2018. Association for Computing Machinery.

[71] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 145–161, New York, NY, USA, 2021. Association for Computing Machinery.

[72] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, jul 2019.

[73] Joseph Noor, Mani Srivastava, and Ravi Netravali. Portkey: Adaptive key-value placement over dynamic edge networks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 197–213, 2021.

[74] Latif U. Khan, Ibrar Yaqoob, Nguyen H. Tran, S. M. Ahsan Kazmi, Tri Nguyen Dang, and Choong Seon Hong. Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet of Things Journal*, 7(10):10200–10232, 2020.

[75] Tie Qiu, Jiancheng Chi, Xiaobo Zhou, Zhaolong Ning, Mohammed Atiquzzaman, and Dapeng Oliver Wu. Edge computing in industrial internet of things: Architecture, advances and challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2462–2488, 2020.

[76] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[77] www.accenture.com. Edge computing for store of tomorrow, April 2022.

[78] Haibo Tang, Huan Zhang, Zhenyu Zhang, Zhao Zhang, Cheqing Jin, and Aoying Zhou. Towards high-performance transactions via hierarchical blockchain sharding. In *Euro-Par 2024: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26–30, 2024, Proceedings, Part I*, page 373–388, Berlin, Heidelberg, 2024. Springer-Verlag.

[79] Zicong Hong, Song Guo, and Peng Li. Scaling blockchain via layered sharding. *IEEE Journal on Selected Areas in Communications*, 40(12):3575–3588, 2022.

[80] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Dag-based blockchain systems. *ACM Comput. Surv.*, 55(12), mar 2023.

[81] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[82] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery.

[83] Ethereum. Side chain, March 2024.

[84] Ethereum. Optimistic rollups, November 2023.

[85] Ethereum. Danksharding, March 2024.

[86] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[87] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cedric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich. IA-CCF: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 467–491, Renton, WA, April 2022. USENIX Association.

[88] Mohammad Reza Nosouhi, Shui Yu, Marthie Grobler, Qingyi Zhu, and Yong Xiang. Blockchain–based location proof generation and verification. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2019.

[89] Google. The go programming language, January 2021.

[90] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[91] implementation of smallbank in golang, 2021.

[92] cloudping.co. Cloudping, January 2024.

[93] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version, 2022.

[94] Sisi Duan and Haibin Zhang. Foundations of dynamic bft. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1317–1334, 2022.

[95] Rust Team. Rust: A language empowering everyone to build reliable and efficient software., January 2024.

[96] Docker, 2024.

[97] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[98] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[99] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 50–61, New York, NY, USA, 2011. Association for Computing Machinery.

[100] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[101] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[102] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118, 2020.

[103] Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1686–1699, New York, NY, USA, 2021. Association for Computing Machinery.

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

[105] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[106] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 189–204. ACM, 2007.

[107] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308. ACM, 2012.

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

[109] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237. ACM, 2017.

[110] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 210–227, New York, NY, USA, 2021. Association for Computing Machinery.

[111] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 683–696, New York, NY, USA, 2022. Association for Computing Machinery.

[112] Yibin Xu, Jingyi Zheng, Boris Düdder, Tijs Slaats, and Yongluan Zhou. A two-layer blockchain sharding protocol leveraging safety and liveness for enhanced performance. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[113] Mitch Jacovetty, Joseph Oglio, Mikhail Nesterenko, and Gokarna Sharma. Trail: Cross-shard validation for cryptocurrency byzantine shard protection, 2024.

[114] Meiqi Li, Xinyi Luo, Kaiping Xue, Yingjie Xue, Wentuo Sun, and Jian Li. A secure and efficient blockchain sharding scheme via hybrid consensus and dynamic management. *IEEE Transactions on Information Forensics and Security*, 19:5911–5924, 2024.

[115] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[116] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery.

[117] Zhihao Chen, Xiaodong Qi, Xiaofan Du, Zhao Zhang, and Cheqing Jin. Peep: A parallel execution engine for permissioned blockchain systems. In *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part III*, page 341–357, Berlin, Heidelberg, 2021. Springer-Verlag.

[118] Yaozheng Fang, Zhiyuan Zhou, Surong Dai, Jinni Yang, Hui Zhang, and Ye Lu. Pavm: A parallel virtual machine for smart contract execution and validation. *IEEE Transactions on Parallel and Distributed Systems*, 35(1):186–202, 2024.

[119] X. Qi, J. Jiao, and Y. Li. Smart contract parallel execution with fine-grained state accesses. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 841–852, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society.

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

[121] Donghyeon Ryu and Chanik Park. Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution . In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*, pages 391–406, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

Acknowledgements

2017년 1월 초, 쌀쌀한 날씨 속에 포항에 도착하여 포항공과대학교 컴퓨터공학과 박사과정 학생으로서 시스템 소프트웨어 연구실에 첫 출근했던 순간이 아직도 생생히 기억납니다. 그로부터 어느덧 8년이라는 시간이 흘러 이제 박사 학위를 받게 되었습니다. 이 과정은 단순히 학문적 성취를 넘어, 제 인생의 중요한 이정표가 되었고, 앞으로 걸어갈 여정을 위한 소중한 밑거름이 되었습니다. 이 자리를 빌려 저를 아낌없이 지원해주신 많은 분들께 깊은 감사의 말씀을 드립니다.

먼저, 박사 과정 동안 저를 물심양면으로 지도해주신 박찬익 교수님께 진심으로 감사드립니다. 교수님의 통찰력 있는 피드백과 세심한 관찰, 풍부한 경험, 그리고 강인한 인내심을 통해 저는 한 걸음 더 성장할 수 있었습니다. 특히, 교수님께서는 연구뿐만 아니라 명확하고 설득력 있는 의사소통 기술을 익히고 다듬는 데에도 큰 도움을 주셨으며, 이를 통해 연구자로서의 자신감을 키울 수 있었습니다. 무엇보다도, 교수님께서 제공해주신 안정적인 연구 환경과 끊임없는 격려는 제가 끝까지 연구에 몰두할 수 있도록 만들어주셨습니다. 교수님의 열정과 지도는 제가 이 과정을 마칠 수 있었던 가장 큰 원동력이었습니다.

아울러, 귀중한 시간을 내어 제 논문 심사에 참여해주신 송황준 교수님, 김광선 교수님, 박지성 교수님, 송민석 교수님께도 깊은 감사를 드립니다. 심사 위원 교수님들께서 주신 유용한 조언은 제 연구의 완성도를 높이고, 미래 연구 방향에 대해 새로운 통찰을 주셨습니다.

긴 시간 동안 시스템 소프트웨어 연구실에서 함께 동고동락하며 연구와 일상에서 큰 힘이 되어준 동료들에게 감사의 말씀을 전합니다. 연구실은 작은 사회와 같아, 다양한 사람들이 모여 서로의 연구에 관심을 가지며 지원하고, 사회적으로도 많은 성장을 이룰 수 있는 특별한 공간이었습니다. 사려 깊은 마음과 든든함으로 힘이 되어준 우창이, 특유의 에너지로 분위기 메이커가 되어준 문현이, 명랑하고 영리한 모습으로 연구실에 활기를 더해준 동현이, 행정 업무를 세심하게 지원해주신 소다미 선생님 덕분에 학위 과정의 어려운 막바지도 무탈하게 보낼 수 있었습니다.

또한, 연구실 졸업생 분들께도 감사의 말씀을 전합니다. 연구실에 처음 들어왔을 때부터 변함없이 따뜻하게 대해주신 정현이 형, 재밌고 진심 어린 대화를 통해 뜻 깊은 생각을 나눌 수 있었던 운성이 형에게도 감사의 뜻을 전합니다. 언제나 편안하고 차분한 분위기를 만들어주었던 융두, 호기심 많고 폭넓은 지식을 공유해준 상원, 재치 있는 입담으로 분위기를 밝게 만들어준 제영, 꾸준하고 성실한 모습으로 귀감이 되어준 하늘, 스마트하고 논리적인 연규, 센스 있는 감각의 지은이, 서글서글한 베트남 친구 Hieu, 연구실 동기로 먼저 졸업한 미래, 푸근하고 열정적인 동민이, 유쾌하고 재치있는 병훈이, 활기차고 긍정적인 해성이 덕분에 연구실 생활이 더욱 풍성하고 기억에 남는 시간이 되었습니다. 비록 연구실 생활을 함께하지는 않았지만, 여러 과제에서 많은 도움과 조언을 주신 박세진 박사님께도 깊은 감사를 드립니다.

마지막으로, 항상 저를 믿고 아낌없이 응원해준 가족에게 진심으로 감사드립니다. 학위 과정 동안 많은 어려움 속에서도 저를 향한 신뢰와 지지를 보내주셨기에 제가 이 자리에 설 수 있었습니다. 정말 감사합니다.

이 박사 학위 논문은 끝이 아니라, 앞으로 더 큰 성장을 위한 발판이 될 것입니다. 학위 과정에서 경험한 소중한 순간들과 많은 분들께 받은 가르침은 제가 새로운 도전에 나서고 지속적으로 성장하는 데 있어 든든한 힘과 영감을 줄 것입니다. 모든 분들께 다시 한 번 깊이 감사드립니다.

Curriculum Vitae

Name : Yongrae Jo

Education

2017. 02. – 2025. 02. Department of Computer Science and Technology, Pohang University of Science and Technology (Ph.D.)

2012. 03. – 2017. 02. Department of Computer Science and Technology, Pusan National University (B.S.)

Experience

Sep. 2017 – Dec. 2017 Teaching Assistant, Microprocessor Architecture and Programming, CSED211

Sep. 2018 – Dec. 2018 Teaching Assistant, Operating Systems, CSED312

Apr. 2017 – Dec. 2018 Development of High-performance and High-reliable Blockchain for Distributed Autonomous IoT Platform

Apr. 2020 – Dec. 2021 Core Technologies for 5G-Aware Blockchain Networks

Apr. 2021 – Dec. 2024 Core Technologies for Hybrid P2P Network-based Blockchain Services

Apr. 2021 – Dec. 2024 Development of Big Blockchain Data Highly Scalable Distributed Storage Technology for Increased Applications in Various Industries

Publications

1. Yongare Jo and Chanik Park, 2024. ”A Hierarchical Blockchain supporting Dynamic Locality in Edge Computing by Extending Execute-Order-Validate Architecture”, In Blockchain-Based Pervasive Systems: Theory, Applications, and Challenges [Special issue]. ACM Distributed Ledger Technologies: Research and Practice (DLT’24), 2024
2. Yongare Jo and Chanik Park, ”Enhancing Ethereum PoA Clique Network with DAG-based BFT Consensus”, 6th IEEE International Conference on Blockchain and Cryptocurrency (ICBC’24), May 27 – May 31, 2024, Dublin, Ireland
3. Yongrae Jo, Jeonghyun Ma and Chanik Park, Toward Trustworthy Blockchain-as-a-Service with Auditing, 40th IEEE International Conference on Distributed Computing Systems (ICDCS’20), November 29 – December 1, 2020, Singapore (Acceptance Rate: 18%, 105 out of 584)
4. Jeonghyeon Ma, Yongrae Jo and Chanik Park, PeerBFT: Making Hyperledger Fabric’s Ordering Service Withstand Byzantine Faults, IEEE Access, 2020, doi: 10.1109/ACCESS.2020.3040443
5. Yongrae Jo and Chanik Park, Codit: Collaborative Auditing for BaaS, 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers December 9th-13th, 2019 Collocated with Middleware 2019, UC Davis, CA, USA
6. Yongrae Jo and Chanik Park, A Blockchain Sharding Protocol supporting Dynamic Locality in Mobile Edge Computing, the 13th International Conference on ICT Convergence (ICTC 2022).

7. Yongrae Jo and Chanik Park, BlockLot: Blockchain based Verifiable Lottery, arXiv preprint arXiv:1912.00642
8. Yongrae Jo and Chanik Park, A Novel Cross-Shard Protocol for Hierarchical State Sharding Blockchain, The 29th ACM Symposium on Operating System Principles, Poster Presentation, October 23 - October 26, 2023, Koblenz
9. Yongrae Jo and Chanik Park, Delegated Byzantine Fault Tolerance Using Trusted Execution Environment, Poster Presentation, 27th USENIX Security Symposium (USENIX Security '18)
10. Jeonghyeon Ma, Yongrae Jo, and Chanik Park, Redesigning Hyperledger Fabric Blockchain with Append-only Ledger, Poster Presentation, 13th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI'18)
11. Yongrae Jo and Chanik Park, MEC-Chain: Scalable Block Chain for Dynamic Locality, Korea Computer Congress (KCC) 2022
12. Yongrae Jo and Chanik Park, A Novel Transaction Processing in Hierarchical Blockchain for Edge Computing (KCC) 2023
13. Haesung Park, Yongrae Jo, and Chanik Park, B-Lottery: Blockchain based lottery system with flexible random seed, Korea Computer Congress (KCC) 2019