

UC Davis

UC Davis Electronic Theses and Dissertations

Title

Enhancing Blockchain Systems: A Comprehensive Study of Consensus, Scalability, and Performance Optimization

Permalink

<https://escholarship.org/uc/item/3q23x6pm>

ISBN

9798189226816

Author

chen, junchao

Publication Date

2026-06-25

Peer reviewed|Thesis/dissertation

Enhancing Blockchain Systems: A Comprehensive Study of Consensus, Scalability, and Performance

Optimization

By

Junchao Chen
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Mohammad Sadoghi Hamedani, Chair

Dongyu Liu

Cindy Rubio Gonzalez

Committee in Charge

2026

Contents

Abstract	v
Chapter 1. Introduction	1
1.1. A Unified, Deployable Permissioned Blockchain Fabric	1
1.2. Security: Blockchains and the Long-Range Attack Problem	3
1.3. Scalability: Sharding and Smart Contract Execution Without Prior Knowledge	4
1.4. Availability: Leader-Based Protocols and Network Partitions	4
1.5. Building a High-Performance, Easy-to-Use Research Platform	5
Chapter 2. Securing Consensus: Power-of-Collaboration against Long-Range Attacks	7
2.1. Introduction: The Vulnerability of Proof-of-Stake and BFT Systems	7
2.2. Preliminaries	10
2.3. The Threat Model: Understanding Long-Range Attacks	10
2.4. Our Proposal: The Power-of-Collaboration (PoC) Protocol	11
2.5. Ensuring Robustness: Handling Malicious Behavior in PoC	18
2.6. Discussion	23
2.7. Theoretical Guarantees: Security, Liveness Arguments	24
2.8. Theoretical Guarantees: Fairness Analysis	27
2.9. Experimental Evaluation: PoC's Performance and Impact	27
2.10. Related work	32
2.11. Summary and Discussion	33
Chapter 3. Scaling Execution: Thunderbolt for Concurrent Smart Contracts in Sharded DAGs	35
3.1. Introduction: The Bottleneck of Smart Contract Execution	35
3.2. Challenges: Parallelism Without Prior Knowledge and Efficient Cross-Shard Processing	38
3.3. System Overview: Thunderbolt's Unified Sharding and DAG Architecture	40
3.4. Core Mechanism 1: The Concurrent Executor (CE) for Single-Shard Transactions	42

3.5.	Core Mechanism 2: Coordinating Single-Shard and Cross-Shard Transactions	43
3.6.	Enhancing Robustness: Non-Blocking Shard Reconfiguration	47
3.7.	Correctness Analysis: Safety and Liveness in Thunderbolt	49
3.8.	Dependency Graph in Concurrency Controller	51
3.9.	Correctness Analysis: Safety and Liveness in Thunderbolt	54
3.10.	Serializability Proof of the Concurrency Controller	55
3.11.	Experimental Evaluation: Concurrency Executor Evaluation	56
3.12.	Experimental Evaluation: System Evaluation	60
3.13.	Related work	63
3.14.	Summary and Discussion	64
Chapter 4.	Ensuring Availability: CASSANDRA for Partial Progress Under Network Asynchrony	66
4.1.	Introduction: The Achilles' Heel of Leader-Based Protocols—Network Partitions	66
4.2.	System Design: CASSANDRA's Leaderless, DAG-Based Protocol	68
4.3.	Achieving Partial Progress: The Power of Proposer Exchange and Rapid Convergence	77
4.4.	Correctness Analysis: Safety and Liveness Guarantees Under Asynchrony	78
4.5.	Process CASSANDRA in Chain	80
4.6.	Experimental Evaluation: Performance During and After Network Partitions	81
4.7.	Related work	87
4.8.	Summary and Discussion	88
Chapter 5.	Chapter 5: ResilientDB – A High-Throughput Fabric for Validating and Comparing BFT Protocols	90
5.1.	Introduction: From Protocol Design to a Practical System	90
5.2.	System Architecture: A Modular Platform for Performance and Flexibility	91
5.3.	Ecosystem and Developer Tooling	93
5.4.	Key Design Decisions: System-Centric Optimizations	96
5.5.	Performance Benchmarking: Evaluating the Fabric's Capabilities	97
5.6.	Summary: ResilientDB as a Research and Development Platform	98
Chapter 6.	Future Work – Leveraging RDMA for Scalable Transaction Execution	100
6.1.	Residual Bottlenecks in the Current Fabric	100

6.2. Overview of Remote Direct Memory Access (RDMA)	101
6.3. Benefits of RDMA for Blockchain Consensus and Execution	102
6.4. Enabling Scalable Execution via Remote Memory Access	103
6.5. Concluding Remarks on Future Directions	104
Chapter 7. Conclusions	105
References	108

Abstract

Existing permissioned blockchain systems fail to simultaneously satisfy the essential properties required for real-world deployment: security against long-range attacks, scalable smart contract execution without prior knowledge of read/write sets, availability under network partitions, high system-level performance, and ease of use as a research testbed.

This dissertation addresses these gaps by designing, implementing, and validating a unified, deployable permissioned blockchain fabric composed of four interdependent innovations.

First, Power-of-Collaboration (PoC) secures any existing PoS/BFT system against long-range attacks. PoC makes ledger rewriting computationally expensive (like PoW) but replaces competitive mining with collaborative mining, eliminating resource waste.

Second, Thunderbolt scales smart contract execution without requiring prior read/write knowledge. It integrates Execute-Order-Validate for single-shard transactions with Order-Execute for cross-shard transactions via DAG-based consensus, achieving up to $50\times$ speedup over sequential DAG execution.

Third, CASSANDRA ensures partial progress under network partitions. This leaderless protocol enables replicas to continue proposing transactions via a DAG and recover rapidly, achieving throughput peaks up to $6\times$ of steady state.

Fourth, ResilientDB integrates the above contributions into a high-throughput permissioned fabric with a protocol-agnostic interface, benchmarking tools, and SDKs. It is open-sourced as an Apache Incubating project, serving as both a production system and an open research testbed.

By systematically addressing security, scalability, availability, and system performance within one unified fabric, this dissertation demonstrates a practical path toward building a complete, deployable permissioned blockchain for enterprise applications.

CHAPTER 1

Introduction

Blockchain technology has experienced significant growth, primarily driven by Bitcoin’s success as a cryptocurrency [1]. A blockchain, often referred to as a distributed ledger, is essentially an append-only data structure maintained by a network of mutually untrusting nodes. Within a blockchain network, all nodes reach consensus on an ordered set of blocks, each containing multiple transactions, forming a comprehensive sequential transaction log.

Traditional database systems deliver robust performance, ensuring high availability and data consistency through primary-backup models and multi-primary configurations. For consistency, these systems employ consensus protocols to replicate data across nodes, mitigating data loss risks [2, 3, 4].

In contrast, Byzantine environments present unique challenges where nodes may be compromised by malicious actors, risking data tampering and unreliable communications. Blockchain systems operating in such environments must implement robust security measures for secure communication and data protection. Many leverage advanced cryptographic techniques, such as public-private key encryption, to safeguard data effectively.

Blockchain systems in controlled environments are known as private (or permissioned) systems, distinguished from public (or permissionless) systems where participation is open to anyone [5, 6, 7]. Current applications span security trading and settlement, asset and financial management, and banking and insurance [8, 9]. While these applications are typically supported by enterprise-grade database systems like Oracle and MySQL, blockchain technology offers transformative potential despite certain performance and cost challenges.

Recent interest in blockchain technology has spurred advancements in distributed Byzantine fault-tolerant (BFT) systems capable of addressing failures and defending against malicious attacks.

1.1. A Unified, Deployable Permissioned Blockchain Fabric

Any blockchain system intended for real-world, enterprise-grade deployment must simultaneously satisfy several fundamental properties. First, it must be **secure** against a wide range of attacks, including those

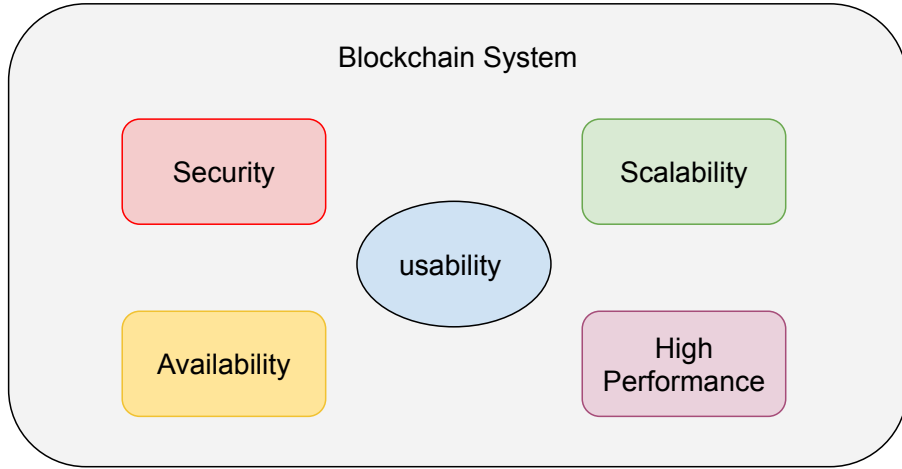


FIGURE 1.1. Properties of blockchain systems.

that exploit the absence of computational cost in ledger rewrites (e.g., long-range attacks). Second, it must be **scalable** in both transaction ordering and smart contract execution, supporting high throughput as the number of participants and state size grows. Third, it must be **highly available**, continuing to make progress even when network partitions or leader failures occur—a requirement often stated as liveness. Fourth, it must be **high-performance** at the system level, not merely at the protocol level, through careful architectural and engineering optimizations. Fifth, to accelerate research and adoption, it should be **easy to use** as a testbed, allowing researchers to implement and compare new protocols fairly and reproducibly.

Existing blockchain platforms, however, fall short of satisfying all these properties together. Permissioned systems often inherit consensus protocols (e.g., PBFT [10], HotStuff [11]) that remain vulnerable to **long-range attacks** because block addition is computationally inexpensive and adversaries can acquire old private keys. Scalability solutions based on sharding typically assume **prior knowledge of transaction read/write sets**—an unrealistic requirement for dynamic smart contracts—and rely on centralized coordinators or two-phase commit for cross-shard transactions, incurring high latency and abort rates. Leader-based consensus protocols, while efficient in good network conditions, **stall entirely during network partitions**, as a new leader must be elected via timeouts, causing prolonged unavailability. Moreover,

most existing systems are not architected as **high-performance, pluggable research platforms**; implementing a new consensus or execution protocol often requires deep modifications to the core codebase, making fair comparisons difficult.

To support this argument, the dissertation addresses five essential properties that any real-world permissioned blockchain must possess (Figure 1.1):

- **Security** against long-range attacks, which exploit the low cost of rewriting ledger history.
- **Scalability** in smart contract execution without requiring prior knowledge of transaction read/write sets.
- **Availability** under network partitions, continuing to make partial progress even when a full quorum cannot be formed.
- **High performance** at the system level, not just protocol level, through careful architectural optimizations.
- **Usability as a research testbed**, allowing fair, reproducible comparisons of different consensus and execution protocols.

The remainder of this introduction examines each of the first four challenge areas in turn.

1.2. Security: Blockchains and the Long-Range Attack Problem

Blockchains are classified as permissionless and permissioned. Permissionless blockchains (e.g., Bitcoin [1], Ethereum [12]) allow any node to join and leave freely, achieving high decentralization at the cost of computational overhead. Permissioned blockchains (e.g., Hyperledger Fabric) operate with a known set of replicas and use BFT protocols for efficiency. However, both types remain vulnerable to long-range attacks [13, 14]: an adversary who acquires old private keys (via theft or bribery) can create an alternative ledger from an early block, and new or rejoining nodes cannot distinguish the genuine chain from the fake one.

Existing countermeasures include key-evolving cryptography (which only delays attacks), checkpointing via a trusted committee (which introduces centralization), or anchoring to Bitcoin’s PoW ledger (which wastes computational resources due to competitive mining). No solution simultaneously provides decentralized, computationally expensive ledger rewriting without massive resource waste.

We propose PoC, a protocol that appends to any existing PoS/BFT system to guard against long-range attacks. PoC makes ledger rewriting computationally expensive (like PoW) but replaces competition with collaborative mining: all miners work on the same puzzle via partitioned search spaces. This eliminates

resource waste, ensures fairness through stake-based slice assignment, and provides accountability through a slice-shifting protocol that penalizes malicious miners. PoC thus secures the permissioned system without compromising throughput or decentralization.

1.3. Scalability: Sharding and Smart Contract Execution Without Prior Knowledge

Background. To scale transaction processing, many blockchain systems employ sharding: partitioning data into non-overlapping shards, each running its own consensus [15, 16, 17]. Cross-shard transactions require atomic commitment protocols (e.g., two-phase commit) to ensure consistency across shards [18, 19]. Most existing sharding systems assume that the read/write sets of transactions are known before execution – an assumption that holds for simple UTXO-based models but fails for dynamic smart contracts, where the exact keys accessed are determined only at runtime.

Limitations of existing solutions. Order-Execute (OE) models can build dependency graphs but need prior read/write knowledge. Execute-Order-Validate (EOV) models (e.g., Fabric) execute speculatively but suffer from high abort rates under contention. Relay-based or multi-shard consensus approaches introduce additional coordinators and communication overhead [20, 21, 22, 23]. No existing sharding system efficiently handles both single-shard and cross-shard smart contract transactions without prior knowledge of access patterns.

We introduce Thunderbolt, a sharded architecture that integrates the EOV model for single-shard transactions (no prior read/write knowledge needed) with the OE model for cross-shard transactions. A DAG-based consensus coordinates all shards without centralized coordinators. Thunderbolt also includes a Concurrent Executor (CE) that uses non-deterministic ordering based on runtime states to minimize aborts. This yields up to 50× speedup over sequential execution on DAG-based protocols, directly addressing the scalability gap.

1.4. Availability: Leader-Based Protocols and Network Partitions

Most practical BFT protocols (PBFT, HotStuff, Raft) are leader-based: a single replica proposes blocks, and others vote to commit. Under normal synchronous network conditions, these protocols achieve high throughput and low latency. However, they rely on the leader being correct and reachable. When network partitions occur or the leader fails, the system must trigger a leader election via timeouts [10, 11, 24, 25]. During this period, the entire system stalls – no new transactions can be committed.

Timeout-based leader election can take seconds or more, causing prolonged unavailability. Even leaderless DAG-based protocols like Tusk [26] still require a quorum to make progress; if no quorum exists due to partitions, they also halt. No existing protocol provides partial progress – the ability to continue ordering some transactions among subsets of replicas during a partition – without sacrificing safety.

We propose CASSANDRA, a leaderless protocol designed for partial progress under asynchrony. Replicas propose transactions in rounds, building a DAG that links proposals even when a full quorum is unavailable. Each replica selects a local strongest proposal; consensus on the global strongest proposal proceeds only when sufficient votes arrive, but replicas never wait indefinitely – timers allow them to move to the next round. During a partition, replicas continue exchanging proposals within their connected subset, achieving non-zero throughput. Upon network recovery, CASSANDRA rapidly converges by exchanging DAG histories, resulting in throughput peaks up to $6\times$ of steady state. This directly addresses the availability gap.

1.5. Building a High-Performance, Easy-to-Use Research Platform

Even the most innovative consensus or execution protocol is of limited practical value if the underlying system is slow, difficult to deploy, or hard to extend. Existing permissioned blockchains (Fabric, Tendermint, etc.) often suffer from suboptimal system-level designs: tight coupling of ordering and execution, inefficient network handling, and lack of modular interfaces for protocol researchers. As a result, comparing different protocols fairly requires re-implementing them from scratch, and system bottlenecks often obscure algorithmic improvements.

No existing platform provides both a high-performance engine and a pluggable, protocol-agnostic interface that allows researchers to implement new BFT protocols without modifying network, storage, or execution layers. Benchmarking tools are often ad-hoc, and deployment is complex.

We present ResilientDB, an open-source (Apache Incubating) permissioned blockchain fabric that embodies a system-centric design philosophy. Key features include:

- Decoupling of transaction ordering from execution, enabling deep pipelining.
- Multi-threaded coroutine architecture with batched, copy-on-write message processing.
- A protocol-agnostic consensus interface (Propose, Vote, Commit) that allows any BFT protocol to be plugged in without touching other components.
- Comprehensive benchmarking suite (YCSB, SmallBank), SDKs (Python, Rust), monitoring dashboards (Grafana), and deployment automation (Ansible, Docker).

We present ResilientDB, an open-source (Apache Incubating) permissioned blockchain fabric that embodies a system-centric design philosophy. ResilientDB serves as the integration backbone for PoC, Thunderbolt, and CASSANDRA. Moreover, even a slower consensus protocol (PBFT) running on ResilientDB can outperform other permissioned blockchains that use faster protocols (Zyzyva) – demonstrating that system-level optimizations are as critical as protocol innovations. Thus, ResilientDB fulfills the high-performance and research testbed requirements, completing the unified fabric.

Securing Consensus: Power-of-Collaboration against Long-Range Attacks

2.1. Introduction: The Vulnerability of Proof-of-Stake and BFT Systems

Decentralized systems built on blockchain technology promise immutable and verifiable ledgers [1, 27, 28]. These systems process client transactions through state-machine replication, requiring consensus protocols capable of handling malicious attacks [11, 29]. The two most widely adopted consensus categories are Proof-of-Stake (PoS) protocols [28, 30] and traditional Byzantine Fault-Tolerant (BFT) protocols [11, 29].

Stake-oriented protocols probabilistically determine which node adds new blocks, with higher-stake nodes having greater proposal probability [30]. Communication-oriented protocols provide equal voting opportunities, reaching agreement through successive vote exchanges [29, 31]. Some systems combine PoS and BFT for efficient consensus [28]. Despite differences, these protocols share a common design: each block includes digital signatures from a quorum of participants, proving quorum agreement on ledger updates.

The Long-Range Attack Problem Decentralized systems employing these protocols face a well-known vulnerability: long-range attacks where adversaries create alternate ledgers, targeting clients or new participants unable to distinguish original from adversarial ledgers [13, 32, 33, 34]. These attacks are feasible because PoS/BFT protocols make block addition computationally inexpensive. Adversaries gaining access to honest node private keys—through either theft or bribery—can create alternate ledgers.

Attack vectors include:

- (1) *Stealing*. Adversaries may steal node keys, a widespread attack resulting in losses up to \$200 million [35, 36, 37].
- (2) *Bribing*. Adversaries may bribe honest nodes to sell private keys, particularly nodes that no longer have system stake. This bribery attack exploits the Tragedy of the Commons [38], where rational participants opt for additional incentives by selling keys.

With compromised keys, adversaries can fork original ledgers at specific block numbers and create adversarial ledgers with alternate blocks. As nodes frequently leave and join decentralized systems [39, 40], adversaries can present adversarial ledgers to new nodes that cannot distinguish authenticity.

Analysis of Existing Countermeasures and Their Remaining Gaps. Prior work has proposed three main families of countermeasures against long-range attacks. Each leaves specific security gaps under realistic adversary models.

- (1) Key-evolving cryptographic techniques [13, 14]. Participants periodically generate new key pairs and discard old ones.

An adversary can still collect old keys over time (e.g., by bribing nodes after they have left the system) and use them to create an adversarial ledger starting from a point before the key evolution. Key evolution only delays the attack; it does not make ledger rewriting computationally expensive. Moreover, it assumes that honest nodes reliably delete old keys – an assumption that may fail under compromise or side-channel attacks.

- (2) State checkpointing [41]. A trusted committee periodically signs the ledger state, and new nodes trust the latest checkpoint.

This introduces centralization – the committee becomes a single point of trust and a target for bribery or coercion. If the committee is large to maintain decentralization, checkpoint latency increases. Even with a committee, an adversary who compromises the committee can forge checkpoints. The approach trades off security against long-range attacks for trust in a small set of replicas.

- (3) Appending ledger state to Bitcoin’s PoW blockchain [42, 43]. Because Bitcoin’s ledger is computationally expensive to rewrite, anchoring checkpoints into Bitcoin makes adversarial ledger creation costly.

This solution inherits Bitcoin’s energy waste (competitive mining) and depends on Bitcoin’s security assumptions (e.g., $\geq 50\%$ honest mining power). It also requires a cross-chain communication mechanism, which introduces additional latency and complexity. Furthermore, it does not eliminate the waste of computational resources – miners still compete, and only one wins.

None of these approaches simultaneously achieves decentralized, computationally expensive ledger rewriting without massive resource waste. They either rely on trust assumptions (checkpointing committees), only delay the attack (key evolution), or inherit the inefficiencies of PoW (Bitcoin anchoring).

Our Contribution: Power-of-Collaboration (PoC) These challenges lead us to conclude that effective long-range attack solutions must: (1) avoid reliance on long-term private key security, (2) reduce computational resource waste, and (3) make ledger rewriting computationally expensive for attackers.

We introduce PoC, a protocol that, when appended to PoS/BFT systems, achieves these goals. PoC works non-invasively on underlying consensus outputs with minimal performance impact. It advocates collaborative mining where all miners work together on compute-intensive puzzles rather than competing.

Comparison with Mining Pools PoC’s security guarantees hold under the following assumptions, which are standard and weaker than those required by prior solutions:

Adversarial computational power is bounded to less than 50% of the total mining power (the same as in PoW). Under this assumption, recreating a chain of length b requires at least $(1/2)^b$ probability of success for the adversary – exponentially decreasing with chain length.

Network is eventually synchronous (the standard partial synchrony assumption for BFT consensus). PoC does not require synchronous network during normal operation; timers and slice-shifting handle asynchrony.

Honest miners follow the protocol and do not collude with adversaries. This is the standard assumption in Byzantine fault tolerance.

Compared to key-evolving cryptography, PoC does not rely on honest nodes deleting old keys – even if old keys are compromised, the adversary still needs to perform computational work to rewrite the ledger. Compared to checkpointing, PoC is fully decentralized; no trusted committee is required. Compared to Bitcoin anchoring, PoC eliminates competitive mining, reducing the time to find a nonce by up to $29\times$ for the same difficulty (Section 2.9). Thus, PoC offers a stronger security guarantee (computationally expensive ledger rewriting) under weaker assumptions (no trusted committee, no key deletion) while improving practicality (less resource waste, lower latency).

Experimental Validation We demonstrate PoC’s effectiveness by appending it to multiple decentralized systems. Experiments with Apache ResilientDB [44] show PoC sustaining 1,000 blocks per second on 128 replicas, requiring $29\times$ less mining time than Bitcoin’s PoW. Using the Diablo [45] benchmarking framework, PoC appended to Diem [46], Algorand [28], Ethereum [27], and Quorum [47] shows minimal throughput impact ($\approx 10\times$).

2.2. Preliminaries

We adopt standard communication and failure models employed by most consensus protocols [29, 31, 48]. We assume a decentralized system $S = R, C$ with replicas set R (n_R total, f_R arbitrarily behaving) and clients set C (arbitrarily many malicious). The $n_R - f_R$ honest replicas follow the protocol and remain live.

Miner Staking. Like PoS systems [28, 49], PoC requires knowledge of total miners participating. Miners must stake resources to participate, determining their required work. We denote miners' set M with total n_M and at most f_M malicious ($n_M \geq 2 * f_M + 1$). Miners and replicas may be the same nodes but are treated separately for exposition.

Advanced Adversary model. Standard cryptographic primitives (MAC, digital signatures) and collision-resistant hash functions are employed. The standard adversary model allows corruption of at most f_R replicas and f_M miners, with message delay and reordering capabilities [28, 29, 50]. An advanced adversary model additionally assumes private key access to all replicas/miners, enabling long-range attacks.

PoC offers Sybil resistance through resource staking. Miners are pseudo-anonymous, identified by public keys, with unique identifiers assigned.

Protocol Guarantees. The underlying PoS/BFT protocol must provide:

Safety.: If two honest replicas order transactions T and T' at sequence number k , then $T = T'$.

Liveness.: Client transactions eventually receive responses.

2.3. The Threat Model: Understanding Long-Range Attacks

2.3.1. PoS and BFT Consensus. PoS protocols allow nodes to add blocks proportionally to invested stake [28, 49], with higher stake increasing block proposal probability. BFT protocols like PBFT [29] and HotStuff [11] designate leaders that propose blocks, followed by multi-round message exchanges ensuring quorum support.

2.3.2. Long-range attack on PoS and BFT. Long-range attacks affect both PoS [51, 52] and BFT [13, 32] protocols. Because block addition is computationally inexpensive, adversaries with compromised keys can fork original ledgers at specific heights, creating adversarial ledgers. As nodes frequently leave and join systems [39, 40], adversaries can present adversarial ledgers to new nodes unable to distinguish authenticity.

Example 1 illustrates this:

EXAMPLE 1. Assume that a decentralized system $\mathcal{S}$ has the following PoS ledger: $\mathcal{B}_1, \dots, \mathcal{B}_k, \dots, \mathcal{B}_n$. Say malicious nodes get access to the private keys of all the honest nodes and decide to create an adversarial

ledger, starting from the k -th block. Once it is the turn of malicious nodes to propose new blocks, they reveal the following adversarial ledger: $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}'_k, \dots, \mathcal{B}'_n, \mathcal{B}'_{n+1}$. Any new node joining $\mathcal{S}$ cannot distinguish between these two ledgers and will choose the longest chain. Similarly, some existing honest nodes, if bribed, may decide to forfeit their ledger and switch to the malicious ledger. Moreover, as time passes, with old nodes leaving the system and new nodes unable to distinguish, a hard-working adversary may be able to affirm the adversarial ledger as the original ledger.

While stealing distributed keys is difficult, bribery presents a rational attack vector—validators with no remaining stake may sell private keys for incentives [38]. Solutions must therefore not rely on long-term private key security and must make ledger rewriting computationally expensive.

2.3.3. Proof-of-Work Consensus. PoW miners select client transactions, pack them into blocks with headers containing previous block hash, Merkle root, difficulty, and nonce [42, 53]. Miners continuously select random nonces until satisfying digest

$$(2.1) \quad \text{digest}(\text{prev} \parallel M_{\mathcal{B}} \parallel \eta) < D$$

The first miner discovering a valid nonce broadcasts the block, and other miners build upon it.

PoW faces two limitations: (1) all but one miner waste computational resources, and (2) multiple miners may find valid nonces, temporarily forking the ledger. Difficulty D dynamically adjusts to maintain target block production rates.

2.3.4. Centralized Pooled Mining. Mining pools allow miners to pool resources, increasing nonce discovery chances [42, 43]. Centralized pools are managed by organizations that create blocks, determine sub-problems, and distribute rewards proportionally after deducting management fees.

While ensuring regular payouts, centralized pools sacrifice decentralization. Pool controllers receive fees and determine rewards, punishments, and membership. Moreover, pools compete with each other, maintaining computational waste.

Decentralized pools eliminate centralized controllers but have failed due to accountability issues—the last block added by any decentralized Bitcoin mining pool was in 2019 [41, 42].

2.4. Our Proposal: The Power-of-Collaboration (PoC) Protocol

PoC aims to guard POS/BFT systems against long-range attacks while providing:

- G1 **Computationally expensive ledger rewriting.** Complex puzzle solving makes adversarial ledger overwriting expensive.
- G2 **Reduced computational waste.** Collaboration ensures miners work on same block with distinct search space subsets.
- G3 **Fairness.** Incentives are distributed among all miners, even without nonce discovery.
- G4 **Accountability.** Penalties for miners failing to find valid nonces present in their search spaces.

Before we describe the design of PoC, we discuss some of the possible solutions and their limitations.

Version 1. Let us assume a decentralized system running a PoS/BFT consensus protocol employs a PoW consensus subsystem to guard itself against long-range attacks. Each batch of transactions that the PoS/BFT protocols commit is forwarded to the PoW subsystem to add to the ledger maintained by PoW miners. Each miner M follows the PoW protocol: creates a PoW block that includes one or more committed batches, a transaction that transfers incentive to its account, the hash of the previous block $prev$, and initiates the search for a valid nonce. When M finds the nonce, it broadcasts its block and the nonce to the other miners. If another miner M' receives a block/nonce, it starts building/mining the next block and includes the hash of the received block/nonce as the previous block. This solution faces the following two limitations: (i) all but one miner waste their computational resources (lack of fairness), and (ii) more than one miner can find a valid nonce, which can temporarily fork the ledger and lead to a subsequent increase in the hardness/difficulty of finding the nonce (Section 2.3.3).

Before we describe the design of PoC, we discuss some of the possible solutions and their limitations.

Version 2. Next, we replace the PoW consensus subsystem with a centralized mining pool, where the pool operator receives the next committed block and creates sub-problems for the pool's miners to mine. This solution does not face any of the above limitations. However, this solution illustrates control by a single operator/organization, which receives fees for its services and decides the incentives/penalties for the pool's miners.

Version 3. Finally, we replace the centralized mining pool with a decentralized mining pool, where miners decide to coordinate with each other without any operator. The following are the challenges for any decentralized mining pool-based solution: (i) which miner decides the content of the block, (ii) how to fairly distribute rewards among the miners, and (iii) how to detect and penalize a malicious miner that delays block mining by not performing designated tasks.

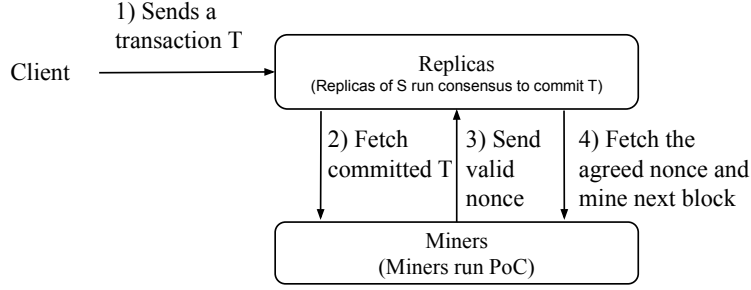


FIGURE 2.1. Transactional flow in a system $\mathcal{S}$ +PoC.

Overview. Our solution should offer the four appealing properties (G1 to G4). Consequently, we design PoC that requires no centralized organization and guards a PoS/BFT protocol from long-range attacks. In Figure 2.1, we illustrate the transactional flow.

(1) Transaction ordering and communication. PoC expects that the underlying PoS/BFT protocol reaches consensus on client transactions among its replicas and forwards every committed batch of transactions to the PoC miners.

(2) Block creation and mining. Once miners are ready to mine, they select a set of ordered batches to form a block. To allow miners to collaborate and reduce computational resource wastage, PoC ensures that all the miners are mining identical blocks, and each miner searches for the valid nonce on a unique subset of the search space. This collaboration helps us meet property G2.

(3) Nonce discovery and attestation. Once a miner discovers a nonce, it broadcasts the nonce to everyone. To ensure that there are no forks of the ledger, PoC requires the decentralized system’s PoS/BFT protocol to attest a nonce. Note: this recursive dependency only helps to select a valid nonce without requiring multiple rounds of communication among the miners; the same task can be done by running a consensus on the nonce among the miners.

(4) Reward distribution. Once a miner finds a valid nonce, each miner receives an incentive. This reward distribution allows PoC to ensure fairness (property G3)).

(5) Failure detection and progress. Byzantine miners may not search for nonce in their subset of the search space. If the valid nonce is present in this subset, then no miner will ever discover it. PoC allows miners to independently discover such malicious attacks and switches honest miners to different subsets to facilitate the discovery of the nonce.

(6) Penalty. PoC guarantees accountability by penalizing malicious miners that failed to find a valid nonce (property G4).

Next, we discuss our PoC assuming no attacks (*good case*). Later, we explain how we handle malicious attacks.

2.4.1. Client Transaction Ordering. Prior to running PoC, we expect the blockchain system S to reach consensus on client transactions. For PoC, the consensus run by S is a black box. The system S is free to run any consensus protocol of its choice. It needs to only provide PoC miners with committed batches of transactions.

Our use of the term committed implies that each committed batch of transactions has been accepted by a quorum of replicas of S and will persist across adversarial failures. For instance, in BFT protocols like PBFT [29] and HOTSTUFF [11], blocks are committed when they have quorum certificates from $2f_{\mathcal{R}} + 1$ replicas. In several other blockchain systems, a batch of transactions is assumed committed if it is at a specific depth in the blockchain. A depth indicates the position of the block in the blockchain; the larger the depth of a block B , the greater the number of blocks that succeed B and the harder it is for another fork to overtake this chain.

The assumption of PoC miners working with only committed batches has two advantages: (1) It frees PoC from having any knowledge on the consensus run by S , and (2) PoC miners will not waste their resources on batches that may not persist.

2.4.2. Chain Communication. Once the blockchain system S has committed a block, we require it to forward the committed block to PoC miners to log it in the ledger. Like popular blockchain systems, Bitcoin and Ethereum, we employ the gossip protocol for broadcasting a block. Here, we are making a simplifying assumption that each node is connected to a sufficient number of honest nodes because, in gossip protocol, each node forwards the message to only its neighbors. Alternatively, the replicas of S can employ either the Information Dispersal Algorithm [54] or Byzantine Reliable Broadcast [55] if they cannot assume a uniform distribution of honest nodes. Each miner accepts a committed block once it receives it from $f_{\mathcal{R}} + 1$ replicas in S , which assures this miner that it did receive a committed block.

2.4.3. Collaborative Mining. PoC introduces the notion of collaborative mining to log each committed block in the ledger such that rewriting the ledger is computationally expensive for an adversary.

Collaborative mining, like centralized pooled mining, should ensure that each miner works on a unique sub-problem so that miners do not waste their computational resources. Thus, we divide the POW hash computation into $n_{\mathcal{M}}$ disjoint sub-problems and requires each miner to work on a *distinct predetermined sub-problem*. Like existing POW systems, we compute the solution space of hash computation for the miners [1].

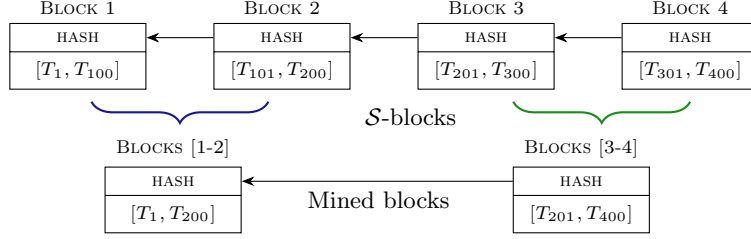


FIGURE 2.2. Illustrating how $\sigma = 2$ contiguous $\mathcal{S}$ -blocks produced by replicas of $\mathcal{S}$ are aggregated into one mined block of PoC. Here, each $\mathcal{S}$ -block includes 100 transactions.

PoC miners have to compute a SHA-256 hash, which is represented as a 32-byte hexadecimal value. Thus, the solution space $\mathbb{S}$ comprises of 2^{256} possible values. We divide $\mathbb{S}$ into $\mathbf{u}$ *slices*; the size of each slice is a system parameter. Given $\mathbb{S}_1, \mathbb{S}_2, \dots, \mathbb{S}_{\mathbf{u}}$ slices, the following holds:

$$\mathbb{S}_1 \cap \mathbb{S}_2 \cap \dots \cap \mathbb{S}_{\mathbf{u}} = \emptyset \quad \text{and} \quad \mathbb{S}_1 \cup \mathbb{S}_2 \cup \dots \cup \mathbb{S}_{\mathbf{u}} = \mathbb{S}$$

PoC assigns each miner one or more consecutive slices based on its stakes. We make a simplifying assumption that each slice is assigned to a miner in $\mathcal{M}$. We discuss the slice assignment scheme in more detail in Section 2.4.5. Given the difficulty D , each miner computes a hash till it reaches the target 32-bit SHA-256 hash (refer to Equation 2.1). This requires each miner to find a nonce η in its set of slices.

Next, we describe the PoC consensus.

2.4.4. PoC Protocol Steps. From an outside view, our PoC protocol works in rounds, and within each round, each miner attempts to find a valid nonce in its pre-determined slices. Next, we explain the PoC protocol under the assumption that each miner knows the next block to mine (Section 2.4.3) and has received this block through chain communication (Section 2.4.2). In the case a miner does not have access to the next block to mine, it can communicate with the other miners to receive all the missing committed blocks.

Block Creation. When a PoC miner receives a block from $\mathbf{f}_{\mathcal{R}} + 1$ replicas, it adds that block to the *list of pending blocks*. The list of pending blocks is an ordered list of committed blocks that a miner is yet to add to the ledger; these blocks are ordered by the sequence number assigned by the PoS/BFT consensus protocol running at $\mathcal{S}$. As the difficulty D of mining a block sets the rate at which blocks are added to the PoC ledger, which in turn impacts the system throughput and latency, at higher difficulties, PoC has a lower throughput than the PoS/BFT consensus protocol. We present a discussion on PoC's difficulty in Section 2.6. Consequently, PoC miners have an ever-growing list of pending blocks, as the rate at which they add these blocks is slower than the rate at which they receive them from the PoS/BFT protocol.

To reduce this gap between total blocks received and blocks mined, PoC allows miners to batch a set of committed $\mathcal{S}$ -blocks, thereby, each *mined block* includes $\sigma > 0$ committed $\mathcal{S}$ -blocks.¹ The value of σ is a protocol parameter. PoC requires miners to only aggregate σ consecutive $\mathcal{S}$ -blocks from the pending list, which is essential for maintaining the block ordering by $\mathcal{S}$. For the sake of discussion, let us assume that each $\mathcal{S}$ -block is assigned a monotonically increasing sequence number k and each mined block is assigned a sequence number b . If the last block mined by miners had sequence number $b - 1$ and the sequence number of last $\mathcal{S}$ -block added to the $(b - 1)$ -th block is $k - 1$, then in the b -th block, each miner will aggregate the following blocks: $k, k + 1, \dots, k + \sigma$. Aggregating $\mathcal{S}$ -blocks in this way is safe as these blocks are already committed by replicas of $\mathcal{S}$. We illustrate this next.

EXAMPLE 2. *In Figure 2.2, the replicas of $\mathcal{S}$ have committed four $\mathcal{S}$ -blocks (using the consensus protocol of their choice) starting with sequence number 1. Assume $\sigma = 2$, then each PoC miner aggregates 2 consecutive $\mathcal{S}$ -blocks into a mined block.*

Each mined block includes a Merkle root of all the transactions; as each $\mathcal{S}$ -block contains a Merkle root of all the transactions, a miners M generates the Merkle root for the mined block by simply hashing the Merkle roots of all the aggregated blocks.

Nonce Discovery. Once a miner knows the valid nonce for $(b - 1)$ -th block, it can initiate the search for the nonce for b -th block. Assuming the miner M_i knows the set of slices it needs to mine (Section 2.4.5), which we denote as $\mathbb{S}_i$, M_i initiates nonce discovery. This search process for a valid nonce η requires M_i to iterate over all the values in its slices $\mathbb{S}_i$. **Nonce Announcement.** When a miner M_i finds a valid nonce η , it creates a message **NONCEFIND**, which includes η and broadcasts this message to all the miners. When a miner M_j receives a **NONCEFIND** message, it terminates the process of nonce discovery if the received nonce is valid. Next, each miner that has access to a valid nonce gossips this nonce to the replicas of $\mathcal{S}$.

Nonce Attestation. Next, PoC leverages the underlying POS/BFT consensus protocol to attest the discovered nonce. Specifically, when a replica R of $\mathcal{S}$ receives matching **NONCEFIND** messages from $f_{\mathcal{M}} + 1$ miners, it creates a transaction that includes the received **NONCEFIND** message as its data. Whenever it is the turn of R to propose a new block, and if an $\mathcal{S}$ -block containing the **NONCEFIND** message for the b -th mined block is yet to be proposed, R proposes a new block that includes this message. We expect honest replicas to prioritize nonce transactions over others to prevent delays in adding newly mined blocks to the ledger.

¹For disambiguation, we use $\mathcal{S}$ -blocks to denote the blocks produced by replicas of $\mathcal{S}$ and mined block to denote the block produced by miners.

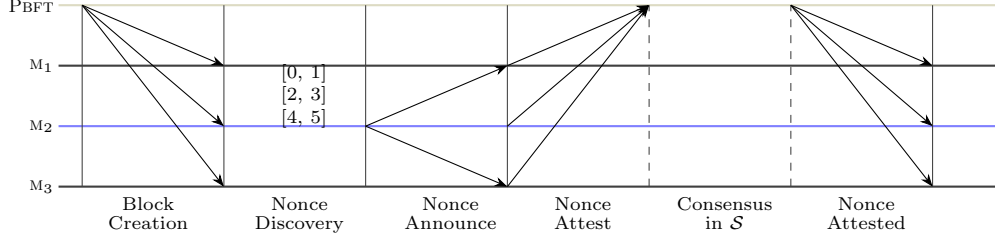


FIGURE 2.3. PoC protocol with miners $\mathcal{M} = \{M_1, M_2, M_3\}$. The solution space $\mathcal{S} = [0, 5]$ is divided into three slices ($[0, 1]$, $[2, 3]$, $[4, 5]$). Assume the valid nonce 2 and miner M_2 discovers it in its slice.

Chain Append. When a miner M_i receives the valid nonce for the b -th block from replicas of $\mathcal{S}$, M_i appends this block to its local ledger and marks the nonce discovery process for the b -th block as complete. Post this, M_i executes the *reward* transactions in the block, which helps to distribute the reward (Section 2.4.5). Finally, M_i begins mining the next block.

We use the following example to illustrate PoC mining.

EXAMPLE 3. In Figure 2.3, the solution space $\mathcal{S} = [0, 5]$ is divided among three miners. The three slices are: $\mathcal{S}_1 = [0, 1]$, $\mathcal{S}_2 = [2, 3]$, and $\mathcal{S}_3 = [4, 5]$. Assume the valid nonce is 2 and it lies in the slice of miner M_2 . Once M_2 discovers the nonce in its slice, it broadcasts the nonce to all the miners, following which each miner requests the replicas of $\mathcal{S}$ to attest this nonce.

Multiple Nonces. In rare scenarios, two or more miners may find valid nonces that help to reach the expected hash. Specifically, miners M_i and M_j may both discover a valid nonce in their respective slices. This situation is *not unique* to PoC, even complex POW computations can have multiple solutions. We need to guarantee that all the miners select the same nonce, which is trivial for PoC as each nonce is attested by replicas of $\mathcal{S}$. If the next proposer for an $\mathcal{S}$ -block receives two or more NONCEFIND messages with distinct valid nonces, it selects one of them as the solution. When, eventually, this block becomes committed, all the PoC miners will have access to the same nonce for block b .

2.4.5. Staking and Rewards. PoC, like POW, rewards its miners with incentives for their participation in the mining process; miners expend their computational resources and would only do so if they make some profit. However, unlike POW, PoC wants to ensure fairness by rewarding each miner for collaboration even though the valid nonce was not in its slice. In PoC, we reward each miner in proportion to the number of slices in its slice set. As stated in Section 2.4.3, in PoC, there are a total of $\mathbf{u}$ slices. We assume the following: (1) Each of these $\mathbf{u}$ slices is assigned to a unique miner. (2) Each miner is assigned a set of

consecutive slices. We require a miner to invest its monetary resources in exchange for each slice it holds. Like existing PoS systems, we term this investment by a miner as *staking* as the miner no longer has access to its invested monetary resources [27, 28, 49].

The economics of converting actual currency into an online tradable commodity is a problem faced by every decentralized system, for which current literature includes several ad hoc solutions. We view this as a non-trivial problem and recognize that it requires a rigorous economic analysis, which, unfortunately, is beyond the scope of this dissertation. Thus, for simplicity, we assume that PoC builds on top of some token Ψ , where Ψ is the cost of purchasing a slice. Each miner exchanges its currency for a set of Ψ . If a miner M_i wants e slices in its set, M_i stakes $e \times \Psi$ tokens. This information is added to the first block of the PoC ledger, which is often referred to as the genesis block. Specifically, PoC-genesis blocks stores the following information: (1) total number of miners ($\mathbf{n}_{\mathcal{M}}$), (2) number of tokens staked by each miner, (3) a public key for each miner, and (4) slice to miner mapping.

During PoC’s collaborative mining, each miner refers to this genesis block to identify the slices it is responsible for mining. Once a miner receives a valid nonce from the replicas of $\mathcal{S}$, it adds a reward to the account of each miner. Like existing systems [1, 27], we assume that the reward is proportional to the fees paid by the clients; each client pays a fee to add its transaction to the ledger and this fee is divided among the miners in proportion to their number of slices. For example, if the client for transaction T pays $\diamond$ tokens and a miner M_i has e slices in its set, M_i receives $\frac{e \times \diamond \times \Psi}{u}$ tokens as a reward. We maintain each miner’s account as a key-value pair in a NoSQL database replicated across all miners/replicas; the key field represents the public key (logged in the genesis block) of each miner, while the value field represents the token balance.

2.5. Ensuring Robustness: Handling Malicious Behavior in PoC

Unlike a centralized mining pool, where the pool operator oversees the activity of all the miners and rewards/penalizes them for their actions, PoC assumes a decentralized setup without a centralized operator. Thus, PoC needs to provide protection from malicious attacks and guarantee accountability while ensuring that it makes it computationally expensive for the adversary to overwrite the ledger using long-range attacks.

First, we expect that the underlying PoS/BFT protocol guarantees safety and liveness properties stated in Section 2.2. Thus, it should be capable of handling attacks by a standard adversary.

Second, a standard adversary can only attack the PoC in the following ways:

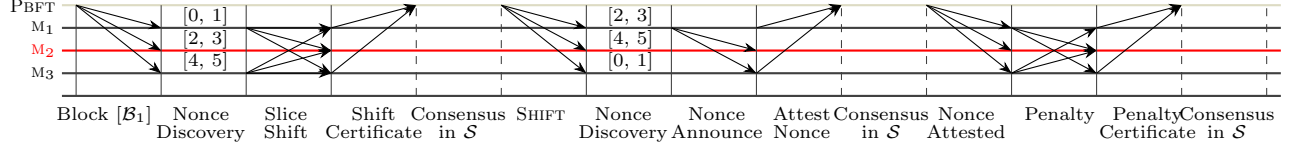


FIGURE 2.4. Slice shifting procedure: assume 2 is the valid nonce and is present in the slice of malicious miner M_2 . M_2 fails to broadcast 2, which triggers slice shifting; and with help of replicas of $\mathcal{S}$, once 2 is found, it is penalized.

A1 As multiple nonces can satisfy Equation 2.1 and if Byzantine miners are fortunate enough to find two such nonces for a mined block, they can equivocate by sending each nonce to only a subset of honest miners and replicas.

A2 A miner decides to not participate in the PoC's collaborative mining or if a miner discovers a nonce in its slice, it decides to not send it to other miners.

Third, an advanced adversary that has access to the keys of any replica/miner can use these keys to forge any message.

Final, each distributed system need to also guard against denial-of-service attacks. To prevent these attacks, we follow the best practices suggested by prior works [56, 57] and assume that the replicas/miners use one-to-one virtual communication channels, which can be disconnected if needed.

Next, we discuss how we handle attacks by a standard adversary on PoC and attacks by advanced adversaries on the entire system.

2.5.1. Standard Adversary. As replicas of $\mathcal{S}$ help in selecting a nonce, Attack A1 is trivially resolved. Each replica R selects a nonce for the mined block at height b only after it receives $f_M + 1$ matching NONCEFIND messages. Whenever it is the turn of R to propose a new block, and if an $\mathcal{S}$ -block containing the nonce for the b -th mined block is yet to be proposed, R proposes a new block that includes this nonce. Thus, honest miners will receive only one nonce. If there aren't sufficient matching messages, then no nonce will be selected and this will lead to the eventual detection of some Byzantine miners.

To resolve Attack A2, next, we present our *slice shifting* protocol, which penalizes miners for malicious behavior.

2.5.2. Slice Shifting. Each honest PoC miner is expected to search for a nonce in its set of slices until it receives a valid nonce from another miner or it has exhausted its slices. A malicious miner may not follow this behavior; it may want to disrupt the process of nonce finding. If the malicious miners are fortunate and the nonce is present in their slices, then honest miners may never receive the nonce, and PoC will come to

a halt. We aim to quickly resolve such a situation, prevent further performance degradation, and provide accountability. We do so by running our *slice shifting* algorithm that switches miner slices under failures. This is done with the aim that when an honest miner has access to the slice held by a malicious miner, it can discover the nonce in that slice and broadcast it to other miners.

Next, we illustrate slice shifting protocol through an example. Post this, we will explain the protocol in detail.

EXAMPLE 4. *Figure 2.4 illustrates the slice shifting protocol. Assume the nonce lies in slice $[2, 3]$ and miner M_2 fails to announce the nonce. Eventually, honest miners timeout while waiting to receive a valid nonce and trigger slice shifting protocol. These miners must create a certificate to prove that a majority wants to do slice shifting. They send this certificate to the replicas of $\mathcal{S}$ for attestation. Post this, each miner works on a new slice. Once, they discover the nonce, they initiate the process of penalizing M_2 .*

Timer Initialization. PoC miners, who have exhausted their slice search space and do not have a valid nonce, need a mechanism to make progress. We follow existing BFT works and require each miner to set a *timer* before it starts mining a slice. Specifically, each miner sets a timer δ for the b -th block and stops δ when it has a valid nonce for the b -th block.

Malicious Miner. If M 's timer δ expires and it does not have access to a valid nonce, M announces to all the other miners that it wishes to initiate the *slice shifting* protocol. The slice shifting protocol runs for at most $f_{\mathcal{M}}$ rounds and deterministically switches slices assigned to each miner. A round of slice shifting only takes place when at least $f_{\mathcal{M}} + 1$ miners request to do so. Specifically, when a miner M_i timeouts, it creates a message $\text{SHIFT}(b, \mathbf{r})$ and broadcasts this message to all the other miners. Here, $\mathbf{r}$ represents the slice shifting round, which is initially set to *zero*. When M receives SHIFT messages from $f_{\mathcal{M}} + 1$ distinct miners, it requests the replicas of system $\mathcal{S}$ to help reach a consensus on slice shifting. To make this request, each miner must broadcast a certificate $\mathcal{C}$ that includes $f_{\mathcal{M}} + 1$ SHIFT messages.

Shift Attestation. When replicas of $\mathcal{S}$ receive a signed $\mathcal{C}$ from $f_{\mathcal{M}} + 1$ PoC miners, they agree to attest this slice shift. This attestation requires the replicas to run consensus on this certificate $\mathcal{C}$; the next proposer for a $\mathcal{S}$ -block includes $\mathcal{C}$ as a transaction in its block. *Note:* consensus on $\mathcal{C}$ is like consensus on any transaction where $\mathcal{C}$ acts as the transactional data. Post consensus, all the replicas gossip this block to the miners. Once a miner M receives a $\mathcal{S}$ -block from $f_{\mathcal{R}} + 1$ replicas that include a committed certificate $\mathcal{C}$, it assumes it is time to shift its slices. Following this, it increments the shift round $\mathbf{r}$ by one and mines the next slice. If in shift round $\mathbf{r}$, M_i was responsible for mining slices $\{\mathbb{S}_i, \mathbb{S}_{i+1}, \dots, \mathbb{S}_j\}$, in round $\mathbf{r} + 1$, M_i will mine

slices $\{\mathbb{S}_{i+1}, \dots, \mathbb{S}_j, \mathbb{S}_o\}$, where $o = (j + 1) \bmod \mathbf{u}$, and $\mathbf{u}$ is the total number of slices. Again, before mining for round $\mathbf{r} + 1$, M_i restarts the timer δ for the k -th block. It is possible that the timer δ again timeouts, due to more failures. In such a case, M_i would need to initiate another round of slice shifting. However, under a standard adversary and reliable network, for each mined block, we need to run the slice shifting protocol only $\mathbf{r} = \mathbf{f}_{\mathcal{M}}$ times.

No nonce – Merge. Although infrequent, PoC miners may encounter cases where no nonce satisfies Equation 2.1. This no nonce situation is scarce in PoW because miners work on different blocks, but it is possible in PoC because all the miners collaborate on the same block. We resolve this situation as follows.

If even after $\mathbf{r} = \mathbf{f}_{\mathcal{M}}$ rounds honest miners do not have access to a valid nonce, we require the miners to terminate their search for the nonce and initiate the *merge* process, which requires miners to mine two or more consecutive mined blocks together, with the hope that mining multiple blocks together increases the probability of finding a nonce. For instance, for the b -th block, if a miner M receives a certificate $\mathcal{C}$ from $\mathcal{S}$ replicas, which has shift round $\mathbf{r} = \mathbf{f}_{\mathcal{M}}$, M concludes that no valid nonce exists for the b -th block. Following this, M creates a new block that merges contents of the b -th and $(b + 1)$ -th blocks. This merged block now serves as the b -th block and miners attempt to find the nonce for this block. The merged block includes a Merkle root, which is the hash of all the transactions in b and $(b + 1)$ -th blocks.

Penalty for Slice Shifting. Frequent slice shifting due to malicious miners is detrimental to the performance of PoC; it forces honest miners to do more work and wastes their resources. To discourage these attacks, we *penalize* malicious miners. We require each miner to track the number of shifts ($\mathbf{r}$) it took to find a valid nonce and to identify the miners that failed to perform this task.

In PoC, identifying malicious miners responsible for $\mathbf{r}$ shifts is a trivial task for honest miners. When a miner M_i receives a valid nonce for the b -th block in shift round $\mathbf{r}$ ($1 \leq \mathbf{r} \leq \mathbf{f}_{\mathcal{M}}$), it identifies malicious miners based on the slice containing the valid nonce. Let $\mathbb{S}_j$ be the slice, then M_i looks into the genesis block to find the initial assignment of the slice. Using this information, M_i determines the miners who held this slice in subsequent $\mathbf{r} - 1$ rounds and adds these miners to the set of malicious miners $\mathcal{M}_{mal}$.

To penalize these malicious miners, we again invoke the replicas of $\mathcal{S}$. Once a miner M has the knowledge of set $\mathcal{M}_{mal}$, it sends a message $\text{PENALTY}(\mathcal{M}_{mal}, b, \mathbf{r})$ to all the miners. Once M receives PENALTY messages from $\mathbf{f}_{\mathcal{M}} + 1$ miners, it creates a certificate (like it created for slice shifting) that includes these PENALTY messages and broadcasts this certificate to each replica in $\mathcal{S}$.

When the replicas of $\mathcal{S}$ receive a PENALTY certificate signed by $f_{\mathcal{M}} + 1$ distinct miners, they add it to a new block for consensus. Post consensus, honest miners penalize malicious miners by deducting their account balances.

2.5.3. Advanced Adversary: Long-Range Attacks. An advanced adversary, unlike a standard adversary, has access to the private keys of honest replicas/miners. It can use these keys to forge any message that has a digital signature. In the context of our system, the following attacks are possible: (1) Multiple committed $\mathcal{S}$ -blocks with the same sequence number, (2) Forgery of nonce, shift, or penalty messages by honest miners. (3) Forgery of $\mathcal{S}$ -blocks, containing nonce, shift, or penalty transactions. If an adversary can compromise the keys of honest replicas/miners, PoC *can not guarantee* fairness and accountability; honest miners may get penalized. However, PoC only guarantees that it is computationally expensive for the adversary to rewrite the existing ledger.

To illustrate that PoC guards against long-range attacks, we theoretically analyze its hardness. We follow Example 1 where starting from the k -th block, malicious parties want to rewrite the ledger. To do so, malicious replicas would first create an alternate set of $\mathcal{S}$ -blocks using the compromised private keys of honest parties. Next, these malicious replicas would forward these $\mathcal{S}$ -blocks to the malicious miners, who will attempt to forge the PoC ledger by creating new mined blocks. These malicious miners will not receive any help from honest miners as they have a local copy of the ledger.

As the combined computational power of honest miners is more than malicious miners, it is safe to assume that in any round of PoC, malicious miners control at most 50% slices and have a 50% chance of finding a valid nonce.² So, $\frac{1}{2}$ is the probability that malicious miners find the solution for an “alternate block” in their set of slices. The probability that malicious miners find the solution for the b consecutive alternate blocks in their set of slices is $(\frac{1}{2})^b$. If $b \geq 7$, the probability is 0.7%. Clearly, if Byzantine miners have to create a large number of alternate blocks, it is impossible for them to find the nonces by just mining their own set of slices. Alternatively, they can search for the nonce in all the slices (entire solution space) but this at least doubles their work.

Next, we measure the actual time required to create an alternate chain by taking into account two parameters: (1) the age of the chain α (in months) and (2) the hashing power ratio of the malicious miners (m) over the honest miners (h), $\frac{m}{h}$. Essentially, it takes $\alpha \times \frac{h}{m}$ months for the malicious miners to reconstruct

²Assuming that an adversary has access to the private keys of honest nodes is orthogonal to the assumption that an adversary also has more computational power than honest nodes. If an adversary controls more computational power, then it can rewrite even the PoW ledger.

an α -month-old chain given their overall hashing power ratio of $\frac{m}{h}$. For example, if the malicious miners hold $\frac{1}{2}$ of the mining power and want to reconstruct a 1-year old chain, it would require two years' worth of computation to create an alternative chain of equal length.

During these two years, the following two things can happen: (1) The original chain will continue growing, which further increases the task of malicious miners. (2) The system is at a stall and honest miners/replicas detect that nothing is getting appended to the ledger and will leave the system. These arguments help us to demonstrate that simply compromising the private keys of honest miners/replicas is insufficient to launch a long-range attack on PoC.

2.6. Discussion

Difficulty. As discussed in Section 2.3.3, difficulty refers to the hardness of finding a valid nonce; it is the rate at which miners produce new blocks. For PoW consensus, its difficulty depends on the following two parameters: (1) the number of miners and the hardware technology available to the miners, and (2) the probability of ledger forks. In PoC, replicas work on the same block, under the standard adversary model, so there is no possibility of forks. Consequently, we need to increase/decrease PoC's difficulty based on the number of miners and the characteristics of the latest hardware technology.

Miner Reconfiguration. PoC assumes that reconfigurations are possible; we allow old miners to leave and new miners to join the system. Like most PoS/BFT systems [11, 46, 58, 59], we assume that (1) each miner only leaves at the boundary of consensus; no miner leaves during an ongoing consensus. (2) despite reconfigurations, less than 50% of the total number of miners are malicious. (3) reconfigurations take place under a reliable network.

If a miner M wants to join or leave PoC mining, it creates a message $\langle \text{JOINMINER}(pk, \Psi) \rangle_M$ and $\langle \text{LEAVEMINER}(pk) \rangle_M$, respectively, and broadcasts this message to all the replicas of $\mathcal{S}$. We use pk to denote the public key of the miner, and Ψ to denote the monetary resources a new miner wants to stake. Ψ also helps to calculate the total number of slices for the new miner.

The next proposer (or leader) on receiving a join/leave request, needs to do the following: (1) create a transaction that includes the join/leave message received from the miner, (2) check if the difficulty of the system needs to be changed and if so, create a transaction that includes the updated difficulty of the system. (3) redistributes the slices among the miners and creates transactions that map each miner to its set of slices. The proposer collects these transactions and proposes a special $\mathcal{S}$ -block. Once replicas reach a consensus on this special block, they forward it to the miners. When miners finish appending this $\mathcal{S}$ -block to the ledger,

they create/delete accounts for the joining/leaving miner and update their set of slices. Note: the miner who sent a LEAVEMINER message needs to participate till the end of this step, and only then will its stake be released and it can leave the system..

Challenges for PoC. Although PoC helps to meet the four desirable properties G1 to G4, it faces two challenges:

(1) *Fortunate miner.* As discussed earlier, a Byzantine miner M may not follow the PoC protocol and skip searching for nonce in its set of slices. If M is fortunate and a valid nonce exists in the slice of an honest miner, then M will receive a reward without doing any work. Due to the collaborative nature of PoC, it is impossible to detect such a malicious miner.

(2) *Restricted access than PoW.* As described in Section 2.4.5, despite requiring miners to solve a puzzle like PoW, PoC requires miners to stake their resources and undergo a reconfiguration protocol before they can join or leave the PoC mining.

2.7. Theoretical Guarantees: Security, Liveness Arguments

We now state and prove the security properties of PoC under the standard adversary model (Section 2.2). We assume that PoC is appended to a decentralized system $\mathcal{S}$, which provides the safety and liveness guarantees stated Section 2.2. PoC adopts the same partial synchrony model adopted in most consensus systems [11, 29]. It guarantees *safety* in an asynchronous environment where messages can get lost, delayed or duplicated. However, *liveness* is only guaranteed during the periods of synchrony [11, 29, 60]. PoC guarantees *fairness* for rewards and penalties only under synchrony.

Security Assumptions of PoC. PoC’s security against long-range attacks relies on the following explicit assumptions, which are strictly weaker than those required by prior countermeasures:

- Computational bound: The adversary controls less than 50% of the total mining power (i.e., number of slices). This is identical to Bitcoin’s security assumption. Under this bound, the probability that an adversary can find a valid nonce for a block faster than the honest miners is negligible for sufficiently deep chains.
- Network partial synchrony: Message delays are bounded after a Global Stabilization Time (GST), but the value of GST is unknown. Before GST, messages may be arbitrarily delayed or re-ordered. PoC’s timer-based progress (slice-shifting) ensures liveness even before GST, as shown in Lemma Lemma 3.

- Honest majority among miners: At least $n_M - f_M$ miners are honest and follow the protocol. This is the same assumption used in standard BFT and PoS systems.

Under these assumptions, we now prove safety, liveness, and fairness.

Safety Argument. We argue the safety of PoC by relying on the agreement property of the system $\mathcal{S}$. Intuitively, PoC ensures that every miner witnesses an identical sequence of discovered nonces. This coherence is achieved by sequencing each discovered nonce into the underlying PoS/BFT protocol. Consequently, irrespective of the number of solutions found to the Proof of Work (PoW) puzzle, all miners eventually converge to a unified nonce³.

THEOREM 1 (Safety). *No two conflicting blocks are settled by the PoC protocol. That is, if two honest miners M_i and M_j add blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ at sequence number b , then $\mathcal{B}_i = \mathcal{B}_j$.*

PROOF. Assume that honest miners M_i and M_j added two conflicting blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ at sequence number b . We know that each mined block has σ contiguous $\mathcal{S}$ -blocks (Section 2.4.4). So each mined block, including $\mathcal{B}_i$ and $\mathcal{B}_j$ have the same number of $\mathcal{S}$ -blocks. As both conflicting blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ have the same sequence number b , it is safe to assume that mined blocks at sequence number $\leq b - 1$ have the same set of $\mathcal{S}$ -blocks. This implies that $\mathcal{B}_i$ and $\mathcal{B}_j$ have at least one distinct $\mathcal{S}$ -block. Miners M_i and M_j must have received this distinct $\mathcal{S}$ -block from replicas of $\mathcal{S}$. This can only happen if replicas of $\mathcal{S}$ committed two distinct blocks at the same sequence number. But, this is a contradiction as we assume that $\mathcal{S}$ guarantees a globally consistent view of the transactions.

Similarly, each miner submits a NONCEFIND message to replicas in $\mathcal{S}$ when it has access to a valid nonce. The replicas of $\mathcal{S}$ add this message as a transaction in the next $\mathcal{S}$ -block. Assume that this NONCEFIND message is in blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ and it is the transaction on which miners differ. If this is the case, then replicas of $\mathcal{S}$ committed different nonce in their $\mathcal{S}$ -blocks, which contradicts our assumption on $\mathcal{S}$. $\square$

Liveness Argument. We argue the liveness of PoC during periods of synchrony.

LEMMA 1 (Commit Availability). *An honest miner eventually receives the k -th $\mathcal{S}$ -block committed by an honest replica.*

PROOF. We argue this lemma by induction over the serialized communication of committed $\mathcal{S}$ -blocks. Assuming a history of $k - 1$ committed $\mathcal{S}$ -blocks for which this property holds, we consider the k -th committed

³We note that Ethereum [27] uses a similar technique to ensure that all replicas observe the same set of signatures generated by its finality gadget [61].

$\mathcal{S}$ -block. When an honest replica has a committed $\mathcal{S}$ -block, it reliably broadcasts that block to the miners. It is thus guaranteed that an honest miner will receive all the committed $\mathcal{S}$ -blocks. The inductive base case involves assuming that all replicas are initialized with a committed genesis ($k = 1$) block, which we can ensure axiomatically. $\square$

LEMMA 2 (Nonce Search). *The first time an honest miner M_i obtains the b -th block containing $\sigma > 0$ committed $\mathcal{S}$ -blocks, it searches for a valid nonce η in slice $\mathbb{S}_i$.*

PROOF. Upon receiving the k -th committed $\mathcal{S}$ -block for the first time, correct miners wait to check if there are $\sigma > 0$ $\mathcal{S}$ -blocks available, since the last $\mathcal{S}$ -block added to the ledger, for mining. If so, M_i aggregates these $\mathcal{S}$ -blocks into mined block b and starts to search for a nonce in its set of slices. $\square$

LEMMA 3 (Shift Liveness). *If a correct miner does not find a valid nonce η in slice $\mathbb{S}_i$ to settle block b within time δ , another miner eventually tries it.*

PROOF. While finding nonce, if the timer δ expires for an honest miner, it broadcast a message SHIFT to other miners. When it receives SHIFT message from $\mathbf{f}_{\mathcal{M}} + 1$ miners it creates a certificate $\mathcal{C}$ comprising of these SHIFT messages and broadcasts this $\mathcal{C}$ to all the replicas. When an honest replica receives $\mathcal{C}$ from at least $\mathbf{f}_{\mathcal{M}} + 1$, it proposes it in the next block. The liveness property of $\mathcal{S}$ ensures that these messages are eventually committed, and Lemma 1 ensures that honest miners are eventually notified of the commit. On receiving a committed $\mathcal{S}$ -block with a certificate $\mathcal{C}$, an honest miner M_i resets its timer and restarts the nonce finding process in slices $\{\mathbb{S}_{i+1}, \dots, \mathbb{S}_j, \mathbb{S}_o\}$, where $o = (j + 1) \bmod \mathbf{u}$ ($\mathbf{u}$ is the total number of slices) if it has not already attempted to find a nonce for the block for $\mathbf{f}_{\mathcal{M}} + 1$. Otherwise, it assumes that no nonce can be found for this block and starts a new mining. As a result, honest miners keep shifting and searching for each other nonces until they are all found. $\square$

LEMMA 4 (Block Settlement). *All honest miners settle block b if a valid nonce η for block b exists.*

PROOF. When an honest miner M_i finds a nonce η , it broadcasts η to the other miners. Following this, each miner submits η to the replicas of $\mathcal{S}$. The liveness property of $\mathcal{S}$ ensures that η is eventually committed by all honest replicas. Lemma 1 then ensures that all honest miners obtain the corresponding committed $\mathcal{S}$ -block. We conclude the proof by noting that if the $\mathcal{S}$ -block contains a valid nonce η , correct miners will settle b . $\square$

THEOREM 2 (Liveness). *Each committed block at $\mathcal{S}$ is eventually settled by PoC.*

PROOF. Through the liveness property of $\mathcal{S}$ and Lemma 1, we conclude that honest miners eventually obtain a block committed by replicas of $\mathcal{S}$. Lemma 2 then ensures that miners search for a valid nonce η to settle this committed $\mathcal{S}$ -block as part of a block b . Finally, an honest miner can find a nonce in its slice $\mathbb{S}_i$ within time δ with non-zero probability. If it doesn't, Lemma 3 ensures that honest miners will try again until they succeed. As a result, honest miners eventually find a nonce η for block b in their slice within time δ . Lemma 4 then ensures that honest miners use η to settle block b and thus the committed $\mathcal{S}$ -block. $\square$

Fairness Argument. In Section 2.8, we argue that PoC is fair and no correct miners are penalized during periods of synchrony.

2.8. Theoretical Guarantees: Fairness Analysis

We argue that PoC is fair and no correct miners are penalized during periods of synchrony.

LEMMA 5 (Penalty Certificate). *There cannot be a penalty certificate $\langle \text{PENALTY}(b, \mathbf{r}, \mathcal{C}) \rangle_{\mathcal{M}}$ unless the timer δ of at least one honest miner expires.*

PROOF. We start by assuming that there exists a penalty certificate $\langle \text{PENALTY}(b, \mathbf{r}, \mathcal{C}) \rangle_{\mathcal{M}}$ and the timers of none of the honest miners have expired. As each penalty certificate needs signatures of at least $\mathbf{f}_{\mathcal{M}} + 1$ miners and at most $\mathbf{f}_{\mathcal{M}}$ miners are malicious, such an assumption is a contradiction. $\square$

THEOREM 3 (Fairness). *No honest miner receives a penalty if (i) it can find a nonce η within time δ and (ii) the network is experiencing a period of synchrony.*

PROOF. Let's assume an honest miner M_i receives a penalty based on shift round r . This implies the existence of a penalty certificate including miner M_i in its list of miners to penalize. Lemma 5 states that this certificate can only exist if the timer δ of at least *one* honest miner expires. Since at least $\mathbf{f}_{\mathcal{M}} + 1$ miners are honest, they will only penalize M_i if they did not receive its nonce before δ . This implies that either miner M_i did not find its nonce before δ (which is a direct contradiction of assumption (i)), or that its nonce did not reach the $\mathbf{f}_{\mathcal{M}} + 1$ honest miners before their timer expires (which is a direct contradiction of assumption (ii)). $\square$

2.9. Experimental Evaluation: PoC's Performance and Impact

Our evaluation aims to answer the following:

- (1) Performance of PoC vs. PoW. (Section 2.9.5)

- (2) Failure handling of PoC. (Section 2.9.6)
- (3) Impact of appending PoC to real-world systems. (Section 2.9.7)

2.9.1. Implementation. We implement the PoC protocol in the the Apache ResilientDB (Incubating), written in C++ [44]; RESILIENTDB is a blockchain framework that provides scalable APIs to implement and test new protocols. It also provides access to an optimized implementation of the PBFT consensus protocol. Our PoC implementation has an LOC count of 4,000, while RESILIENTDB has 50,000 LOC.

2.9.2. Evaluation Setup. We run experiments on AWS c5.9xlarge (36 vCPUs and 72 GiB memory) with up to 128 miners and clients. In all experiments, except Section 2.9.7, *unless explicitly stated*, we use the following setup: deploy 128 replicas in RESILIENTDB to run PBFT consensus on $\mathcal{S}$ -blocks of size 100. In each experiment, first 20% of the time we set as warmup and results are collected over the remaining time period. We average results over *five* runs to remove noise. We use *ED25519*-based DS for signing messages; PBFT makes use of *CMAC* for replica-to-replica communication. Clients issue requests in a closed-loop; a client waits to send its next transaction after it receives response for its previous transaction.

2.9.3. Benchmark. We run two types of experiments: (1) impact of appending PoC to PBFT. (2) impact of appending PoC to real-world blockchains. For the first experiment, clients create *YCSB* [62, 63] transactions from the Blockbench [63] framework and issue these transactions to the PBFT consensus. These single-operation transactions are key-value store operations that access a database of 600 k records.

For the second experiment, we extend the Diablo benchmarking framework [45], which issues client transactions to four state-of-the-art blockchain platforms.

2.9.4. Scalability of ResilientDB. First, we illustrate the scalability of RESILIENTDB fabric. This experiment will serve as a baseline for the future experiments, where we will illustrate the impact of PoC on the system throughput. In Figures 2.5(a) and (b), we measure the peak throughput (transactions per second) and latency for PBFT consensus protocol on varying the block size ($\mathcal{S}$ -block) from 1 to 1 k on a system of 128 replicas.

We observe that although PBFT hits its peak throughput at $\mathcal{S}$ -block size of 150, the optimal $\mathcal{S}$ -block size is 100 as the throughput is only 11.5% less than the peak, while the latency is $3\times$ lower.

Beyond an $\mathcal{S}$ -block size of 150, there is no increase in throughput because the queues that store messages at replicas are full and can no longer process newer requests, which increases the wait time (latency) for clients.

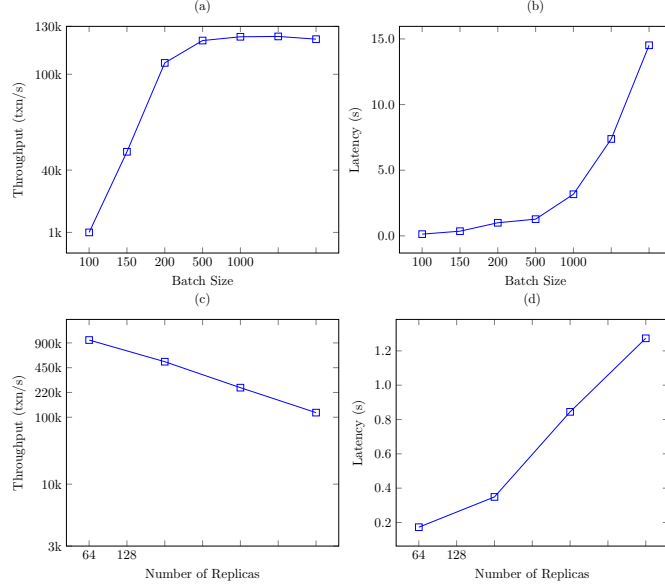


FIGURE 2.5. Evaluation of RESILIENTDB architecture.

Protocol	Mining Time (s)			Latency (s)		
	$D = 8$	$D = 9$	$D = 10$	$D = 8$	$D = 9$	$D = 10$
PoW (Sequential Mining)	696	> 2h	> 5h	1233	> 3h	> 10h
PoW	328	4108	> 5h	652	8200	> 10h
PoW + 30% PoC	7	165	3103	17	303	6950
PoC	3	48	738	6	84	1260

FIGURE 2.6. Time to find a nonce by a system of 128 miners while ensuring that these mining protocols meet RESILIENTDB PBFT’s throughput of 1k blocks per second.

$D = 8$	$D = 9$	$D = 10$
5 possible solutions on average	2 possible solutions on average	1 solution

FIGURE 2.7. The number of possible solutions on different difficulties.

Next, in Figures 2.5(c) and (d), we evaluate the scalability of PBFT on increasing number of replicas from 16 to 128, while setting the $\mathcal{S}$ -block size to 100. Unsurprisingly, on increasing the number of replicas, there is a drop in the peak throughput (consequential increase in latency) because there is a corresponding increase in the number of messages communicated per consensus; on moving from 16 to 128 replicas, the throughput drops by 86.8%.

2.9.5. Scalability of PoC. In this set of experiments, we measure the time it takes for various mining schemes to append a block to the ledger. In Figure 2.6, we compare PoC against three baselines: (1) Bitcoin’s PoW consensus, where each miner selects value uniformly at random and checks if it is a valid nonce. (2) PoW consensus where each miner sequentially iterates over each value in the search space (starting from 0)

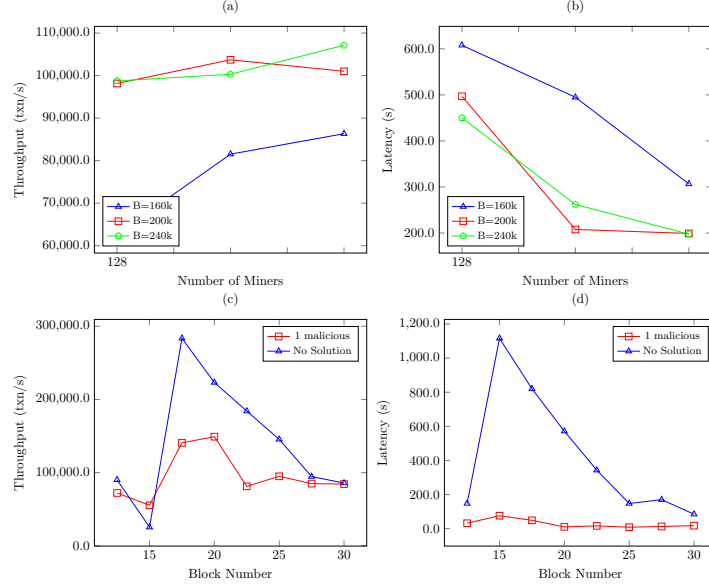


FIGURE 2.8. Peak throughput and average latency attained by PoC under different conditions at $D = 8$.

until it finds a valid nonce, (3) PoW with 30% miners running PoC, which allows us to approximate the impact of the largest centralized mining pool in Bitcoin [64].

In this experiment, we want to meet the following three goals: (1) The mining scheme reaches the same throughput (blocks added to the ledger per second) as RESILIENTDB’s throughput at 128 replicas, which is approximately 100k txns/s (or 1k blocks per second). (2) Aggregate precise number of $\mathcal{S}$ -blocks into a mined block so that we can observe least latency (difference between time an $\mathcal{S}$ -block is received to the time it is added to the ledger). (3) PoC can address multiple solutions. For example, for PoC, the time to mine a block at $D = 8$ is 3s, so we add 3k $\mathcal{S}$ -blocks in each mined block. The latency for each experiment is approximately twice the mining time because each transaction waits for a time equivalent to the mining time during the block generation phase and the mining process. We observe that sequential PoW mining requires at least $2\times$ more mining time and latency than randomized mining. In comparison, PoW with 30% mining pool does reduce the mining time and latency. However, at $D = 10$, PoC yields up to $4.2\times$ and $29\times$ less mining time than 30% PoC and PoW, respectively. In Figure 2.7, we also demonstrate that multiple solutions may be located in different slices when D is smaller. Like at $D = 8$, each block contains around 5 solutions. However, PoC ensures that each miner obtains the same solution to guarantee the safety of the block.

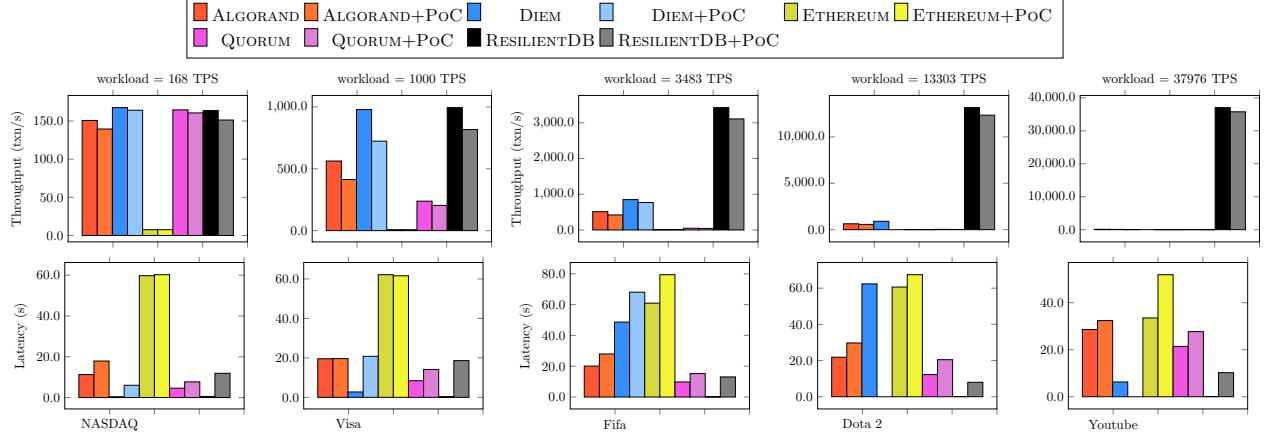


FIGURE 2.9. Impact of appending PoC (running at difficulty $D = 8$) to different Diablo blockchains.

Next, in Figures 2.8(a) and (b), we increase the number of miners from 64 to 128 while fixing the difficulty, $D = 8$. For each setting, there is a specific size of mined block at which it sustains the throughput of PBFT and achieves least latency. We observed 3k, 5k, and 7k to be such block sizes. We know that the peak throughput of PBFT is 100k txns/s and latency at $D = 8$ is around 6s. Thus, a PoC protocol with 64 miners hits the peak performance at 7k while a PoC protocol with 96 miners hits peak performance at 5k. Thus, we can conclude that PoC is non-invasive and has minimal impact on the system throughput.

2.9.6. Resilience to Failures. Next, we illustrate the effect of failures on PoC by studying two types of failures: 1 malicious miner and No nonce (Section 2.5.2) in Figures 2.8(c) and (d). In the malicious miner experiment, we simulate a Byzantine miner, which does not broadcast the solution of the 10-th block, which causes remaining miners to timeout and perform one round of the slice-shifting protocol. In the no nonce experiment, we ensure that no nonce satisfies the 10-th block, which leads to $\mathbf{f}_{\mathcal{M}} + 1$ rounds of slice shifting. These experiments cause the latency to shoot up for the 10-th block (up to $5\times$ for the no nonce case). To quickly bring latency and throughput to the steady state, we merge the blocks in the no nonce case.

2.9.7. Appending PoC to Blockchains in the Wild. Finally, we illustrate that PoC can be integrated to state-of-the-art blockchain systems to guard them against long-range attack with minimal impact. Specifically, we append PoC to *four* popular blockchains part of the Diablo [45] framework: DIEM [46], ALGORAND [28], QUORUM [47] and ETHEREUM [27]. Our primary goal is to showcase that appending PoC to these blockchains causes minimal impact on their performance.

Diablo benchmarking suite provides access to a framework, written in Golang, for evaluating popular blockchain systems. Diablo is composed of three types of nodes: *primary*, *secondary*, and *chain* nodes. The primary node is responsible for generating transactions and delivering them to the chain nodes via secondary nodes (mimicking a mempool functionality). The secondary nodes assist the primary node in reducing resource overhead, e.g., CPU and network bandwidth, by disseminating transactions to the chain nodes and collecting results. The chain nodes run various blockchains.

To incorporate RESILIENTDB into Diablo framework, we developed a Go SDK that provides an interface to send transactions from Diablo to RESILIENTDB. To append PoC to different chains, we implement a **Go-Server** agent on each chain node that periodically fetches blocks from the local chain on the chain node through their SDKs implemented using Golang. Specifically, the **Go-Server** provides a unified entry for PoC to obtain the block data. Note that in Diablo clients submit requests in an open loop.

Setup. Like Diablo authors, we use AWS c5.9xlarge (36 vCPUs, 72 GiB memory) machines. For each blockchain, we deploy one primary and 10 secondaries. We run each experiment on the workloads provided by Diablo. The maximum observed throughput (in transactions per second or TPS) of these blockchains as per the original paper is: NasDAQ (168 TPS), Visa (1000 TPS), Fifa (3483 TPS), Dota 2 (13303 TPS), and Youtube (37976 TPS).

Results. Figure 2.9 summarizes our findings. DIEM can sustain 1000 TPS and QUORUM only 204 TPS; their performance drops significantly when the workload increases. ETHEREUM attains a consistently low throughput in part due to its default block generation period (15 seconds). ALGORAND exhibits a more stable performance of 500-600 TPS across workloads. All blockchains suffer from higher latency (as expected) when the load increases, in a sense, artificially over-saturating the system. We observe that RESILIENTDB can achieve a high throughput of 37,967 TPS, which nearly matching the injected load. When PoC is added to any system, we observe that the throughput drops at most by 10%-20% due to an increased communication and added network latency. We argue that this is a negligible cost as PoC helps these blockchains prevent long-range attacks, which continues to be a major vulnerability in their design.

2.10. Related work

BFT has been studied extensively in the literature [29, 31, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76]. A sequence of efforts [11, 29, 48, 77, 78, 79, 80, 81, 82, 83, 84] have been made to reduce the communication cost of the BFT protocols: (1) linearizing BFT consensus [11, 60], (2) optimizing for geo-replication [85], and (3) sharding [86, 87, 88, 89, 90]. Nevertheless, all of these protocols face long-range attacks [91].

Alternatively, prior works have focussed on designing PoS protocols that permit the node with the highest stake to propose the next block [28, 49, 92, 93, 94]. However, even these protocols suffer from long-range attacks if adversary has access to the private keys.

Existing work to protect against long-range attacks includes: (1) checkpointing the state through a trusted committee [13, 95, 96, 97], (2) key-evolving cryptographic techniques [28, 94, 98], (3) verifiable delay functions [99], and (4) appending the state to Bitcoin [34, 52].

(1) *Trusted Committee* solutions aim to periodically checkpoint the state of a PoS blockchain on another canonical chain, which is maintained by a committee of trusted members [13, 95, 96, 97]. These systems assume that the trusted members cannot be compromised, and thus, new nodes that wish to join the system can distinguish between the PoS blockchain and the canonical chain. Moreover, small size committees mimic a centralized system, while large committees increase latency for checkpoints.

(2) *Key-evolving cryptographic techniques* force participants to periodically discard old keys and generate new keys [28, 94, 98]. These works assume that honest nodes will discard their old keys after they generate a new pair; the onus is on the honest nodes.

(3) *Verifiable delay functions* provide proof that helps differentiate between a ledger created long ago versus a recently created adversarial ledger [99]. However, nothing prevents an adversary from initiating the creation of the adversarial ledger at the time of genesis. Once it has access to the private keys of other nodes, it can build blocks on top of this ledger, which makes it impossible for a new node to distinguish between the two ledgers.

(4) *Appending the state to Bitcoin* is a popular solution against long-range attacks for many recent papers [34, 51, 52]. Bitcoin employs PoW consensus, which requires miners to compete and thus wastes computational resources. Prior solutions to improve PoW or Bitcoin [50, 100, 101] do not eliminate this competition.

With PoC, we show how to make it computationally expensive for an adversary to rewrite the ledger while forcing miners to collaborate and conserve their computational resources. Further, we show experimentally that collaboration helps PoC to waste fewer resources for a given difficulty.

2.11. Summary and Discussion

In this dissertation, we presented our novel PoC protocol, which, when appended to existing PoS/BFT protocols, guards them against long-range attacks. Like PoW, PoC makes it computationally expensive

for an adversary to rewrite the ledger. However, unlike POW, PoC introduces collaborative mining that requires miners to work with each other instead of competing.

Scaling Execution: Thunderbolt for Concurrent Smart Contracts in Sharded DAGs

3.1. Introduction: The Bottleneck of Smart Contract Execution

The emergence of blockchain technology has spurred significant interest in developing resilient systems capable of processing data and transactions under Byzantine conditions, including software errors, hardware failures, network disruptions, and coordinated malicious attacks [20, 89, 90, 102, 103, 104]. These systems enhance reliability and security by enabling collaboration among multiple independent participants [1, 12, 105, 106, 107, 108, 109, 110, 111].

Smart contracts [112, 113], as programmable transaction frameworks embedded in blockchain platforms, empower developers to address real-world challenges through decentralized solutions [114, 115]. However, their adoption is often hindered by execution delays caused by runtime contract code analysis [116]. To overcome this limitation, researchers are actively exploring performance optimization strategies for contract-based blockchain systems.

Several strategies have emerged to improve execution within blockchain systems.

Transaction Processing Models: One practical approach involves enhancing transaction processing. Most blockchain systems adopt the Order-Execute (*OE*) model, where transactions are ordered through consensus before execution [11, 29, 117, 118]. *OE*-based systems often employ deterministic concurrency controls by constructing transaction dependency graphs to optimize parallelism [119, 120, 121, 122]. However, platforms like Hyperledger Fabric [20] utilize the Execute-Order-Validate (*EOV*) model, executing transactions before consensus, to enhance flexibility and Optimistic Concurrency Control (*OCC*) [123] to improve the concurrency.

Scalability via DAG and Sharding: Another critical area for advancement in blockchain technology is scalability, particularly in supporting parallel execution. Recent advancements leverage Directed Acyclic Graph (DAG)-based consensus protocols to improve scalability. These protocols enable replicas to submit

proposals concurrently by building a DAG that links new proposals to historical ones. This architecture has gained significant recognition in the industry due to its robust security features, exceptional scalability, and capability to support smart contracts [124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134]. Complementary to DAG-based approaches, sharding techniques allow parallel transaction processing across each shard, reducing consensus overhead [22, 135, 136, 137, 138, 139, 140, 141, 142].

However, the above approaches do not effectively improve the execution of smart contracts:

Challenge1: Enhancing Transaction Parallelism without advanced knowledge. While *OE*-based solutions leverage dependency graphs to optimize parallelism, they require prior knowledge of transaction read/write sets, a constraint incompatible with dynamic smart contracts. Conversely, *EOV*-based approaches face high transaction conflict rates, necessitating advanced conflict resolution algorithms.

Challenge2: Efficient Cross-Shard Transaction Processing. Existing solutions for cross-shard atomicity, such as relay-based protocols (Sharper [143], BrokerChain [144], and SharDAG [145]) and traditional Two-Phase Commit (2PC) [138, 146], introduce significant delays due to inter-shard coordination. While multi-shard consensus [143, 147] mitigates 2PC limitations, it sacrifices scalability in large-scale networks with high contention.

The challenges described above raise the question of whether it is possible to design a sharding system that does not depend on understanding the read and write sets of transactions, nor requires additional coordinators to manage cross-shard transactions.

We propose Thunderbolt, an innovative sharding architecture that seamlessly processes both single-shard (Single-shard TXs) and cross-shard transactions (Cross-shard TXs) without centralized coordinators. Thunderbolt effectively integrates the *OE* and *EOV* models, where the *EOV* model enhances parallelism in the execution of Single-shard TXs, while the *OE* model minimizes the abort rate when handling Cross-shard TXs across different shards. Furthermore, the *OE* model ensures a coordinated execution order between these two types of transactions, maintaining the overall correctness of the execution process.

Similar to conventional sharding systems, Thunderbolt organizes transactions into distinct shards to mitigate potential conflicts. In particular, each replica within Thunderbolt corresponds to a single shard and acts as a shard proposer, proposing transactions within that shard. Thunderbolt employs a DAG-based consensus protocol [124, 125, 126, 127, 128, 129, 130, 131, 132, 133] to reach an agreement on the execution results provided by each shard proposer.

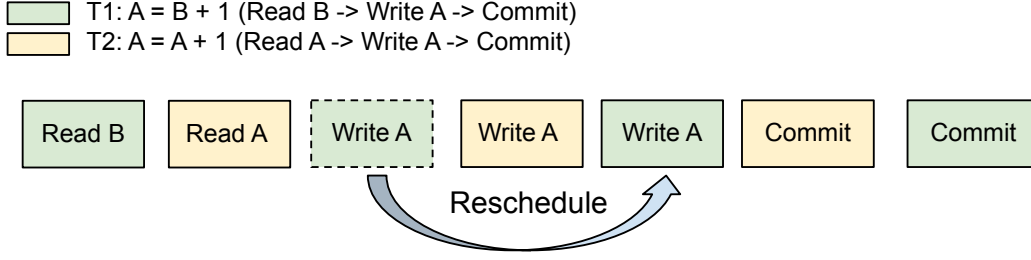


FIGURE 3.1. A transaction rescheduling to avoid abortion by moving the *WriteA* on T_1 after the *WriteA* on T_2 . They all obtain the correct result based on their operations.

Inspired by Sui’s epoch switching [148], Thunderbolt employs round-robin scheduling [149] to rotate shard proposers periodically, enhancing the system’s security and liveness. Proposer rotation is triggered on-demand if a shard fails to propose transactions within a timeout, with seamless DAG transitions preserving protocol continuity.

Thunderbolt also introduces a concurrent executor (*CE*) designed to improve the execution of Single-shard TXs before reaching consensus. Unlike traditional concurrency protocols that primarily manage conflicts based on the order of arrival [150, 151, 152], the *CE* utilizes a non-deterministic ordering system based on the execution run-time states of each transaction. This innovative approach minimizes the abort rate, thereby reducing transaction latency. As illustrated in Figure 3.1, the *CE* effectively reschedules transactions based on their run-time executions to prevent aborts. For instance, transaction T_2 , which would ordinarily conflict with the write operation of transaction T_1 can be successfully committed without cancellation.

In summary, this chapter makes the following contributions.

- To our knowledge, Thunderbolt is the first sharding consensus mechanism that combines the *OE* and *EOV* models based on DAG-based protocols without requiring any additional coordinators to determine the order between Single-shard TXs and Cross-shard TXs.
- We introduce a new concurrency paradigm that implements a parallel preplay for Single-shard TXs (concurrent consensus execution). Thunderbolt preplays Single-shard TXs followed by parallel verification without needing prior knowledge of the read/write sets.
- Thunderbolt features a non-blocking shard reconfiguration protocol that allows for the rotation of shard assignments without pausing either DAG dissemination or the consensus layer.
- We have implemented a concurrent executor to enhance the parallelism of executing smart contracts without prior knowledge of the read/write sets on Single-shard TXs. The execution engine

dynamically arranges transactions based on current assessments to reduce abort rates due to conflicts.

- Our evaluation of Thunderbolt demonstrates an impressive $50\times$ speedup over Tusk [124] with sequential execution using the SmallBank workload on 64 replicas built on Apache ResilientDB (Incubating) [44, 117].

3.2. Challenges: Parallelism Without Prior Knowledge and Efficient Cross-Shard Processing

3.2.1. Smart Contract. A smart contract is a digital protocol, introduced by Nick Szabo in the mid-1990s [113], designed to facilitate, verify, or enforce the negotiation or performance of a contract automatically. Unlike traditional contracts, which rely on legal systems for enforcement, smart contracts are self-executing and operate on blockchain technology. They are written in code and run on decentralized platforms like Ethereum, ensuring transparency, security, and immutability. Smart contracts eliminate the need for intermediaries, reduce the risk of fraud, and enable trustless transactions between parties. They are widely used in various applications, including decentralized finance (DeFi), supply chain management, and digital identity verification.

These contracts consist of custom functions that operate on user accounts with associated balances. Once deployed to the network, transactions invoking the functions specified in the contract are proposed to execute predefined operations that interact with the user accounts. However, the execution of the contract code occurs within the Ethereum Virtual Machine (EVM) [153], which results in the read and write sets of the contract being indeterminate prior to execution.

3.2.2. DAG-based protocols. Thunderbolt leverages a Directed Acyclic Graph (DAG) structure to address scalability challenges in blockchain systems. This architecture enables efficient transaction proposal mechanisms while maintaining robust security, high throughput, and native support for smart contracts. DAG-based protocols have gained significant traction in the industry due to their ability to decouple transaction dissemination from consensus processes [124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134].

In contrast to traditional linear blockchains, DAG-based protocols allow multiple replicas to propose transactions concurrently. These transactions are constructed into a deterministic DAG structure, ensuring a consistent topological ordering across all honest replicas. Recent advancements in this domain, including Tusk [124, 125], BBKA-Chain [133], Shoal/Shoal++ [130, 131], Mysticeti [154], and Cordial Miners [127], demonstrate how DAGs streamline consensus by separating data propagation from finality mechanisms.

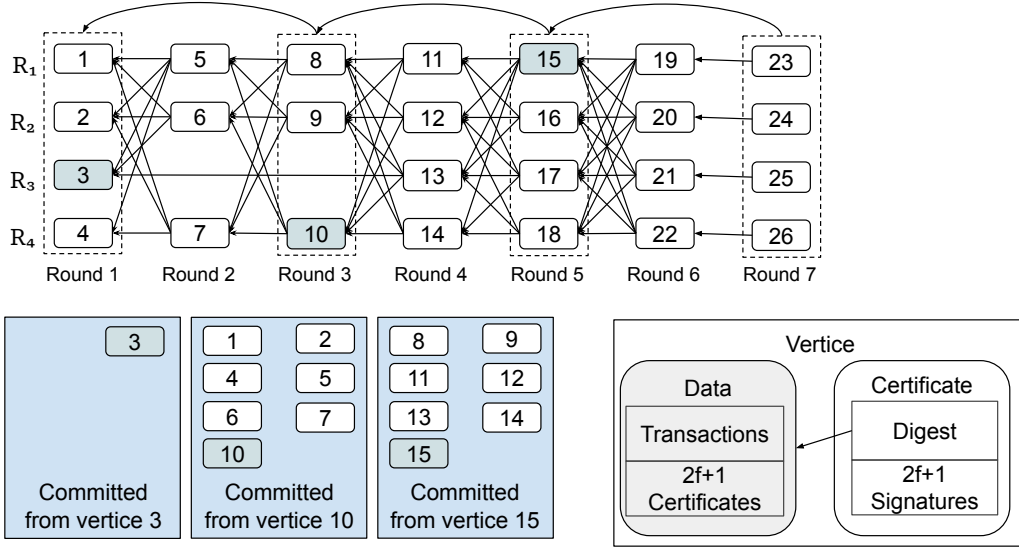


FIGURE 3.2. Overview of a DAG-based protocol on Tusk generated by 4 replicas. Each vertex contains both data and a certificate. Each data refers to $2f + 1$ certificates from the previous round. Each certificate is generated from its corresponding data, which has received signatures from $2f + 1$ replicas. The leader vertex for round r , the solid vertex used to commit the data in its causal history, will be determined before processing round $r + 2$.

The protocol operates in synchronized rounds, where each DAG vertex in a round consists of two components:

- **Data Payload:** Contains transactions and references to at least $2f + 1$ certificates from the prior round.
- **Certificate:** A quorum of $2f + 1$ cryptographic signatures attesting to the validity of the vertex and its dependencies.

During each round, replicas broadcast their proposed vertices to the network. A vertex becomes certified once $2f + 1$ signatures are collected, enabling it to serve as a dependency for new vertices in subsequent rounds. This iterative process ensures liveness while preserving the DAG's causal ordering.

Vertex finalization occurs at fixed intervals, typically every two rounds in Tusk [124] or three rounds in DAG-Rider [126]. A designated leader (selected via round-robin scheduling [149] or distributed randomness [?]) proposes a vertex for commitment. A leader vertex in round r is eligible to be committed during round $r + 2$ (as in Tusk): 1) The replica must have received at least $2f + 1$ vertices from round $r + 1$, and 2) The leader vertex must be referenced by a minimum of $f + 1$ vertices in round $r + 1$.

DAG-based protocols provide the following properties:

- Validity: if an honest replica R has a vertice B in its local view of the DAG, then R also has all the causal history of B .
- Consistency: if an honest replica R obtains a vertice B_r in round r from replica P , then, eventually, all other honest replicas will have B_r .
- Completeness: if two honest replicas have a vertice B_r in round r , the causal histories of B_r are identical.

3.3. System Overview: Thunderbolt’s Unified Sharding and DAG Architecture

Thunderbolt advances smart contract execution efficiency through an innovative sharding architecture augmented by a dynamic shard reconfiguration mechanism. This mechanism counters potential censorship attacks, such as post-execution transaction suppression or biased transaction selection, ensuring network integrity.

Unlike conventional sharding systems that partition replicas into isolated groups governed by separate consensus protocols, Thunderbolt employs a single unified consensus protocol jointly maintained by all replicas. Each replica operates as a shard proposer, processing transactions specific to its assigned shard. To coordinate cross-shard transaction ordering and execution, Thunderbolt leverages a DAG-based consensus protocol, enabling global agreement on transaction validity while preserving shard-level parallelism via the predetermined leaders underlying the consensus.

Thunderbolt employs both *EOV* model and *OE* model to address Single-shard TXs and Cross-shard TXs.

- Single-shard TXs (*EOV* Model): Transactions confined to a single shard are processed non-deterministically by a Concurrent Executor (*CE*) within the corresponding shard proposer. After the local preplay, results undergo a consensus across all shards to ensure state consistency.
- Cross-shard TXs (*OE* Model): Cross-shard TXs employ an optimistic concurrency control protocol with deterministic finalization. Atomic commitment is achieved post-consensus, allowing tentative execution optimistically while guaranteeing rollback-free confirmation.

Figure 3.3 demonstrates an example where two shard proposers propose two single-shard transactions. Further details on Single-shard TXs and Cross-shard TXs are provided in Section 3.4 and 3.5.

Thunderbolt also allows the migration of each shard to another replica to avoid a censorship attack, such as dropping transactions.

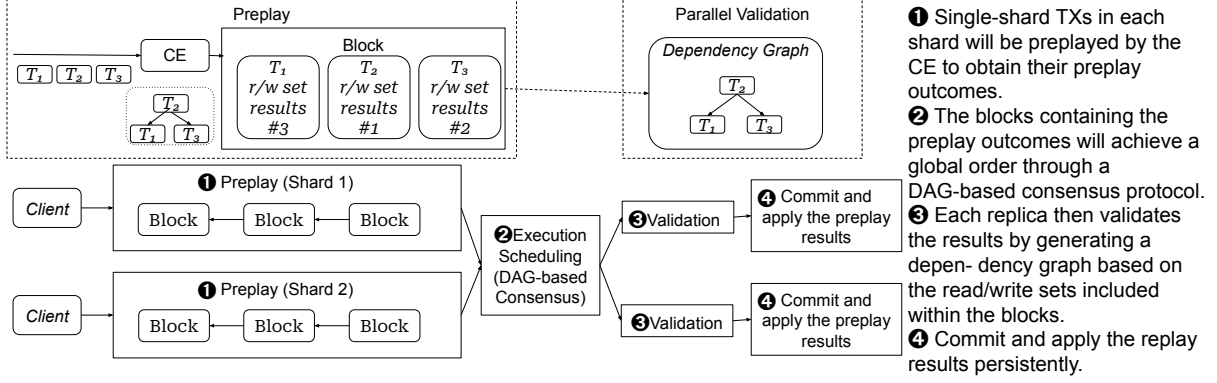


FIGURE 3.3. The dataflow of Single-shard TXs.

3.3.1. System, Threat and Data Model. In this section, we describe the system, threat and the data model. We leave the discussion of the Single-shard TXs, the Cross-shard TXs, and the shard reconfiguration in in Section 3.4, 3.5, and 3.6.

3.3.2. System Model. Thunderbolt is composed of n replicas, each of which serves a dual purpose: functioning as a shard and a replica. As a shard, a node maintains a distinct set or partition of data. In its role as a replica, it preserves a copy of the transaction log. Thus, each replica may also be designated as a shard proposer. Furthermore, clients direct transactions to the appropriate shards, and every replica engages in the consensus process to establish a cohesive order for these transactions.

In summary, a node in Thunderbolt assumes three essential roles: 1). It operates as a shard proposer, managing Single-shard TXs within each shard in an independent manner. 2). It acts as a replica that contributes to the consensus process. 3). It serves as a leader that commits Cross-shard TXs in a total order in accordance with the consensus protocol. For clarity in the accompanying illustrations, the terms "replicas" and "shard proposers" may be used interchangeably. We will utilize the term "shard proposers" when delineating shard procedures and will revert to "replicas" in discussions regarding the consensus protocols.

3.3.3. Threat model. We consider a set of n replicas (or shards), where at most f of these replicas can be faulty and $n = 3f + 1$. The f faulty replicas may exhibit any arbitrary behavior, including Byzantine failures, while the remaining replicas are assumed to be honest and will adhere to the protocol's specifications at all times. Additionally, we assume that clients are not trustworthy and would not expect to send transactions to all of the shards associated with their transactions. The network is expected to be eventually synchronous [155], meaning that messages sent from a replica will eventually arrive within a global stabilization time (GST), which remains unknown to the replicas. Communications between replicas use

authenticated point-to-point channels, with messages signed by the sender using a public-private key pair for authentication.

3.3.4. Data model. The data model assumes that each transaction includes a contract code with functions to access data belonging to the sender in the shard. The contract involves two types of operations: $\langle Read, K \rangle$ and $\langle Write, K, V \rangle$. Here, K represents the key required for access and V is the value that needs to be written to the key K . The contract code is Turing-complete and users could not obtain any information without execution. We also assume that the functions of the contract are idempotent.

Our system is designed with the understanding that data must be partitioned and that each key is assigned a shard ID (SID) before it can be utilized. These $SIDs$ are predefined and recognized across all shards. They fulfill a dual function: they guide transactions to the appropriate shard proposer and support parallel processing among multiple shards, thereby enhancing overall system efficiency (Section 3.5). The actual method for partitioning is orthogonal to our work and any existing techniques can be utilized [156, 157, 158, 159, 160].

3.4. Core Mechanism 1: The Concurrent Executor (CE) for Single-Shard Transactions

Thunderbolt processes Single-shard TXs through three core components: preplay, execution scheduling, and validation. During each round, a shard proposer initiates the workflow by preplaying a batch of Single-shard TXs. This generates a block containing critical preplay outcomes, which the proposer propagates to other shards via a DAG-based consensus protocol. During consensus, these blocks undergo parallel validation across shards. Once a replica commits a block, it applies the preplay results to its storage.

3.4.1. Preplay. In Thunderbolt, shard proposers play a pivotal role by preplaying transactions to determine their outcomes before block creation. A concurrent executor (CE) is employed to preprocess batches of transactions efficiently, producing detailed outputs for each transaction. These outputs include: the read/write sets accessed during execution, the corresponding execution results, and a scheduled execution order (as depicted in Figure 3.3).

The scheduled order establishes a deterministic serialized sequence, ensuring transaction results remain consistent when executed in the prescribed order. The read/write sets reveal the specific data accessed by each transaction. Crucially, these sets cannot be predetermined and are derived exclusively via the preplay process.

3.4.2. Execution Scheduling. Thunderbolt integrates with any DAG-based dissemination layer that employs a consensus protocol to establish a total block order across replicas. In each round r , a shard proposer R proposes a CE -generated block to the DAG, creating a new vertex in the graph. This vertex links to all prior vertices, including those proposed by R in round $r - 1$. For clarity, "vertices" in the DAG are hereafter referred to as "blocks" in the following sections, implying they have been certified by the protocol.

3.4.3. Validation. When a replica receives data for round r via the DAG (Section 3.2.2), Thunderbolt initiates a rigorous validation process to verify the integrity of preplay results within the blocks. Validators construct a local dependency graph using the read/write sets to enable parallel transaction validation rather than sequential checks, optimizing system throughput. Notably, blocks from round $r - 1$ are validated before those from round r for the same shard proposer.

During re-execution, validators confirm that the computed read sets match the values recorded in the block. A valid dependency graph guarantees consistent read-set results and ensures the final state of each key aligns with the block's declared values. If discrepancies in read-set values are detected, the block is flagged as invalid and discarded. Until blocks are committed, replicas retain preplay results in local storage, either to process Cross-shard TXs (Section 3.5) or until DAG reconfiguration occurs (Section 3.6).

3.5. Core Mechanism 2: Coordinating Single-Shard and Cross-Shard Transactions

Cross-shard TXs involve multiple shards and require consensus to establish a total execution order. This total order ensures that all Cross-shard TXs are executed consistently across the involved shards. However, each shard proposer preplays Single-shard TXs independently and replicates the results; in contrast, the total order for the Cross-shard TXs must be determined first before they can be executed (for instance, preplay optimization cannot be employed for Cross-shard TXs). Therefore, Thunderbolt must coordinate the sequencing between the Cross-shard TXs and the Single-shard TXs to guarantee consistent execution outcomes across all replicas. Prior approaches often rely on coordinators to establish inter-shard order [137, 138, 139, 143, 144, 145, 146, 147], but these introduce communication overhead that degrades performance.

3.5.1. Rules for Proposals. To address issues, Thunderbolt leverages the DAG's predetermined leaders to enforce a consistent partial order between single-shard and cross-shard transactions via the following rules:

- G1) If a leader L commits both a Single-shard TX and a Cross-shard TX, the Single-shard TX must be committed first.

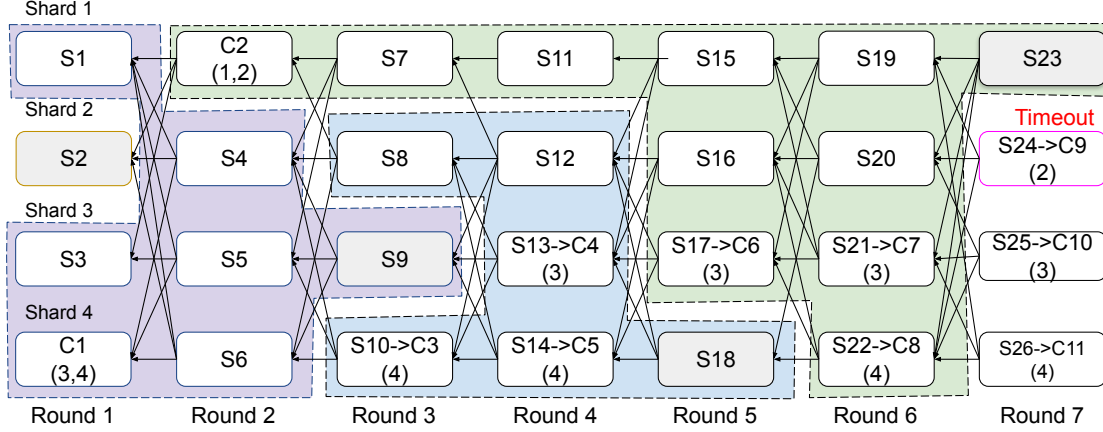


FIGURE 3.4. An example illustrates the commitment of each leader in the process of converting Single-shard TXs to Cross-shard TXs on Tusk. In this DAG, S_i denotes a Single-shard TX, while $C_i(\{X\})$ represents a Cross-shard TX associated with shards $\{X\}$. The leaders in the odd rounds are selected using round-robin selection. The blocks committed by the same leader have borders of the same color. S_i will be converted to C_i if there is any conflict blocks (S_{10}) or could not receive the leader block in time (S_{24}).

G2) If leader L_i commits Cross-shard TX X in round i , any Single-shard TX Y committed by leader L_j in round j (where $j > i$) cannot execute until X is finalized.

It is worth knowing that Leaders are predetermined per round using deterministic methods (e.g., round-robin or global random coins). To enforce these rules, Thunderbolt applies the following proposal rules:

- P1) Cross-shard TXs are submitted directly to the DAG, bypassing the CE .
- P2) Leaders committing a batch of transactions must finalize all Single-shard TXs before Cross-shard TXs.
- P3) If a shard proposer SL proposes a Single-shard TX X in round r , and the current round's leader L differs from SL , SL must:
 - Wait for L 's proposal before preplaying X .
 - Convert X to a Cross-shard TX if any uncommitted Cross-shard TX Y in L 's history conflicts with X .
 - Otherwise, preplay X and submit the results.
- P4) If a shard proposer SL proposes a Single-shard TX X in round r , and a prior leader's uncommitted Cross-shard TX Y (in round $q < r$) conflicts with X , SL converts X to a Cross-shard TX.
- P5) If leader L in round r commits a Cross-shard TX X related to shard A but lacks A's proposal in round $r - 1$, L defers committing A and A's subsequent proposals.

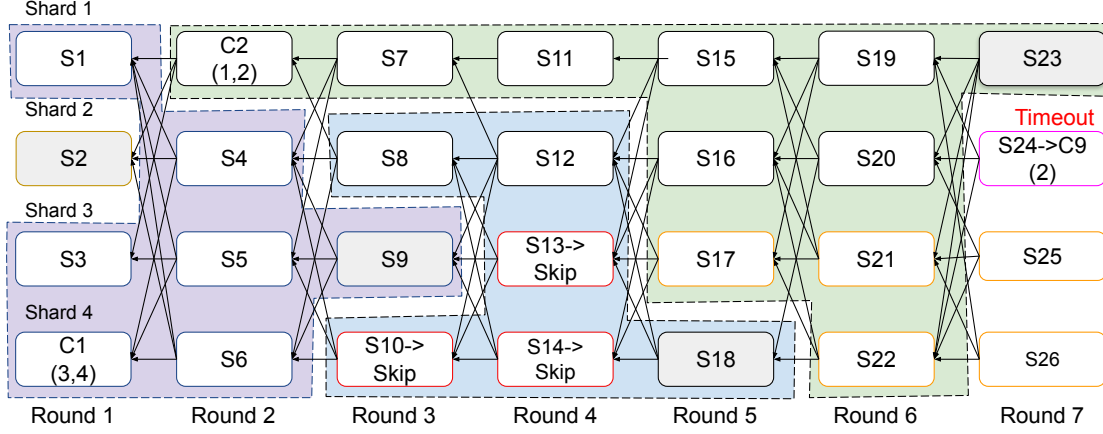


FIGURE 3.5. An example of proposing skipping blocks to restart the preplay of the Single-shard TXs. The Single-shard TXs (S17, S21, and S22), which would be converted to Cross-shard TXs after round 5 in Figure 3.4, can replay their executions before delivering to the DAG.

P6) If a shard proposer SL proposes a Single-shard TX X in round r but the leader L 's proposal is delayed beyond a timeout, SL converts X to a Cross-shard TX.

These rules enable Thunderbolt to process Cross-shard TXs without blocking shards while maximizing parallelism for single-shard transaction preplay.

EXAMPLE 5. Figure 3.4 illustrates Thunderbolt's handling of single-shard and cross-shard transactions on Tusk: Single-shard TXs will be executed before Cross-shard TXs under the same leader (Rule P2). S10 is converted to Cross-shard TX C1 due to its dependency on S9's leader history (Rule P3). S13 and S14 become C4 and C5, respectively, because C1 remains uncommitted until round 5 (Rule P4). Leader S18 in round 5 skips C2 and subsequent transactions due to missing S11 (Rule P5). In round 7, S24 converts S4 to Cross-shard TX C9 after failing to receive leader S23's proposal (Rule P6).

3.5.2. Parallel Execution. When processing Cross-shard TXs, Thunderbolt preserves all sharding metadata for each transaction. Rather than sequential execution, Thunderbolt employs deterministic concurrency control mechanisms, such as QueCC [119], to construct dependency graphs using cross-shard metadata (SID). This enables parallel execution while maintaining consistency across shards.

3.5.3. Message Failures. In practical deployments, network unreliability may delay message delivery. To mitigate this:

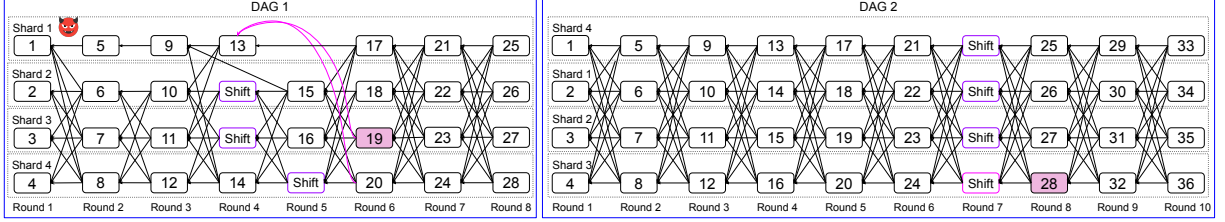


FIGURE 3.6. An example of DAG reconfiguration. Shard 1 in DAG 1 which is a malicious replica that delays the blocks from round 2, which triggers a reconfiguration to a new DAG ($K = 2$). Additionally, a further reconfiguration to DAG 2 will take place after the replicas have proposed six rounds ($K' = 6$).

- If a leader L cannot include all Single-shard TXs linked to a Cross-shard TXs due to network delays (e.g., missing shard proposals), L bypasses the affected Cross-shard TX and subsequent transactions from the same shard ($C'2$ in Figure 3.4 on Rule P5). This prevents violations of global ordering G2 by excluding incomplete transaction sets. These transactions are later finalized by subsequent leaders.
- If a shard proposer fails to receive the leader's proposal within a round's timeout window, it cannot preplay its Single-shard TX (Rule P6). The proposer instead promotes the transaction to a Cross-shard TX and submits it directly to the DAG (such as $S24$ in Figure 3.4).

3.5.4. Preplay Recovering. Under Rule P3, a shard proposer SL must convert a Single-shard TX to a Cross-shard TX if it detects conflicting uncommitted Cross-shard TXs in prior leader histories. While this ensures safety, it forfeits the performance benefits of preplay, such as the blocks in shard 3 after round 3 in Figure 3.4.

To recover preplay, SL must verify that all conflicting Cross-shard TXs have been finalized by preceding leaders. Since a leader CL_r in round r is finalized within two subsequent rounds (Section 3.2.2), SL can safely preplay new single-shard transactions once: 1) It receives $2f + 1$ certificates in round $r + 1$, and 2) CL_r is referenced by at least $f + 1$ blocks in round $r + 1$. If SL identifies any conflicting Cross-shard TXs while proposing Single-shard TXs, SL submits skip blocks to the DAG until prior leaders are finalized. For instance, $S10$, $S13$, and $S14$ in Figure 3.5 are converted to skip blocks. Consequently, subsequent transactions, like $S17$ and $S22$, are reverted to the *EOV* model, restoring preplay capabilities.

3.6. Enhancing Robustness: Non-Blocking Shard Reconfiguration

In Byzantine environments, compromised replicas may enable attackers to launch censorship attacks, threatening the integrity of transactions within their assigned shards.

Thunderbolt employs a round-robin selection mechanism [161] to rotate shard proposers if a leader fails to propose transactions for K rounds or at fixed intervals of K' rounds (where $K' > K$).

This rotation serves dual purposes:

- Preventing transaction duplication (DDOS [162, 163]): Proposers perform local deduplication to block malicious clients from flooding the system with redundant transactions, a known challenge in DAG-based protocols [125, 129, 148].
- Mitigating censorship: Regular rotation limits the window for a compromised proposer to disrupt operations.

Unlike traditional consensus protocols that depend on notification messages to alter primary replicas, Thunderbolt introduces an innovative mechanism that uses the underlying DAG protocols to facilitate the seamless transition to a new DAG and reconfigure shard proposers. We leverage a round-robin approach to select a new proposer that if the current proposer of shard X is replica R_i , the subsequent proposer of shard X will be $R_{(i \bmod n)+1}$.

However, the transmission of blocks to a new proposer may experience delays or omissions due to network issues or the actions of a malicious proposer. If the new proposer for round r is unable to receive the proposal committed in round $r - 1$ from the previous proposer, the new proposer will stop operations until the block arrives to ensure safety.

To address this challenge, Thunderbolt introduces Shift blocks to reach agreements among shards when a shard reconfiguration should be initiated and switch to a new DAG to process further transactions.

A replica R in a shard broadcasts a Shift block in round r under the following conditions:

- (1) R receives no block from a shard proposer after round $r - K$.
- (2) R has proposed blocks for at least K' rounds.
- (3) R received $f + 1$ Shift blocks from distinct replicas at round $r - 1$.
- (4) R does not broadcast the Shift block before.

EXAMPLE 6. *In Figure 3.6 where $K = 2$ and $K' = 6$, shards 2 and 3 broadcast Shift blocks in round 4 after failing to receive blocks from shard 1 in rounds 2–3. Despite receiving a block in round 4, shard 4 broadcasts a Shift block in round 5 upon receiving two Shift blocks from peers, prioritizing liveness assurance.*

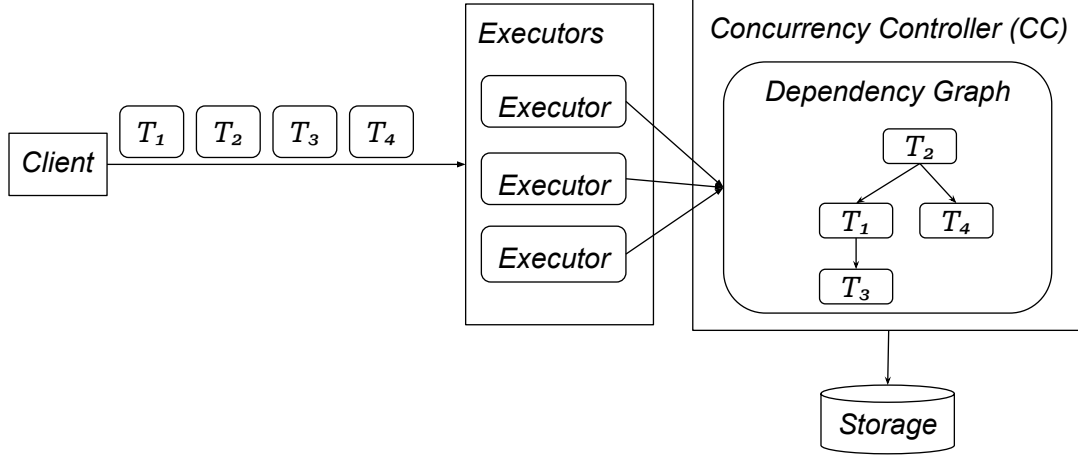


FIGURE 3.7. The architecture of the *CE* consists of a set of executors that execute transactions. The concurrency controller utilizes a dependency graph to determine the order of transactions and their execution results.

3.6.1. Non-blocking Reconfiguration. Since at $2f + 1$ honest replicas will commit the same block during the same round (Section 3.2.2), we designate the round of the first commit block that includes $2f + 1$ Shift blocks as the ending round for the current DAG. Then, **each replica will begin a new DAG from the same ending round to ensure the system's safety.** For instance, shard 2 will propose block 15 at round 5 after proposing a Shift block at round 4. Finally, block 19 from shard 3 at round 6 is selected as the leader during the consensus process. Subsequently, shard 3 commits all historical blocks from block 19, including the Shift blocks from other shards. At this point, all shards will transition to the new DAG (DAG 2) and start executing transactions within the new shard. Moreover, each replica will propose a Shift block every $K' = 6$ blocks in the new DAG (DAG 2) to facilitate a transition to the next DAG. This measure is intended to protect against censorship attacks, which may involve dropping transactions, failing to propose blocks, or prioritizing certain transactions over others. Additionally, the non-blocking mechanism provides protection against malicious proposers as the reconfiguration process requires a minimum of $2f + 1$ Shift blocks to be effective.

3.6.2. Uncommitted Transactions. Due to the two-round leader commitment latency, transactions uncommitted in the ending round of a DAG are discarded and must be resubmitted. Only transactions from the last two rounds or those excluded by the leader are affected. Clients will automatically retransmit transactions lost due to the reconfiguration.

Time	Transactions	Operations	Dependencies	Execution Order
0	Initial DB	$D = 3$	$\{\}$	$\{\}$
1	$T_1:(W, D, 3)$	T_1 writes $D = 3$	$\{T_1\}$	$\{\}$
2	$T_2:(R, D, 3)$	T_2 reads D on T_1 : ($D = 3$)	$\{T_1 \rightarrow T_2\}$	$\{\}$
3	$T_3:(R, D, 3)$	T_3 reads D on T_1 : ($D = 3$)	$\{T_1 \rightarrow T_2\}$ $\{T_1 \rightarrow T_3\}$	$\{\}$
4	T_3 : Commit	Wait for T_1	$\{T_1 \rightarrow T_2\}$ $\{T_1 \rightarrow T_3\}$	$\{\}$
5	$T_1:(W, D, 5)$	T_1 writes $D = 5$. Abort T_2, T_3	$\{T_1\}$	$\{\}$
6	$T_3:(R, D, 5)$ (Re-execute)	T_3 reads D on T_1 : ($D = 5$)	$\{T_1 \rightarrow T_3\}$	$\{\}$
7	T_1 : Commit	Commit T_1	$\{T_1 \rightarrow T_3\}$	$\{T_1\}$
8	T_3 : Commit	Commit T_3	$\{T_1 \rightarrow T_3\}$	$\{T_1, T_3\}$
9	T_2 : (W, D, 3)	Invalid and re-execute		
10	$T_2:(R, D, 5)$ (Re-execute)	T_2 reads D on T_1 : ($D = 5$)	$\{T_2\}$	$\{T_1, T_3\}$
11	T_2 : (W, D, 2)	T_2 writes $D = 2$	$\{T_1 \rightarrow T_2\}$ $\{T_1 \rightarrow T_3\}$	$\{T_1, T_3\}$
12	T_2 : Commit	Commit T_2	$\{T_1 \rightarrow T_2\}$ $\{T_1 \rightarrow T_3\}$	$\{T_1, T_3, T_2\}$

TABLE 3.1. An example of generating the dependency while executing the transactions $\{T_1, T_2, T_3\}$ that access the data D and determining the execution order.

3.6.3. Censorship Attacks. Malicious replicas may attempt censorship via message drops, transaction rescheduling, DDoS attacks, or proposal halting [162, 163]. As each replica governs an entire shard in Thunderbolt, such attacks can paralyze specific shards. The reconfiguration mechanism counters this by periodically reassigning shards to new replicas, limiting the impact window of compromised replicas.

3.7. Correctness Analysis: Safety and Liveness in Thunderbolt

The concurrent executor (CE) is a critical component enabling Thunderbolt to process Single-shard TXs concurrently during the preplay phase. As a nondeterministic concurrency control executor, CE generates a serialized execution order, read/write sets, and execution results for transaction batches. These outputs allow any replica to independently verify correctness, even though the CE -derived order may differ from the transactions' arrival sequence.

The architecture of CE is illustrated in Figure 3.7, where a set of executors executes the transactions, and a concurrency controller (CC) determines the execution results among the transactions.

Transactions progress through a two-phase data flow: an execution phase (operations processing) and a finalization phase (commit/abort decisions). Table 3.1 exemplifies this workflow using transactions $\{T_1, T_2, T_3\}$.

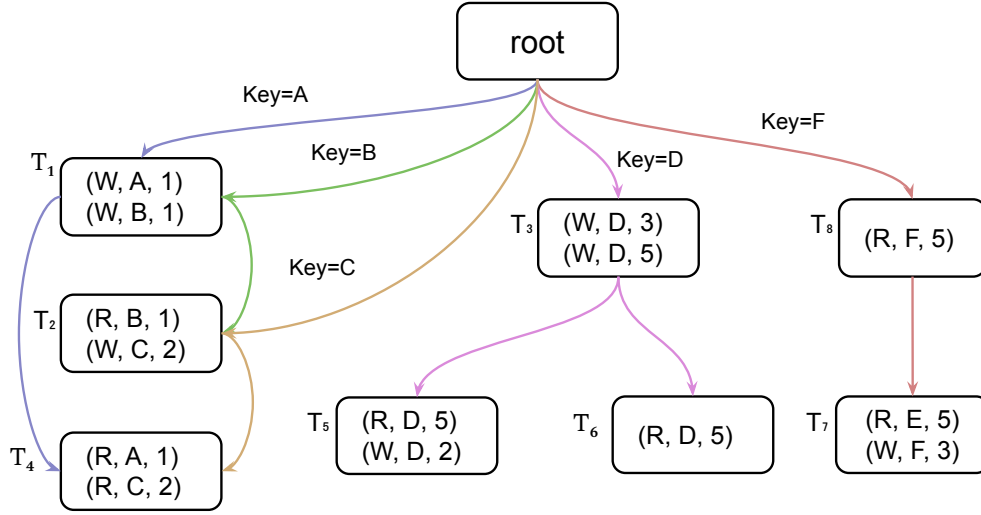


FIGURE 3.8. Dependency Graph on *Thunderbolt*. Edges with the same color represent a dependency graph with a specific key.

3.7.1. Execution Phase. During the execution phase, the executors access the data within *CC* directly and *CC* maintains a dependency graph to keep track of the relationship between transactions and all the results are stored in the graph directly to avoid accessing the disk IO. The critical characteristic of *CC* is that *CC* only maintains the graph based on the current operations among the transactions without requiring any read/write set knowledge. Furthermore, *CC* mechanism is nondeterministic, which means it can arrange transactions in any order based on their current states. The order of two transactions will not be established until a dependency is created. For instance, in the case of two conflicting transactions, denoted as T_1 and T_2 , which both only write the same key, either transaction may be ordered first. Both execution orders, $[T_1, T_2]$ and $[T_2, T_1]$, are considered valid. A dependency is established based on the commit times of these transactions. If T_1 commits before T_2 , the execution order $[T_1, T_2]$ indicates that T_1 is prioritized over T_2 . Additionally, a third transaction, T_3 , can influence this dependency. A dependency is formed if T_3 reads the value following T_2 's write before T_1 and T_2 commit, resulting in a unique execution order of $[T_1, T_2, T_3]$. As a result, a dependency indicator will be generated for the two conflicting transactions once a read-write conflict arises or when both transactions have been committed.

While receiving an operation from a transaction T sent by the executors, identified by accessing the key K , denoted as O_k , *CC* checks the relationships among the transactions. If T conflicts with other transactions or has been aborted by other transactions, the operation O_k will not be considered valid. For instance, T_2 at

time 9 in Table 3.1 is an example of an invalid operation, as it was aborted by T_1 at time 5 due to its outdated read in D . Transaction T will be aborted in such cases and require reexecution. Otherwise, if the operation O_k is valid, it will be added to the dependency graph (section 3.8.1) and obtain the operation result, such as the value V that O_k intends to read. We can obtain the operation results from other transactions directly based on the dependency graph to allow reading uncommitted data, such as T_2 reads D on T_1 at time 2.

3.7.2. Finalization Phase. During the finalization phase, the executor informs CC that the executor has completed all the operations. Then CC will update the results to the storage asynchronously once all its dependencies have been committed and assign the execution order to the transactions. If CC has terminated the transaction due to conflicts with other transactions, CC aborts the transaction and notifies the executor to restart the execution.

3.8. Dependency Graph in Concurrency Controller

This section explains the dependency graph G , which is central to the CC component. It plays a crucial role in maintaining the causal relationships between transactions during the preplay process in CE . The CC component ensures that the sequential order of execution defined by G is valid.

3.8.1. Dependency Graph Construction. A Dependency Graph is a graph $G(V, E)$ that plays a crucial role in tracking the causal relationship between transactions in CC . Each node $v \in V$ represents a specific transaction. Additionally, each edge $e(u, v, k) \in E$ indicates a connection between two transactions u and v on a key K . This relationship is represented as $u \rightarrow_k v$. For example, in Figure 3.8, transaction T_5 generates an edge $e(T_3, T_5, D)$ from T_3 because T_5 acquires the value 3 of key D from T_3 . Without loss of generality, we have assigned a root node denoted R and added edges $e(R, u, k) \in E$ for each $u \in V$ that accesses the key K but does not have any incoming edge on key K , such as T_7 and T_8 in Figure 3.8.

If the graph G is acyclic, a sequential order can be established by generating a topological order. It is crucial that every transaction must obtain the same causal order in any topological order from G to ensure consistency. Therefore, G is considered a valid graph only if any sequential order generated from the topological order is a valid serialization order and produces the same results. By following any correct order, all transactions will yield the same execution results. However, because of the nondeterministic characteristic, the results may not be the same as those executed in their arrival order. Each node u maintains all records of the operations triggered by a transaction u , including the resulting values. The sequence of the linking nodes establishes the order of commitment between the two transactions.

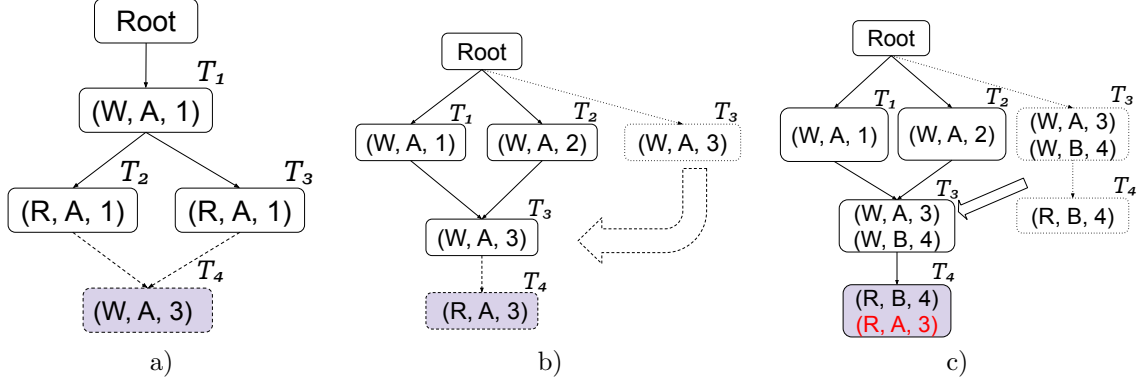


FIGURE 3.9. An example of incorporating operations into the dependency graph and adjusting the graph to ensure correctness. a) T_4 writes $A=3$. b) T_4 reads A . c) T_4 reads key A on its existing node and retrieves the value from T_3 .

Since a transaction is an atomic commitment, we combine the internal operations to simplify the in-node states. However, to trace the conflicts between two nodes, we must retain the first operation if it is a read and the last operation if it is a write, to ensure that the causal relationship is not lost. Thus, we remain at most two operations in the nodes: the first read and the last write.

To help illustrate the algorithm, we define the types of each node depending on the operations it contains on a key:

- A node $v \in V$ is a read node R_v^k if the first operation on key K is a read.
- A node $v \in V$ is a write node W_v^k if v contains write operations on key K .
- The root node R is a write node.

3.8.2. Generating New Nodes. This section presents the process of adding operations from a new transaction to the dependency graph G .

CC creates a new node whenever an operation O_k is received from a new transaction T . If O_k is a write operation, T needs to establish a connection to each casual relation. To avoid pointing to the root and assuming that the earlier transaction will commit first, the non-write nodes v on key K , which only contains reads, without any outgoing edges (not dependent by other nodes) are selected, and edges $e(v, u, k)$ are added, pointing to u (Figure 3.9 (a)).

On the other hand, when a read operation is performed, CC selects the latest write node u to obtain the latest value or selects the root to read the data value from storage if no write nodes exist. If the write node u is selected, we need to make all other write nodes W_v^k contain a path to u to guarantee the correctness for the read after write between u and v . Finally, the operation and its result $\langle \text{Type, Key, Result} \rangle$ will be

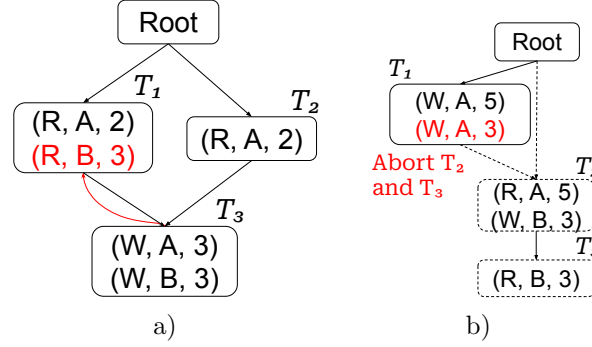


FIGURE 3.10. Cycle of conflicts and cascading aborts. a) T_1 reads B and adds a dependency from T_3 following the rules in Section 3.8.2. b) Cascading aborts from T_1 since T_1 wants to write A that breaks the read on T_2 .

written into the node u . An example of adding a new read operation on A from T_4 is depicted in Figure 3.9 (b). T_4 selects T_3 to read to obtain $A = 3$ and adds an edge from T_3 and a record $\langle R, A, 3 \rangle$ is logged down in the node. Then T_3 will add two edges from T_1 and T_2 , respectively.

3.8.3. Operations on Existing Nodes. When receiving an operation O_k for the key K from an existing transaction T in G , CC will select the corresponding node u to attach the record. If O_k is a read operation, the result will be directly retrieved if u contains the record for key K . Otherwise, it will proceed with the new node operation as specified in Section 3.8.2 to choose a previous one to access the value. Figure 3.9 (c) illustrates an instance where T_4 reads key A as its second operation and retrieves the value from T_3 . If O_k is a write operation, the operation will be appended to the node.

3.8.4. Conflict Detection. Appending the records to an existing node may lead to transaction conflicts. For instance, a transaction updates the value again but it has been read by another transaction or a dependency cycle is created due to the dependency on another key since we always find the latest write to retrieve the value. Figure 3.10 (a) depicts a scenario in which T_1 attempts to retrieve the value of B from T_3 , which has established a dependency from T_1 due to key A , which results in the creation of a dependency cycle. In this case, CC will try to read the value from its ancestor, like B reads the value from the *Root* Figure 3.10 (a). If there is still any conflict with other transactions, CC will trigger the abort process.

Once conflicts are detected, CC triggers an abort process as follows:

- (1) If u only contains read operations, abort T itself.
- (2) If u contains write operations, cascading abort from T .

In Figure 3.10 (b), we need to abort T_2 and T_3 since T_1 contains a write operation. However, in Figure 3.10 (a), we only need to remove T_1 and keep T_3 alive.

3.9. Correctness Analysis: Safety and Liveness in Thunderbolt

In this section, we conduct an analysis of the safety and liveness properties of Thunderbolt. Safety is defined as if two conflicting transactions, T_1 and T_2 , are executed within an honest replica in a specific order O (for instance, $T_1 < T_2$), it is expected that all other honest replicas will also execute T_1 and T_2 in the same order O . Liveness, on the other hand, is characterized by the assurance that client requests will consistently receive a response.

3.9.1. Proof of Safety. We prove the safety of Thunderbolt. We will first analyze the safety **within the same DAG**.

THEOREM 4. *In the case of two conflicting transactions, T_1 and T_2 , which may occur as either Single-shard TXs within the same shard or as Cross-shard TXs, if T_1 is executed prior to T_2 in an honest replica, any other honest replica will also execute T_1 before T_2 .*

PROOF. The DAG protocol ensures that the sequence of two blocks proposed by the same proposer aligns with their commit order. Consequently, the order of two Single-shard TXs proposed by the same proposer will be preserved. Furthermore, the order of Cross-shard TXs is established by the consensus, resulting in a consistent global order. This leads to the uniform execution of transactions T_1 and T_2 across all honest replicas, irrespective of whether the transactions are Single-shard TXs or Cross-shard TXs. $\square$

THEOREM 5. *In the scenario where two conflicting transactions occur, where T_1 is a Single-shard TX and T_2 as a Cross-shard TX, if an honest replica executing T_1 prior to T_2 , any other honest replicas will execute T_1 before T_2 as well.*

PROOF. Consider a contradiction scenario in which replica R_1 executes T_1 prior to T_2 , while replica R_2 executes T_2 before T_1 . We also suppose leader X commits T_1 and leader Y commits T_2 . Then it can be shown that R_1 is the proposer of T_1 and T_1 is not converted into a cross-shard transaction. Since R_2 executes T_2 ahead of T_1 , the round of leader Y must be less or equal to the round of leader X ($Y \leq X$). However, if the round r to T_1 occurs before Y ($r < Y$), then T_2 cannot be committed (Rule P5). Conversely, if $r \geq Y$, then according to Rule P4, T_1 should be converted into a Cross-shard TX if T_2 remains uncommitted. Thus, it is impossible for R_1 to execute T_1 before T_2 while R_2 executes T_2 before T_1 . $\square$

THEOREM 6. *When two conflicting transactions, T_1 and T_2 , are executed by honest replicas in the order of $T_1 < T_2$ across two different DAGs, it is ensured that all other honest replicas will execute T_1 and T_2 in the same order within a single DAG.*

PROOF. Section 3.6.1 illustrates that all honest replicas will transition to the new DAG starting from the same ending round. Refer to the consistency and completeness property of the DAG (Section 3.2.2), all honest replicas will execute transactions T_1 and T_2 within the same DAG. $\square$

Consequently, we can draw the conclusion that: If two transactions are isolated, they can be executed in any order. Otherwise, all the honest replicas will execute them in the same order. Thus, Thunderbolt holds its safety guarantees.

3.9.2. Proof of Liveness. In an environment where all replicas operate effectively, they will propose blocks within the same DAG. Each block suggested by the respective shard proposers will ultimately be committed. When a malicious replica is identified, the honest replicas will respond by proposing a Shift block. If fewer than $2f + 1$ Shift blocks are proposed, the DAG will remain unchanged, and all replicas will continue functioning within the current DAG while proposing new blocks. Conversely, if there are $2f + 1$ Shift blocks, all honest replicas will transition to the new DAG within the same round, as detailed in Section 3.2.2. After a minimum of $2f + 1$ honest replicas successfully relocate to the new DAG, they will be empowered to propose new blocks. Provided that all replicas maintain proper behavior, they will consistently propose blocks within the same DAG, ensuring that each shard proposer's proposed blocks are duly committed.

3.10. Serializability Proof of the Concurrency Controller

In this section, we will present the proofs that establish the correctness of *CC*. We will start by defining key concepts related to serializability, followed by a detailed analysis of our findings regarding correctness.

We consider a set of transactions $T = \{T_i\}$ (where $1 \leq i \leq n$) alongside a predefined commit order *CO*. *CC* generates a sequential order denoted as $SO = [T_1, T_2, \dots, T_n]$, producing an outcome represented as $OUT = [OUT_1, OUT_2, \dots, OUT_n]$ that each transaction reads and writes some values on some keys. Let *SE* denote one of the possible sequential orders of *SO*, with OUT' reflecting its corresponding outcomes. *CC* is deemed serializable if the condition $OUT = OUT'$ holds true.

Definition 7 (Read-Complete). *If T_i reads a value updated by T_j in *SO*, T_i will also read the value updated by T_j in *SE*. If transaction T_i reads a value updated by a transaction T_j in *SO*, then T_i will also read the value updated by T_j in *SE*.*

Definition 8 (Write-Complete). *If transactions T_i and T_j both write new values to key K , but T_i commits before T_j , which generates an order $T_i < T_j$ in SO , then T_i will also write the values to K before T_j when in SE .*

THEOREM 9. *CC is considered both Read-Complete and Write-Complete when the dependency graph G is always valid.*

PROOF. Firstly, if G is valid, we know that when a read node R_v^k (representing a node u that reads on key K) retrieves a value from a write node W_u^k on key K , all the write nodes that are updating values on K either have a direct path to u or a path from v . This guarantees the correctness of read-after-write operations. Consequently, if transaction T_i reads values updated by transaction T_j on key K , where T_j always exists (with the root being a write node), it follows that no other transactions updating values on K will be located between T_i and T_j . Therefore, for any execution order SO generated by G , where T_i reads the value updated by T_j , it can be inferred that T_j will also read the same value from T_i in SE . This demonstrates that CC is Read-Complete. Secondly, since SO is an execution order in CC , if T_i commits before T_j , then T_i will precede T_j in SO . Consequently, T_j will update the values after T_i in SE as well. Therefore, we can conclude that CC is Write-Complete. $\square$

THEOREM 10. *CC is serializable if CC is both Read-Complete and Write-Complete.*

PROOF. Since CC is Read-Complete, every transaction T_i that reads the values of certain keys will retrieve the same values regardless of the order generated by G . Additionally, since CC is Write-Complete, every transaction T_i that updates the values of certain keys will maintain a consistent order among all the orders generated by G . Therefore, the transactions will yield the same outcomes in any generated orders SO and SE ($OUT = OUT'$). $\square$

3.11. Experimental Evaluation: Concurrency Executor Evaluation

This section evaluates Thunderbolt by assessing its performance on the CE and the Thunderbolt protocol. We implement all the baseline comparisons using Apache ResilientDB (Incubating) [44, 117]. Apache ResilientDB is an open-source incubating blockchain project that supports various consensus protocols. It provides a fair comparison of each protocol by offering a high-performance framework. Researchers can focus solely on their protocols without considering system structures such as the network and thread models.

We will begin by comparing CE with two baseline protocols: OCC [123] and 2PL-NO-WAIT [164]. Additionally, we will analyze the performance of Thunderbolt, which is built on Tusk [124], and we will also

use Tusk as a baseline for our comparisons. For our input workload, we will utilize SmallBank [165, 166], a benchmarking suite that simulates common asset transfer transactions. This suite is also used to evaluate variant block systems [102, 167, 168, 169, 170, 171, 172].

3.11.1. Baseline Protocols. We implement OCC [123] and 2PL-NO-WAIT [173, 174] to compare the performance against our concurrent executor. We set up our experiments on AWS c5.9xlarge consisting of 36 vCPU, 72GB of DDR3 memory. We use LevelDB as the storage to save the balance of each account.

OCC Each executor is responsible for locally executing transactions. When an operation within a transaction T requires reading the value of a key K that the executor has not previously accessed during the execution, the executor will retrieve the value from the storage. Each value also contains a version to indicate the time the value was obtained. Any write operation will update the values locally. Upon completion of T , all the updated values will be forwarded to a central verifier. The verifier will cross-check the value versions by comparing them with the current versions in the storage. If there is a mismatch, the commit will be rejected, necessitating the re-execution of T .

2PL-No-Wait Each executor performs transactions by directly accessing the storage through a central controller. When an operation within a transaction T requires the read or update of a key K , the controller will lock K to prevent conflicts. If an operation seeks to access K but discovers that another executor has locked it, the executor will release all locks and re-execute T . Upon the completion of T , all the results will be transmitted to storage, and all locks will be released.

3.11.2. Experiment Setup With Smallbank. SmallBank [165] is a transactional system that comprises six distinct transaction types, five of which are designed to update account balances, while the remaining transaction is a read-only query that retrieves both checking and saving the account details of a user. Our focus is on two types of transactions: SendPayment and GetBalance, which are used to transfer funds between two accounts and retrieve account balances, respectively. Our objective is to evaluate the performance under varying read-write balance workloads. During a SendPayment transaction, the account balances are updated by reading the current balance and then writing the new values back. We created 10,000 accounts and conducted each experiment 50 times to obtain the average outputs.

We evaluated the impact of parallel execution. We measured the performance by uniformly selecting GetBalance with a probability of P_r while SendPayment with $1 - P_r$. We follow a Zipfian distribution to select accounts as transaction parameters and set the Zipfian parameter θ . The value of θ determines the

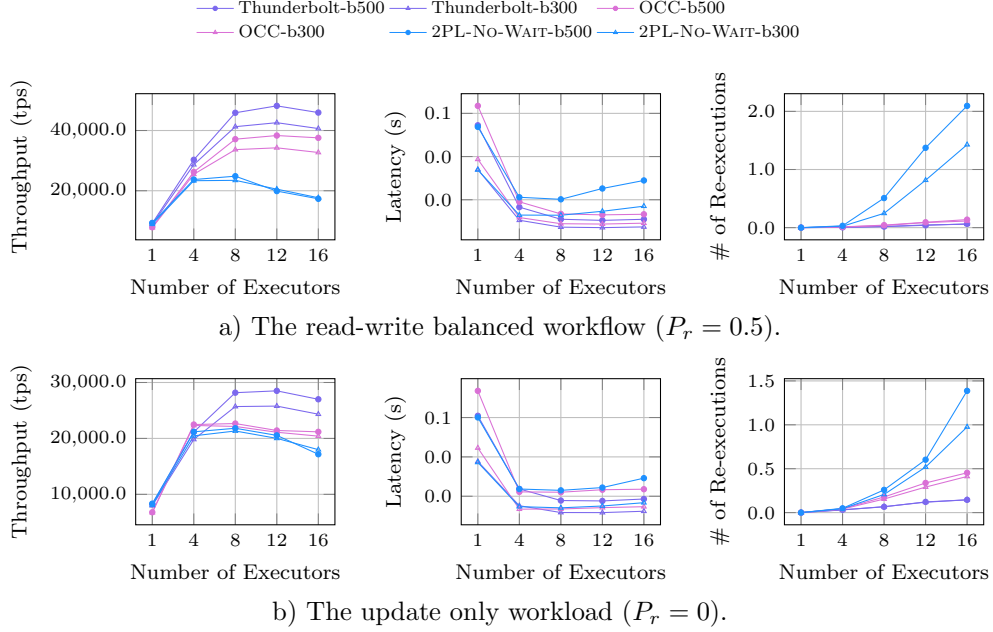


FIGURE 3.11. Evaluation of CE on different numbers of executors.

level of account contention, with higher values leading to higher contention. We focus only on data workloads with high contention by setting $\theta = 0.85$.

3.11.3. Impact from Concurrent Executor. We first evaluate the impact of increasing the number of executors to execute the transactions, then measure the aborts produced by each protocol. We ran two batch sizes $b300$ and $b500$ for each protocol: Thunderbolt- $b300$, Thunderbolt- $b500$, OCC- $b300$, OCC- $b500$, 2PL-NO-WAIT- $b300$, and 2PL-NO-WAIT- $b500$. We set $P_r = 0.5$ to measure a read-write balanced workflow and $P_r = 0$ on an update-only workflow.

Number of Executors In the read-write balanced workflow, the results depicted in Figure 3.11 (a) show that 2PL-NO-WAIT protocols with different batch sizes all experience a performance drop when increasing the number of executors beyond 8. However, Thunderbolt and OCC protocols with all the batch sizes obtain their highest throughputs on 12 executors and maintain stable throughput. Thunderbolt- $b500$ obtained 43K TPS while OCC- $b500$ achieved 35K TPS.

In the update-only workflow, the results shown in Figure 3.11 (b) indicate that OCC and 2PL-NO-WAIT stopped increasing earlier in 4 executors (both around 22K TPS) while Thunderbolt provides a peak throughput (28K TPS) in 12 executors.

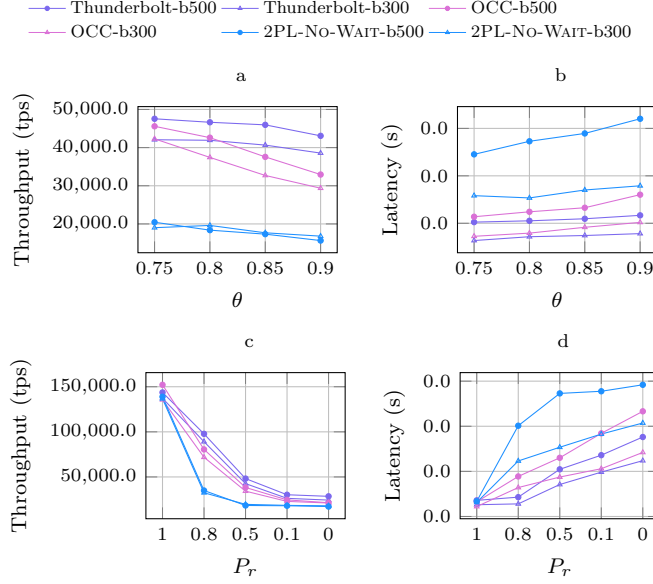


FIGURE 3.12. Throughput and average latency within varying θ settings with $P_r = 0.5$ (a and b) and varying P_r settings with $\theta = 0.85$ (c and d).

These experiments demonstrate that all the protocols do not obtain significant benefits for a large number of executors in a high-competition workflow. However, Thunderboltcan still achieve more parallelism with more executors.

Evaluation of Abort Rates As we increased the number of executors, we also measured the average number of re-executions for the transactions. The results in Figure 3.11 indicate that when the number of executors goes beyond 8, all 2PL-NO-WAIT protocols experience a significant increase in the rate of abortions, leading to a drop in throughput from 24k to 18k in the read-write balanced workflow. While OCC protocols provide a lower rate within the read-write balanced workflow. However, Thunderbolt achieves the lowest abortions, with Thunderbolt-b500 reducing 50% of the abortions from OCC-b500 and 90% from 2PL-NO-WAIT-b500 in all the experiments.

Evaluation of θ We will now provide an evaluation with various θ values, using a read-write balance workload of $P_r = 0.5$. In particular, we examine high contention workloads where $0.75 \leq \theta \leq 0.9$, as this range represents the primary focus of our study. The results illustrated in Figure 3.12 (a and b) demonstrate that at $\theta = 0.75$, both OCC and Thunderbolt show comparable performance. However, as θ increases to 0.9, OCC experiences a significant decline in performance, while Thunderbolt continues to achieve higher throughput levels. In contrast, the 2PL-NO-WAIT approaches steady throughput, attributable to its locking strategy.

Evaluation of P_r In this evaluation, we will look at how different values of P_r affect the read and write ratio in the workload, with $\theta = 0.85$. The results in Figure 3.12(c and d) show that when $P_r = 1$ (all read), all protocols behave similarly. However, the OCC protocol performs slightly better because it allows non-blocking local executions. When conflicts happen, the 2PL-NO-WAIT shows a sharp decline in performance. In contrast, both Thunderbolt and OCC also perform worse when we decrease P_r , which leads to more conflicts. On the other hand, as we increase the value of P_r , Thunderbolt achieves better throughput and lower latency than OCC, even when all operations are write-only. All protocols show similar latency at $P_r = 1$, but as P_r increases, the latency for 2PL-NO-WAIT rises, while Thunderbolt continues to perform better than OCC.

3.12. Experimental Evaluation: System Evaluation

We conducted evaluations to determine the impact of Thunderbolt built on Tusk. In our evaluation, we compared the performance of Thunderbolt with Tusk, which executes transactions in order after reaching a total order after DAG protocols. We evaluated the impact of different components of the protocol by comparing the results between three systems:

- Thunderbolt: Leverage CE + parallel verification.
- Thunderbolt-OCC: Combine OCC + parallel verification.
- Tusk: Utilize the OE model.

As the serialized verification will execute the transactions in order to verify the results, we will expect that any DAG-based protocols with serialized verification will provide the same behavior with Tusk. We also leveraged SmallBank as the input workload.

Each replica was configured with CE comprising 16 executors to process transaction batches of 500, alongside 16 validators to verify blocks post-consensus. We scaled the system from 8 to 64 replicas. By default, K' was set to a sufficiently large value to disable shard rotation. In the final phase of our evaluation, we examined the impact of varying K' values, which govern the frequency of shard reconfiguration (Section 3.12).

SmallBank Throughout the system evaluation, we focus on the SmallBank workload with a read-write balanced scenario ($P_r = 0.5$), where half of the transactions are read-only.

In the smallbank workload, Transaction addresses were selected from a pool of 1000 users with a skew parameter $\theta = 0.85$ to simulate a high-contention workload.

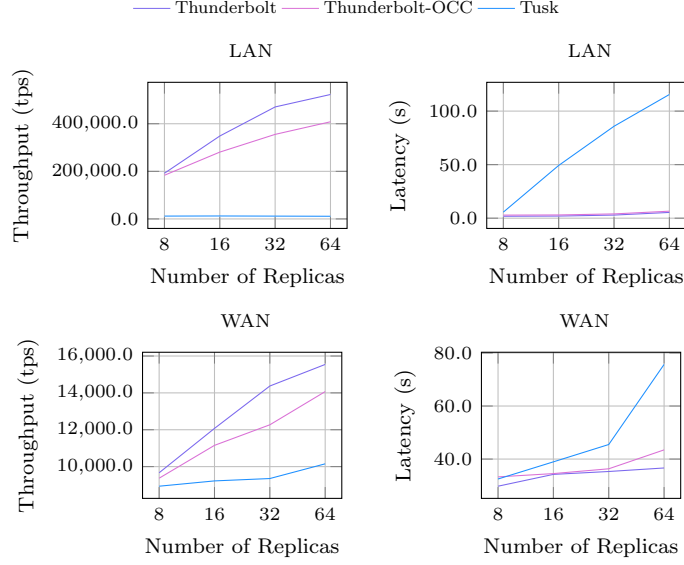


FIGURE 3.13. Throughput and average latency within different replicas within $P_r = 0.5$ in LAN and WAN.

The LAN results, as shown in Figure 3.13, demonstrate that Thunderbolt significantly outperformed Tusk’s sequential execution model, achieving a $50\times$ speedup. Specifically, Thunderbolt reached 500K TPS with 64 replicas, compared to Tusk’s 11K TPS. This improvement highlights the benefits of executing transactions in parallel.

We also compared Thunderbolt with Thunderbolt-OCC, which replaces the *CE* with *OCC*. While Thunderbolt-OCC matched Thunderbolt’s throughput at 8 replicas, it lagged behind at scale, achieving only 400K TPS with 64 replicas. Furthermore, Thunderbolt maintained a low transaction latency of 5 seconds, whereas Tusk’s latency soared to 100 seconds under the same conditions.

The WAN results demonstrate similar behavior but with higher latency. The latency gap between Thunderbolt and Tusk becomes smaller because the WAN latency begins to dominate cost, becoming a bottleneck.

Cross-shard Transactions Next, we evaluated the impact of Cross-shard TXs using 16 replicas. We randomly assigned a percentage $P\%$ ($0 < P \leq 100$) of transactions to be processed by two shards. Additionally, we assessed the benefits of parallel execution by comparing Thunderbolt-OCC.

As shown in Figure 3.14, the performance of both Thunderbolt and Thunderbolt-OCC declined as the percentage P increased. In scenarios with only single-shard transactions ($P = 0$), both systems achieved 100K TPS. However, when P increased to 8%, Thunderbolt-OCC’s throughput dropped to 16K TPS, while

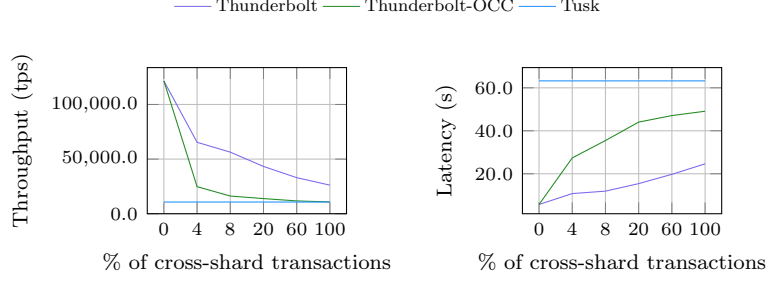


FIGURE 3.14. Throughput and average latency within different ratios of cross-shard transactions within 16 replicas.

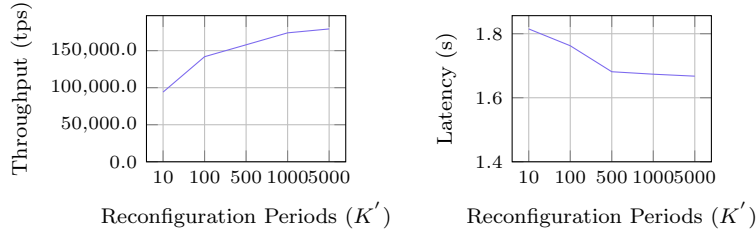


FIGURE 3.15. Throughput and average latency within different reconfiguration periods within 8 replicas.

Thunderbolt maintained a significantly higher throughput of 64K TPS. Thunderbolt-OCC’s performance aligned closely with Tusk, achieving approximately 10K TPS. In contrast, Thunderbolt delivered 19K TPS even when all transactions were cross-shard, demonstrating the advantages of its parallel execution model and non-deterministic ordering on CE .

Latency metrics further highlighted Thunderbolt’s superiority. Under high-contention workloads, Thunderbolt achieved a transaction latency of 24s seconds, while Thunderbolt-OCC’s latency was nearly double at 50 seconds.

Reconfiguration Periods Now, we analyze the performance using different reconfiguration periods K' to transition the shard proposers into a new DAG on 8 replicas. Figure 3.15 demonstrates that Thunderbolt exhibited lower performance with smaller K' values (80K TPS with $K' = 10$), attributed to the costly transition between DAGs. Conversely, when K' was increased to over 1000, Thunderbolt demonstrated significantly improved stability, achieving a throughput of 180K TPS. Additionally, the average latency decreased from 1.9s to 1.7s as K' increased from 10 to 5000. Figure 3.16 also shows the average run time of committing proposals per 100 rounds, that is $\frac{1}{100} \sum (T_{commit(i)} - T_{commit(i-1)})$ where $T_{commit(i)}$ is the time of

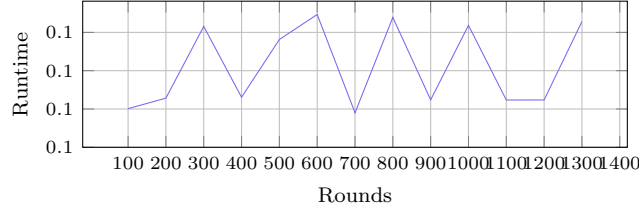


FIGURE 3.16. Average latency per 100 rounds and reconfigure the shard per 300 rounds.

committing round i . We set K' as 300 and it was demonstrated that Thunderbolt will not get stuck during the reconfiguration. The runtime of each round is around $0.07s$ to $0.1s$.

Failures Finally, we evaluated the impact of replica failures within 16 replicas. We forced f replicas ($f = 1$ or $f = 2$) to stop working during the experiments. We randomly designated a percentage $P\%$ ($0 < P \leq 100$) of the transactions to be processed by two shards.

Figure 3.17 reveals that Thunderbolt still can provide higher throughputs when some shards stop working. When one replica failed to propose transactions, Thunderbolt 1 ($f = 1$) obtained $78K$ TPS working on all Single-shard TXs ($P = 0$) and $17K$ TPS on all Cross-shard TXs while when two replicas failed to propose transactions, Thunderbolt 2 ($f = 2$) obtained $66K$ TPS on all Single-shard TXs and $15K$ TPS on all Cross-shard TXs. However, the results show that the latency remains stable even if some replicas fail to process, benefiting from the leader rotation of the DAG protocols.

3.13. Related work

Sharding on DAG-based protocol: Several studies [22, 135, 136, 137, 138, 139, 175, 176] have underscored the necessity of implementing sharding to enhance scalability within blockchain systems. Directed Acyclic Graph (DAG)-based blockchains [124, 124, 125, 125, 126, 126, 127, 128, 129, 130, 131, 132, 133, 134, 177, 178, 179, 180, 181, 182] present a promising alternative for improving concurrent transaction processing through the utilization of the DAG data structure. However, it is noteworthy that only a limited number of approaches offer effective sharding strategies for DAG protocols [138, 143, 144, 145, 147] These strategies often depend on account-based mechanisms to achieve eventual atomicity, employing incentive mechanisms through either the two-phase commit (2PC) protocol or additional coordinating entities.

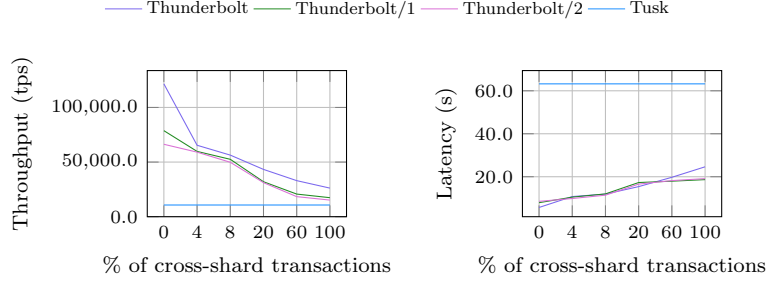


FIGURE 3.17. Throughput and average latency within different ratios of cross-shard transactions within 16 replicas when f ($f = 1$ or $f = 2$) replicas failed.

Execute-Order-Validate: Hyperledger Fabric [20] introduces the Execute-Order-Validate (EOV) framework, which demonstrates high performance primarily in low-contention workloads. To enhance this framework, various techniques have been developed [102, 171, 183, 184, 185, 186], including methods for reordering transactions within a block and improving execution processes. In contrast, Thunderbolt addresses scalability by distributing transactions across multiple shards, thereby enabling each shard proposer to execute transactions in parallel.

Concurrent Execution: Deterministic approaches [119, 120, 121] have been proposed to improve the efficiency of transaction execution by creating a dependency graph. This graph allows for concurrent execution while minimizing conflicts between transactions. Furthermore, segmenting transactions [187, 188, 189, 190] has been introduced as an effective method for reducing these conflicts. CHIRON [191] and BlockSTM [192] offer nondeterministic execution by extracting dependencies from smart contracts or by determining the execution order based on transaction arrival times. Conversely, Thunderbolt adopts a different methodology that it does not depend on arrival times or read/write set information for transaction processing. Instead, it dynamically assigns execution orders to effectively minimize transaction conflicts.

Concurrent Consensus: A robust and scalable blockchain framework is crucial for the implementation of real-world applications [193, 194, 195, 196]. For example, the PoE [118] model utilizes speculative execution protocols, and protocols, including RCC [117], FlexiTrust [197], and SpotLess [198], incorporate multiple leaders to enhance parallel processing capabilities. However, these protocols currently do not support reconfiguration.

3.14. Summary and Discussion

We introduce Thunderbolt, an innovative sharding system designed to enhance smart contract execution by integrating the Order-Execute and Execute-Order-Validate models to efficiently handle both single-shard

transactions (Single-shard TXs) and cross-shard transactions (Cross-shard TXs). We have developed a concurrent executor that significantly improves the performance of Single-shard TXs without requiring prior knowledge of read/write sets. Additionally, Thunderbolt effectively distributes transactions across multiple shards and utilizes Directed Acyclic Graph (DAG)-based protocols to maintain consistency between single-shard and cross-shard transactions. A key feature of Thunderbolt is its ability to utilize the inherent properties of the DAG to facilitate a non-blocking transition to a new DAG structure. This allows for the rotation of proposers for each shard, helping to prevent malicious activity from any single proposer. Our performance evaluations show that Thunderbolt achieves an impressive speedup of up to $50\times$ compared to the native execution provided by Tusk.

Ensuring Availability: CASSANDRA for Partial Progress Under Network Asynchrony

4.1. Introduction: The Achilles' Heel of Leader-Based Protocols—Network Partitions

Blockchain technology has emerged as a leading solution in modern distributed transaction systems. It has significantly advanced decentralized digital transformation by facilitating peer-to-peer transactions and removing the necessity for intermediaries. This technology incorporates essential features such as decentralization, transparency, immutability, and traceability, which enhance its applicability across various use cases. At the core of blockchain are consensus protocols, which are vital for ensuring that transactions achieve a global order. Protocols such as Practical Byzantine Fault Tolerance (PBFT) [29] are instrumental in guaranteeing data security among all participants involved in the system.

Consensus protocols have garnered significant attention within the field of distributed systems as essential mechanisms for establishing reliable systems. These protocols are widely recognized as fundamental components and have undergone extensive research focused on enhancing throughput, minimizing latency, and improving scalability. Research in the area of consensus has distinctly outlined the boundaries related to safety and liveness (or progress) guarantees that can be provided. The CAP theorem [199], introduced by Brewer, illustrates a critical trade-off inherent in distributed systems: the inability to simultaneously ensure both safety and liveness. To clarify, an algorithm is deemed safe, or consistent, if it ensures that no adverse events occur; that is, every response sent to a client is accurate. Conversely, an algorithm is considered live if it guarantees that some positive outcome will eventually occur. Within distributed systems, availability serves as the liveness property, ensuring that every request ultimately receives a response. Consequently, in an unreliable distributed system, designing a system that maintains both consistency and availability, particularly during network partitions, remains a significant challenge [15, 16, 17, 200].

Unfortunately, blockchain systems operate in unreliable networks, and network failures occur frequently. These network partitions can cause the system to stop functioning, leading to financial losses. For example, shortly after the introduction of the first Bitcoin failure on December 11, Bitcoin experienced an 80% loss

in value within the following year [201]. In 2020, Ethereum faced significant network outages and a possible chain split on the morning of November 11, caused by the infrastructure provider Infura’s use of outdated client software and an unexpected code change [202]. Similarly, Solana experienced disruptions in 2021 due to high load from NFT minting and duplicate transactions [203]. Therefore, enterprise blockchain projects face challenges in addressing availability issues during network partitions [204, 205]. This brings us to the question of whether a system can keep processing or undergo partial progress under network failures.

In the realm of large-scale distributed systems, leader-based partial-synchrony replication protocols are integral to ensuring consistent results for clients such as Paxos [24] in Crash Fault Tolerant (CFT) systems while PBFT [29] and HotStuff [11] in Byzantine Fault Tolerant (BFT) systems.

While these protocols effectively maintain data consistency, they rely on a single leader for transaction commitment and do not guarantee availability. Common occurrences of replica crashes and network partitions [206] often lead to diminished system responsiveness and potential operational impasses. To address these challenges, contemporary clusters integrate fault-tolerant coordination engines, such as Chubby [207], Boxwood [208], or ZooKeeper [209], to oversee leader election and coordination. These engines enable seamless leader changes and transaction relocation in the event of leader failures. Notably, these protocols heavily rely on timeouts due to their partial-synchrony assumptions. Consequently, in the event of leader failure, a non-trivial leader selection process is triggered, potentially resulting in decreased system liveness under adverse network conditions.

While current leader-based methodologies have presented various solutions to improve the consistency and availability of systems, the performance of the protocols continues to be a significant challenge. Raft [25], MultiPaxos [210], and RCC [117] seek to enhance throughputs by operating multiple instances in parallel, yet they still rely on a single leader to propose transactions. Conversely, DAG-based consensus mechanisms such as ePaxos [211] and Tusk [26] function without a designated leader, allowing replicas to achieve transaction consensus as long as a majority is formed. The adoption of leaderless consensus protocols has demonstrated effectiveness in upholding system consistency and availability. However, in cases where the system encounters partitions and replica failures to form a quorum for consensus, these protocols still suffer from system outages.

In this chapter, we present CASSANDRA, a system designed to support both crash-fault tolerant (CFT) and Byzantine-fault tolerant (BFT) systems. CASSANDRA operates as a leaderless protocol, allowing each replica to independently propose transactions from clients. CASSANDRA functions in rounds, and each proposer commits proposals in two phases: proposal exchange and proposal commitment. In the proposal exchange phase, proposers share their respective proposals with one another. Subsequently, each proposer selects the

strongest proposal from the sharing received and votes for that proposal. During the proposal commitment phase, these votes facilitate an agreement on a global strongest proposal, which is then committed by each proposer.

Proposals are structured as a Directed Acyclic Graph (DAG) to establish a partial order among all the proposals. When the strongest proposal is committed, all proposals that are in a partial order with respect to this strong proposal will also be committed. This allows CASSANDRA to achieve partial progress even when the network is partitioned. If the strongest proposals selected by each proposer do not receive enough votes, the proposers will skip the commitment and move on to the next round.

This chapter makes the following main contributions:

- We propose a leaderless protocol called CASSANDRA aiming with an Auto-Synchronous replication model to achieve non-zero throughput on both CFT and BFT systems. Proposers in CASSANDRA are able to process progress under network partition by building a Direct Acyclic Graph (DAG).
- CASSANDRA does not rely on timeouts to avoid waiting for the commitments of previous rounds, leading to system blocking.
- Novel adaptation techniques enabling CASSANDRA to optimize its leader choice to avoid suffering from leader failures or leader attacks.
- We deploy our CASSANDRA protocol in the wild over the Twitter data on social interactions such as tweeting (with mentions) and re-tweeting along with quoting, replying, and liking a tweet. Our results show that once the network is restored, the system throughput peaks at $6\times$ of the steady state due to partial progress made during the partition.
- We evaluated a prototype of CASSANDRA on Apache ResilientDB (Incubating) [212] against Multi-Paxos [42] for CFT setting while Tusk [26] and RCC [117] for BFT setting.

4.2. System Design: CASSANDRA’s Leaderless, DAG-Based Protocol

In this section, we provide the design of CASSANDRA for both CFT and BFT.

We first illustrate a basic protocol in this section which contains two phases to commit the transactions. Then, we propose an optimized version, Fast CASSANDRA, to reduce two phases into one to improve the performance.

4.2.1. System Model. Prior to explaining the design of CASSANDRA, we lay out the system model.

We make standard assumptions also made by existing CFT protocols [24, 25] and BFT protocols [29]. We assume a replicated system $\mathcal{S} = \{\mathcal{R}, \mathcal{C}\}$, where $\mathcal{R}$ denotes the set of replicas, and $\mathcal{C}$ denotes the set of clients. Each client $C \in \mathcal{C}$ can send its transactions to any of the replicas $R \in \mathcal{R}$.

This replicated system has a total of n replicas, of which at most f are faulty. The f faulty replicas can fail from stopping or crashing in CFT while they can behave maliciously in BFT. We adopt the same partial synchrony model as existing CFT and BFT protocols; We assume an eventually synchronous network [155] that messages sent from a replica will ultimately arrive in a global stabilization time (GST), which is unknown to the replicas. Safety is guaranteed under asynchrony, while liveness is only guaranteed during periods of synchrony.

We assume the number of replicas in CFT is $n=2f+1$ while it is $n=3f+1$ in BFT setting. Furthermore, we refer to MP as the majority of replicas as $f+1$ in CFT and $2f+1$ in BFT systems.

To simplify the description, in the following sections, we refer to replicas as honest replicas, which are non-faulty.

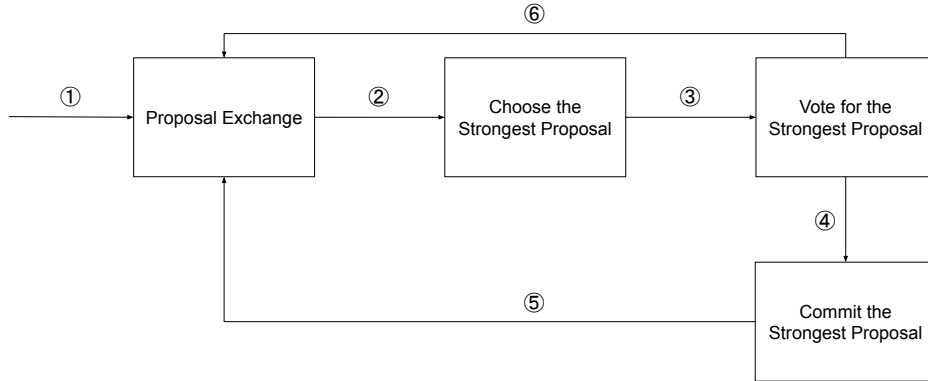


FIGURE 4.1. CASSANDRA workflow overview.

4.2.2. Cassandra Workflow Overview. Figure 4.1 illustrates the workflow of CASSANDRA, in which each replica, also called a proposer, runs in rounds to propose transactions. In each round, each replica shares a proposal with each other proposer in the proposal exchange phase (①). The proposal not only includes the new transactions from the clients but also refers to the received proposals to provide a change to those uncommitted proposals to be committed, which forms a directed acyclic graph (DAG). Then, each proposer chooses the strongest proposal as the leader proposal among the proposals received in this round (②). A proposal voting phase is followed to vote for the leader proposal utilizing a consensus protocol (③).

If a majority of the replicas agree on the same proposal, each replica will commit all the proposals referred by that leader proposal (④) and start the next round (⑤). Otherwise, if the consensus protocol fails to commit due to divergent voting or some failures, like network partitions or malicious leaders, the replicas will stop waiting and start exchanging the new proposal in the next round (⑥).

However, unlike most of the DAG-based protocols [26, 125], CASSANDRA does not rely on a reliable broadcast. Each proposer will go to the next step even if it does not receive enough messages from other proposers by starting a timer in each step. These timers allow CASSANDRA with the capability to process the proposals partially under an unhealthy network.

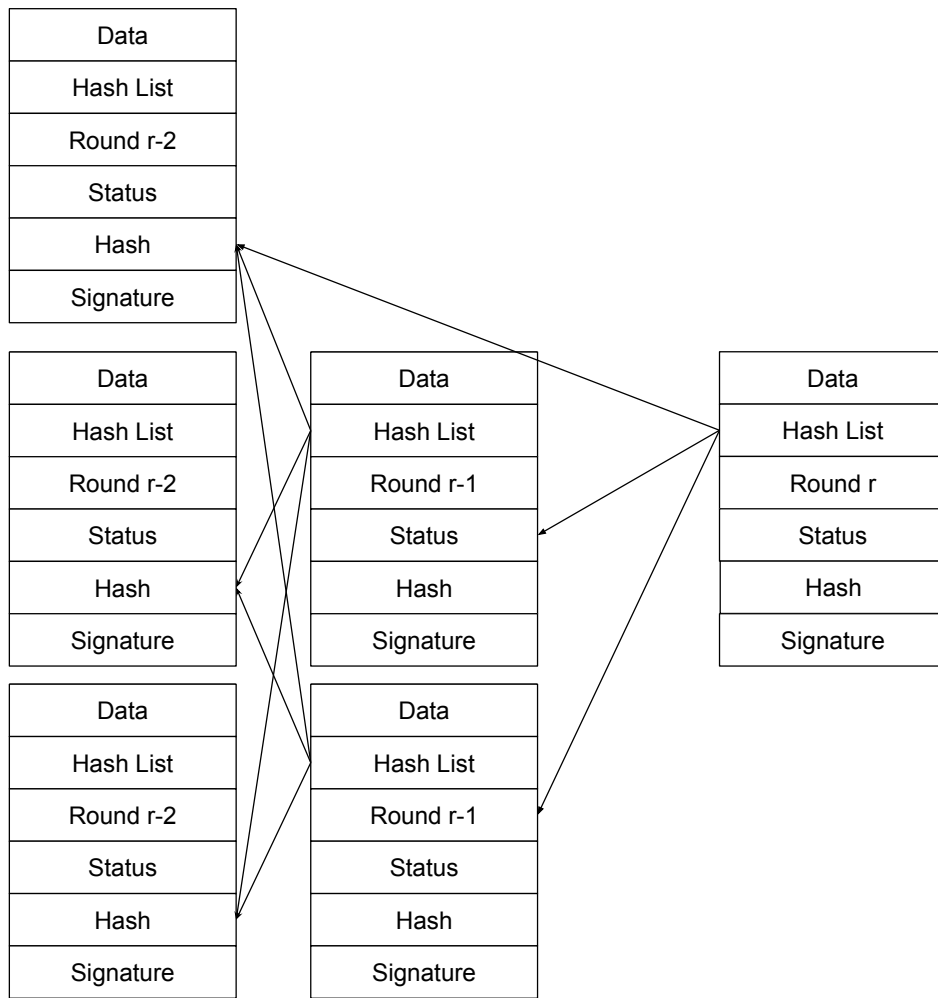


FIGURE 4.2. The strongest proposals in CASSANDRA generate a single chain, while each strongest proposal may include the proposals not being chosen as the strongest proposals.

Process transactions in round r by replica R_i :

- 1: Let G be the local graph in R
 - 2: Let MP be the set containing a majority of the proposers.
 - 3: **event** Receive a transaction T from the client in round r **do**
 - 4: Generate a proposal p by build a graph connecting all the proposals R_i has received in previous rounds
 - 5: Broadcast p and wait for the proposals P from all the proposers or the timer times out.
 - 6: **for** Each proposal p' in P **do**
 - 7: Merge p' with G
 - 8: Select the proposal SP_r with highest priority from $resp$
 - 9: Vote(SP_r)
-

FIGURE 4.3. Proposal Exchange in round r to share proposals among the replicas and vote for the strongest local proposal.

4.2.3. Proposals. Each proposer will generate a proposal in each round including the transactions received from the clients and broadcast the proposal to other proposers. Each round will vote for one proposal as the leader proposal called the strongest proposal (SP). The voted proposals must have a reference to the last SP chosen in the previous round. Thus, these strongest proposals (SPs) generate a unique single chain in CASSANDRA.

As only one proposal will be chosen, to allow other proposals to have a chance to be committed, each new proposal will refer to the unchosen proposals as well, resulting in a directed acyclic graph (DAG) (Figure 4.2).

4.2.4. Proposal Exchange. CASSANDRA, similar to traditional consensus protocols [24], operates in rounds to designate the strongest proposal as the leader responsible to commit the proposals. In each round, every proposer presents a new proposal (Section 4.2.3) to all other proposers and awaits the proposals of all the proposers. The new proposal must connect to the last SP the proposer obtained.

To guarantee that the network issue will not stop the liveness of the system, the proposer will set up a defined time period T_{ex} . If the proposer cannot receive all the proposals within time T_{ex} , it will stop waiting and go into choosing the strongest proposal, called local strongest proposal (LSP).

Unlike traditional timeouts, T_{ex} is set to a sufficiently small value to not interfere with the system. The purpose of employing a timer in this context is to maximize the collection of responses from other replicas, thereby facilitating each proposer to share their proposals even under adverse network conditions.

4.2.4.1. Proof of Availability. As shown in Section 4.2.3, each new proposal will refer the unchosen proposals in previous rounds with their digests. Once a proposer P receives a new proposal from other proposers, it will validate the referred proposals. If P has not obtained those proposals, P will ask their proposers to resent their contents. If P fails to obtain the content, P will discard this proposal.

However, this will bring a challenge for a malicious attack. If a malicious proposer delivers distinct proposals to proposers A and B , each of which is referred by the new proposals, they need to send additional messages to exchange these two proposals. These additional messages may result in a DDOS attack [?].

To resolve this malicious attack, CASSANDRA only allows each proposer to refer to its own proposals. If a proposer refers to some existing proposals in its new proposals, it will suffer from a heavy request to ask for the contents. Figure 4.3 shows the procedure for exchanging proposals in round r .

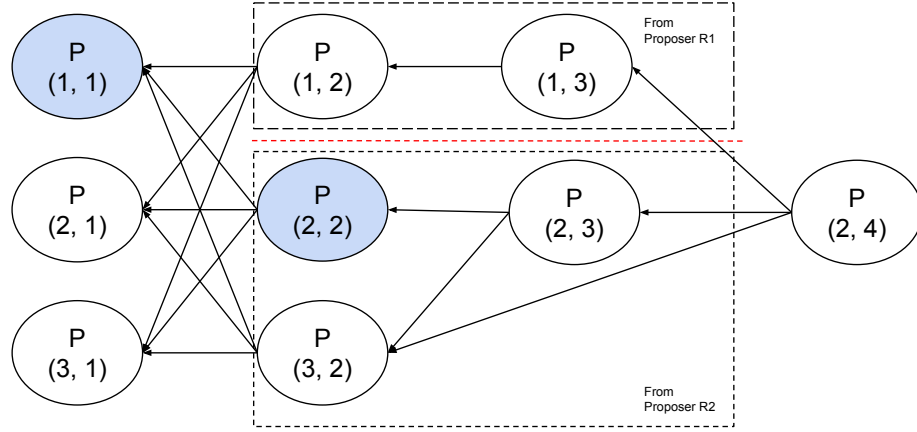
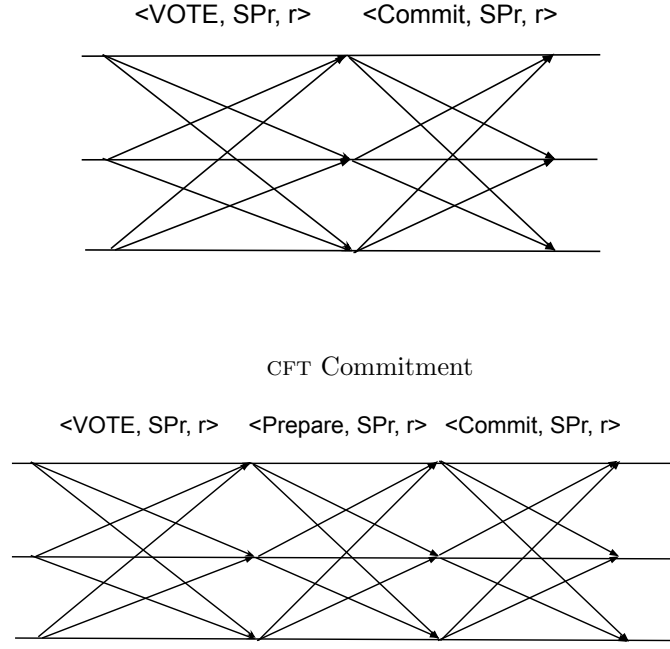


FIGURE 4.4. Merge proposals from three replicas.

4.2.5. Select the Strongest Proposal. Upon reception of proposals from distinct replicas, the proposer will proceed to assign priorities to each proposal. This process aims to designate a proposal as the leader to commit the history to the proposal graph. In the event that the number of proposals is less than MP , the proposer will still choose the leader to notify other replicas of the decision on this round.

As only one proposal is selected as the leader proposal to be committed, each proposal will connect to the proposals in previous rounds to form a DAG. Those proposals connected to the leader proposal will also be committed. We will discuss the priority algorithm in Section 4.2.9.

The priority assignment is a benefit for the proposers running in a Byzantine environment in which malicious proposers can choose their favorite ones in their subgraph, resulting in the delay of other proposals. The priority assignment is deterministic and predicated on the commit history of each proposer. The proposal with the highest priority will be chosen as the local strongest proposal in that round.



BFT Commitment

FIGURE 4.5. Strongest proposals voted from each proposer will be committed by a consensus agreement.

-
- 1: **event** Vote for the strongest proposal SP_r in round r **do**
 - 2: Broadcast $\langle \text{VOTE}, SP_r, r \rangle$
 - 3: **event** Received $f + 1$ identical vote messages for a strongest proposal SP_r in round r **do**
 - 4: Broadcast $\langle \text{Commit}, SP_r, r \rangle$
 - 5: Mark SP_r as *Voted*
 - 6: **event** Received $f + 1$ identical commit messages in round r **do**
 - 7: Commit SP_r
 - 8: Mark SP_r as *Committed*
 - 9: **event** Received all vote messages in round r but no $f + 1$ vote messages for a strong proposal SP_r **do**
 - 10: Go to next round.
 - 11: **event** A timer T times out but no $f + 1$ vote messages for a strong proposal SP_r **do**
 - 12: Go to next round.
-

FIGURE 4.6. Proposal Commitment for CFT.

4.2.6. Consensus on Strongest Proposal. Following a proposer's vote for the local strongest proposal, SP_r , in round r , the proposer initiates the proposal commitment phase to agree on a global strongest proposal for round r .

In the proposal commitment phase, each proposer broadcasts its local strongest proposal SP_r via a vote message $\langle \text{VOTE}, SP_r, r \rangle$. Then, the proposal will go through a series of phases to reach an agreement: 2

```

1: event Vote for the strongest proposal  $SP_r$  in round  $r$  do
2:   Broadcast a vote message  $\text{iVOTE}, SP_r, r_i$ 

3: event Received  $f + 1$  identical vote messages for a strongest proposal  $SP_r$  in round  $r$  do
4:   Broadcast a prepare message  $\text{iPrepare}, SP_r, r_i$ 
5:   Mark  $SP_r$  as Voted
6: event Received  $2f + 1$  identical prepare messages in round  $r$  do
7:   Broadcast a commit message  $\text{iCommit}, SP_r, r_i$ 
8:   Mark  $SP_r$  as Prepared
9: event Received  $2f + 1$  identical commit messages in round  $r$  do
10:  Commit  $SP_r$ 
11:  Mark  $SP_r$  as Committed
12: event Received all vote messages in round  $r$  but no  $2f + 1$  vote messages for a strong proposal  $SP_r$  do
13:  Go to next round.
14: event A timer  $T$  times out but no  $2f + 1$  vote messages for a strong proposal  $SP_r$  do
15:  Go to next round.

```

FIGURE 4.7. Proposal Commitment for BFT.

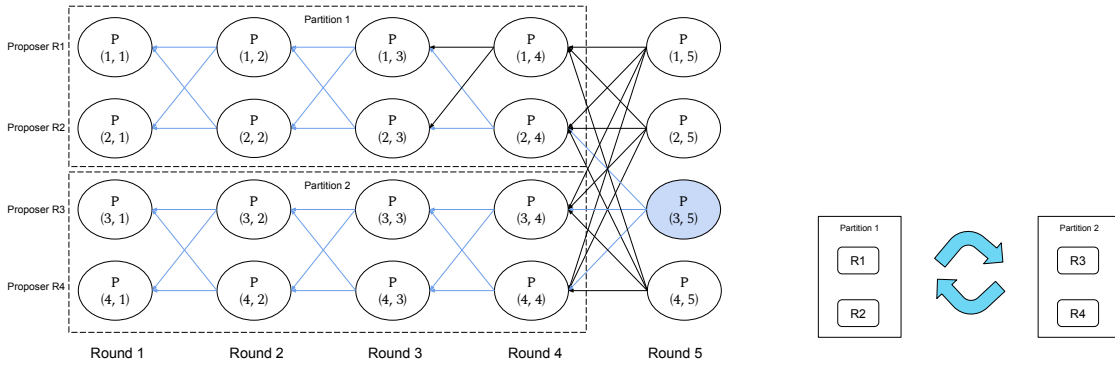


FIGURE 4.8. Partial Progress in two partitions while network failures. Two partitions are converged when the network is recovered. Proposals in previous rounds (round 1-4) are committed by the new proposal $P(3, 5)$. Proposals only need to be transferred between the replicas in each partition.

phases in CFT while 3 phases in BFT. Figure 4.6 and Figure 4.7 show the algorithms to commit the strongest proposals in CFT and BFT.

4.2.6.1. CFT. For CFT, upon the receipt of identical vote messages $\langle VOTE, SP_r, r \rangle$ from MP , like $f + 1$ replicas, each replica broadcasts a commit message $\langle COMMIT, SP_r, r \rangle$ and mark the proposal SP_r as *Voted*. Upon the receipt of identical commit messages from MP , commit the proposal SP_r for round r and the proposals in its subgraph and mark the SP_r as *Committed*.

4.2.6.2. BFT. For BFT, there will be an additional phase. upon the receipt of identical vote messages $\langle VOTE, SP_r, r \rangle$ from MP , like $2f + 1$ replicas, each proposer broadcasts a prepare message $\langle PREARE, SP_r, r \rangle$ and mark SP_r as *Voted*. Upon the receipt of identical prepare messages $\langle PREARE, SP_r, r \rangle$ from MP , each proposer broadcasts a commit message $\langle COMMIT, SP_r, r \rangle$ marking SP_r as *PreCommitted*.

Upon the receipt of identical commit messages from MP , commit the proposal SP_r for round r and the proposals in its subgraph and mark SP_r as *Committed*.

```

1: Let  $G$  be the local graph.
2: event Vote for the strongest proposal  $SP_r$  in round  $r$  with its history graph  $G_{SP}$  do
3:   Broadcast ¡VOTE,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 

4: event Received  $f + 1$  identical vote messages ¡VOTE,  $SP_r$ ,  $G_{SP}$ ,  $r_i$  for a strongest proposal  $SP_r$  in round  $r$  do
5:   CheckCommit( $G_{SP}$ )
6:   Broadcast ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 
7:   Mark  $SP_r$  as Voted
8: event Received  $f + 1$  identical commit messages ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ , in round  $r$  do
9:   CheckCommit( $G_{SP}$ )
10:  Broadcast ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 
11:  Mark  $SP_r$  as Committed
12:  Commit  $SP_r$ 
13: event Received all vote messages in round  $r$  but no  $f + 1$  vote messages for a strong proposal  $SP_r$  do
14:  Go to next round.
15: event A timer  $T$  times out but no  $f + 1$  vote messages for a strong proposal  $SP_r$  do
16:  Go to next round.
17: function CheckCommit (Graph  $G_{SP}$ )
18:   Merge  $G_{SP}$  into  $G$ .
19:   for Each proposal  $p$  in  $G_{SP}$  do
20:     if The  $f + 1$  messages contain  $p$  then
21:       if  $p$  is marked as Voted then
22:         Mark  $p$  as Committed

```

FIGURE 4.9. Commit proposals and recovery the history for CFT.

```

1: Let  $G$  be the local graph.
2: event Vote for the strongest proposal  $SP_r$  in round  $r$  with its history graph  $G_{SP}$  do
3:   Broadcast ¡VOTE,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 

4: event Received  $f + 1$  identical vote messages ¡VOTE,  $SP_r$ ,  $G_{SP}$ ,  $r_i$  for a strongest proposal  $SP_r$  in round  $r$  do
5:   CheckCommit( $G_{SP}$ )
6:   Broadcast ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 
7:   Mark  $SP_r$  as Voted
8: event Received  $f + 1$  identical commit messages ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ , in round  $r$  do
9:   CheckCommit( $G_{SP}$ )
10:  Broadcast ¡Commit,  $SP_r$ ,  $G_{SP}$ ,  $r_i$ 
11:  Mark  $SP_r$  as Committed
12:  Commit  $SP_r$ 
13: event Received all vote messages in round  $r$  but no  $f + 1$  vote messages for a strong proposal  $SP_r$  do
14:  Go to next round.
15: event A timer  $T$  times out but no  $f + 1$  vote messages for a strong proposal  $SP_r$  do
16:  Go to next round.
17: function CheckCommit (Graph  $G_{SP}$ )
18:   Merge  $G_{SP}$  into  $G$ .
19:   for Each proposal  $p$  in  $G_{SP}$  do
20:     if The  $f + 1$  messages contain  $p$  then
21:       if  $p$  is marked as Voted then
22:         Mark  $p$  as Committed

```

FIGURE 4.10. Commit proposals and recovery the history for BFT.

4.2.7. Failures. As a replica may experience message delay or network partition, the replica may not obtain sufficient voted messages for the local strongest proposal. To guarantee the liveness of the replicas, we assign a commit timer T_{commit} . When a timer T_{commit} times out, and the replica could not receive enough vote messages for a proposal, or the replica has received all the proposals, the replica will skip the commit phase and go into the next round.

As a replica will skip the current round due to the timeout, a committed proposal may be skipped as well. Thus, while making an agreement on the strongest proposal in the current round, it needs to replay the decisions in the previous round. However, it is hard to determine if a proposal has been committed without a single leader. To address this challenge, when we vote for the strongest proposal SP_r for round r , the proposal will include the history as well. Therefore, while processing the consensus for SP_r , we can also replay the consensus for the previous strongest proposals. Figure 4.9 and Figure 4.10 show the algorithm to run a consensus on the history of SP_r . For each consensus phase, if a majority of replicas have confirmed the strongest proposal can be committed, then this strongest proposal will be marked as *Committed*.

4.2.8. Commitment. Before replica R commits the strong proposal SP_r after the consensus in round r , R needs to determine all the globally strongest proposals in previous rounds. However, the recovery process in Section 4.2.7 already helps us to recover the status of each strongest proposal. Thus, while committing SP_r , we will go through each round i in its history graph and commit each global strongest proposal SP_i that has been marked as *Committed* in round i . If no global strongest proposal in round r , that round will be skipped. Figure 4.11 shows the procedure to commit the strongest proposal in round r .

```

1: Let  $lcr$  be the last round to commit the strongest proposal.
2: event Commit the strongest proposal  $SP_r$  in round  $r$  do
3:   for round  $r'$  from  $lcr + 1$  to  $r$  do
4:     Let  $p$  is the strongest proposal in round  $r'$ 
5:     If  $p$  does not exist, skip this round
6:     Commit each proposal in the graph of  $p$ 

```

FIGURE 4.11. Commit the strongest proposal SP_r in round r .

4.2.9. Calculation of the Priorities. For a given set of proposals $P = \{p_1, p_2, \dots, p_n\}$ in round r received from MP , we will choose one of the proposals $SP_r \in P$ as the strongest proposal. The selection contains two parts: the hash value $Hash(p_i)$ and if the proposal contains the proposal with some status marked in the consensus (Section 4.2.6) in round $r - 1$. For any proposal that includes the proposal with some status in round $r - 1$, the smaller the hash value it provides, the higher priority the proposal obtains.

$$(4.1) \quad SP_r = \arg \min_{sp \in P} Hash(sp) \text{ and } ContainSP(sp)$$

$$(4.2) \quad SP_r = \arg \min_{sp \in P} |sp.sourceID - coinValue| \text{ and } ContainSP(sp)$$

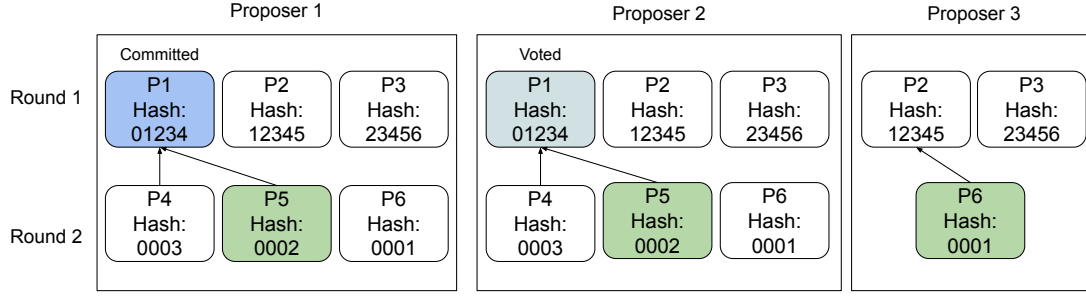


FIGURE 4.12. Example to choose the strongest proposal among three proposers in CFT. In round 1, P_1 has been chosen as the global strongest proposal SP_1 in round 1. However, only proposer 1 marked P_1 as *Committed* while proposer 2 marked P_1 as *Voted* as proposer 2 has not received the commit message and proposer 3 does not receive P_1 . In round 2, proposers 1 and 2 will select P_5 as the local strongest proposal, although they have received P_6 from proposer 3 with a smaller hash value. Proposer 3 will choose P_6 as the local strongest proposal since it does not contain any proposal with status in round 1.

where $ContainSP(sp)$ is the hash of the proposal in round $r - 1$ not marked as None is included SP_r or no such proposal. Figure 4.12 shows an example with three proposers. In round 1, the proposal SP_1 with Hash 01234 has been committed but it is only marked as Voted within proposer 2 due to the message delay while proposer 3 does not receive the proposal. In round 2, proposer 1 and proposer 2 will not count the proposal with hash 0001 from proposer 3 as it does not include the proposal SP_1 in the previous round. Thus they choose the proposal with hash 0002 as the strongest one. However, since proposer 3 does not contain the proposal SP_1 , it can choose any proposal according to their hash values. Thus, the proposal with 0001 will be selected. Consequently, if two proposers receive the same set of proposals in round r , they will vote for the same local strongest proposal SP_r . However, if two proposers generate disparate graphs due to receiving different sets of proposals, some proposals could not arrive within the time T , they may vote for different local strongest proposals in that round.

4.3. Achieving Partial Progress: The Power of Proposer Exchange and Rapid Convergence

It is common for replicas to crash or get network partitioned [206]. These failures not only impact a system's responsiveness to its clients but also have the potential of making the system stuck.

However, CASSANDRA will keep proposing proposals in rounds by connecting the proposals in the previous round to be extended, even though no quorum of replicas is formed.

This partial progress allows replicas to recover in a fast way as once the network is recovered, replicas only need a smaller number of messages to exchange the proposals to help them catch up with each other.

Suppose two groups of replicas are isolated from each other, and the number of replicas in each group is N_1 and N_2 . The number of transactions needed to be exchanged is $N_1 + N_2$ which is less than the protocol without any partial process which needs $(N_1 + N_2)^2$ messages. This is because replicas will share their DAGs in the new round and the replicas in the first group can ask any of the replicas in the second to fetch all the content of the missing transactions. For instance, Figure 4.8 shows two groups that replicas R_1 and R_2 are partitioned from R_3 and R_4 . In the first 3 rounds, transactions in these two partitions are only processed internally and no strongest proposals are committed as could not receive enough referring proposals. In round 4, the network is recovered, and R_2 receives all other replicas. Next, all the proposers select $P(2, 4)$ as the strongest one. Thus, in round 5, all the replicas will commit $P(2, 4)$ that will cover the same DAGs. Since the graph included by the proposals only contains the digests, R_2 needs to query the contents from other replicas. However, R_2 only needs to access R_3 or R_4 since both R_3 and R_4 maintain the contents.

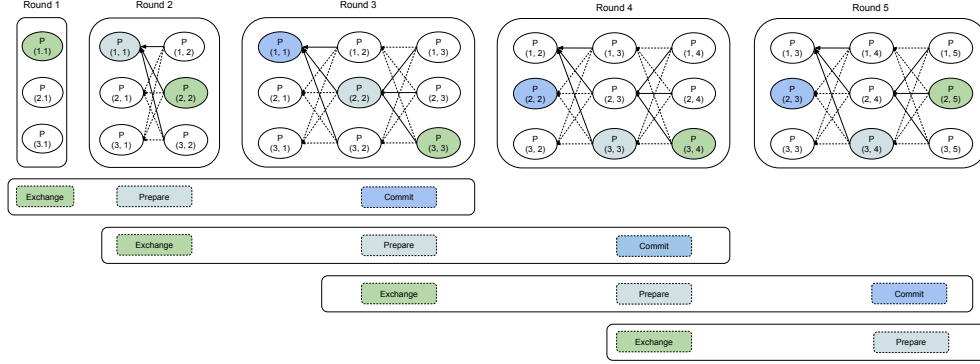


FIGURE 4.13. Chain CASSANDRA Overview.

4.4. Correctness Analysis: Safety and Liveness Guarantees Under Asynchrony

In this section, we provide the evidence that CASSANDRA guarantees the correctness of the system by analysing the safety and liveness of the system.

4.4.1. Safety. Now in this section, we will provide the proofs that the CASSANDRA is safe. We say a system is safe if every honest replica obtains the same state in a specific round.

LEMMA 6. *If a local strongest proposal SP_{r_1} has been committed as the global strongest proposal in round r_1 , at least $f + 1$ replicas contain SP_{r_1} with status Voted or Committed in CFT and at least $2f + 1$ replicas contain SP_{r_1} with status Prepared or Committed in BFT.*

LEMMA 7. *If a local strongest proposal SP_{r_1} has been committed as the global strongest proposal in round r_1 , and a replica R commits a local strongest proposal SP_{r_2} in round r_2 ($r_1 < r_2$), SP_{r_1} will be committed in R as well.*

PROOF. In CFT, since R commits SP_{r_2} in round r_2 , at least $f + 1$ proposers vote for SP_{r_2} in round r_2 . Among these proposers, at least one proposer contains SP_{r_1} before voting SP_{r_2} (Lemma 8). Thus, while receiving $f + 1$ votes from different proposers, at least one of the votes contains SP_{r_1} in its history. After merging the graph, the proposers broadcasting the commit messages will include SP_{r_1} in their graphs. Therefore, at least $f + 1$ replicas contain SP_{r_1} with status *Voted* or *Committed*. Finally, SP_{r_1} will be committed once receiving $f + 1$ commit messages from SP_{r_2} including SP_{r_1} in their graphs with *Voted*.

Similarly, in BFT, while R commits SP_{r_2} in round r_2 , at least $2f + 1$ proposers vote for SP_{r_2} in round r_2 , of which at least one proposer contains SP_{r_1} before voting SP_{r_2} (Lemma 8). Thus, after merging the graph from $2f + 1$ messages, the proposers broadcasting the commit messages will include SP_{r_1} in their graphs. Therefore, at least $2f + 1$ replicas contain SP_{r_1} with status *Prepared* or *Committed*. Finally, SP_{r_1} will be marked as *Committed* once receiving $2f + 1$ commit messages of SP_{r_2} . $\square$

LEMMA 8. *Only one local strongest proposal SP_r will be committed as the global strongest proposal in round r .*

PROOF. If there are two different local strongest proposals being committed from two replicas, there should be at least one honest replica vote for both of these proposals. Thus, it is impossible that two different local strongest proposals are committed in the same round. $\square$

4.4.2. Liveness. Next we will discuss the liveness in CASSANDRA.

LEMMA 9. *If the network is healthy and all the replicas can communicate with each other, a global strongest proposal will be chosen.*

PROOF. If the network is healthy, all the replicas will obtain the set of proposals and generate the same graph. Thus, they will choose the same local strongest proposal and finally this proposal will be committed as the global strongest proposal. $\square$

LEMMA 10. *If a proposal proposed by a proposer will be eventually committed, the system is alive.*

Process transactions in round r by replica R_i :

- 1: Let G be the local graph in R
- 2: Let MP be the set containing a majority of the proposers.
- 3: Let S be the status containing a majority of the proposers.
- 4: **event** Receive a transaction T from the client in round r **do**
- 5: Generate a proposal p by build a graph connecting all the proposals in G with the proposal status.
- 6: Broadcast p and wait for the proposals P from MP or time out.
- 7: **for** Each proposal p' in P **do**
- 8: Merge p' with G
- 9: Select the proposal SP_r with highest priority from the proposal in round r in $resp$
- 10: Update the state of SP_r as *Vote*
- 11: **for** Each proposal p_{r-1} in round $r - 1$ in $resp$ **do**
- 12: **if** The state of p_{r-1} is *Vote* and $resp$ contains $f + 1$ same proposals p_{r-1} **then**
- 13: Mark p_{r-1} as *Commit*
- 14: Commit p_{r-1}

FIGURE 4.14. Chain CASSANDRA for CFT.

Process transactions in round r by replica R_i :

- 1: Let G be the local graph in R
- 2: Let MP be the set containing a majority of the proposers.
- 3: Let S be the status containing a majority of the proposers.
- 4: **event** Receive a transaction T from the client in round r **do**
- 5: Generate a proposal p by build a graph connecting all the proposals in G with the proposal status.
- 6: Broadcast p and wait for the proposals P from MP or time out.
- 7: **for** Each proposal p' in P **do**
- 8: Merge p' with G
- 9: Select the proposal SP_r with highest priority from the proposal in round r in $resp$
- 10: Update the state of SP_r as *Vote*
- 11: **for** Each proposal p_{r-1} in round $r - 1$ in $resp$ **do**
- 12: **if** The state of p_{r-1} is *Vote* and $resp$ contains $f + 1$ same proposals p_{r-1} **then**
- 13: Mark p_{r-1} as *Prepare*
- 14: **for** Each proposal p_{r-2} in round $r - 2$ in $resp$ **do**
- 15: **if** The state of p_{r-1} is *Prepare* and $resp$ contains $f + 1$ same proposals p_{r-1} **then**
- 16: Mark p_{r-1} as *Commit*
- 17: Commit p_{r-1}

FIGURE 4.15. Chain CASSANDRA for BFT.

4.5. Process Cassandra in Chain

As discussed in previous sections, CASSANDRA leverages the proposal exchange phase to share their proposals and select the strongest one as the leader proposal, then leverages the consensus to agree on this leader proposal. However, CASSANDRA still needs to process the history to determine the proposal status in the previous rounds. Moreover, each of the consensus phases involves all-to-all broadcasting communications.

Thus, we create a chain CASSANDRA which combines all the all-to-all broadcasting communications that pipelines the phases in one step. In each pipeline phase, we can also determine the proposal status in previous rounds.

4.5.1. ProposalExchange. Fast CASSANDRA has the same proposal exchange as CASSANDRA. Each proposer shares the proposal in each round and waits for the proposals from others in a bounded time. Once receiving proposals from MP or the timer times out, each proposer selects the strongest proposal in their local graph.

As we pipeline each round by combining each broadcasting communication, we add one more status *Selected* for the proposal in the new round. Once having selected the strongest proposal in round r , mark the proposal as *Selected* then go to next round.

Thus, we will have status $\{Selected, Vote, Commit\}$ for CFT and $\{Selected, Vote, Prepared, Commit\}$ for BFT for each proposal.

Then, while having selected the strongest proposal in round r , we need to update the status in previous rounds. Since we pipeline the steps, we need to check some previous rounds.

4.5.1.1. CFT. If the proposer receives at least $f + 1$ strongest proposal with *Selected* in round $r - 1$, then making it as *Voted*. If the proposer receives at least $f + 1$ strongest proposal with *Voted* in round $r - 2$, then making it as *Committed*. If the strongest proposal in round $r - 2$ has been marked as *Committed*, then commit the proposal.

4.5.1.2. BFT. If the proposer receives at least $2f + 1$ strongest proposal with *Selected* in round $r - 1$, then making it as *Voted*. If the proposer receives at least $2f + 1$ strongest proposal with *Voted* in round $r - 1$, then making it as *Prepared*. If the proposer receives at least $2f + 1$ strongest proposal with *Prepared* in round $r - 2$, then making it as *Committed*. If the strongest proposal in round $r - 2$ has been marked as *Committed*, then commit the proposal where it also becomes the global strongest proposal in round $r - 2$.

4.5.2. ProposalCommit. While committing the globally strongest proposal in round $r - 2$, we also need to recover the global strongest proposal in previous rounds as discussed in Section 4.2.7. Then we commit the global strongest proposal round by round.

4.6. Experimental Evaluation: Performance During and After Network Partitions

In this section, we evaluate the performance of CASSANDRA on both the CFT and BFT versions.

4.6.1. Evaluation on cft Cassandra. We first evaluate CFT CASSANDRA with a multi-paxos [213]. MultiPaxos consensus protocol remains the popular choice among major large-scale production systems such as Chubby [207], Google Spanner [214], Azure Storage [215], and Amazon DynamoDB [216].

MultiPaxos is a well-known optimization from the paxos protocol [24] by reducing unnecessary messages. In MultiPaxos, once the leader is elected, it gains the ability to directly propose values for an unbounded number of instances during the Accept Phase, thus eliminating the need for the repeated Prepare messages. The Prepare Phase only comes into play in the event of a leader crash.

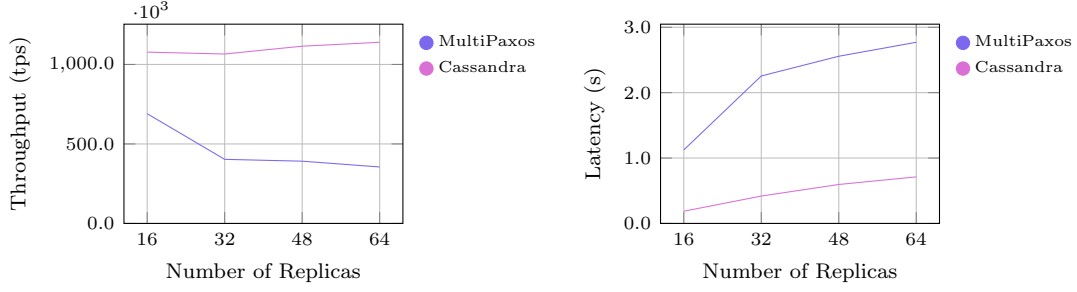


FIGURE 4.16. Performance between CFT CASSANDRA and MultiPaxos protocol

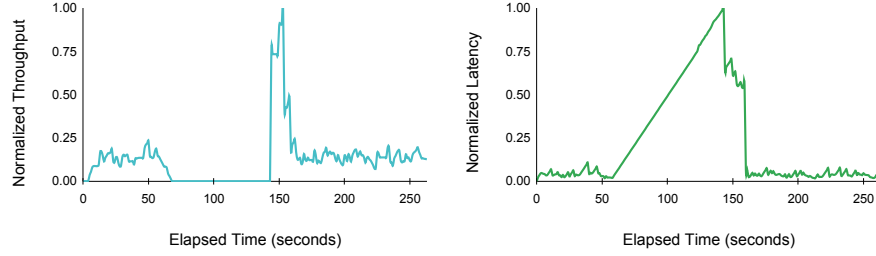


FIGURE 4.17. Impacts of network partition on throughput and latency when partial progress is enabled.

4.6.1.1. *Experiment Setup.* Our experiments were conducted on an AWS c5.4xlarge instance, featuring 16 vCPUs and 32GB of DDR3 memory. Each experiment had a duration of 60 seconds, and the results were then averaged. Additionally, transactions are grouped into batches to optimize network utilization.

4.6.2. **cft Cassandra on Twitter Data.** To evaluate our partial progress conjecture in practice, we deploy our CASSANDRA protocol in the wild over the Twitter data. We aim to capture social interactions such as tweeting (with mentions) and re-tweeting along with quoting, replying, and liking a tweet, all modeled with transactional semantics. More importantly, we further evaluate the impacts of delays introduced by network partitions that may occur due to possible adversarial attacks in order to disrupt the flow of information or by injecting misinformation (i.e., conflicting transactions) into a partitioned social network. Our aim is to study the effects on throughput and latency upon network recovery to demonstrate partial progress as the system heals itself.

Setup and Benchmark. We run our experiments on a local cluster and deploy each replica on a virtual machine having a 4-core CPU and 8 GB memory. We run CASSANDRA on 100 replicated machines, where the primary mode of communication is gossip to exhibit decentralized deployment. We inject 10,000 transactional tweets per second, where each transaction has a payload of under 1KB. The experiments are

run for 260 seconds, where at the 60 seconds mark, the network undergoes partition, the partition lasts for 60 seconds, and the network is restored at 120 seconds. To collect results after reaching a steady state, we discard the measurement during the warmup period, and measurement results are collected over three runs.

Promise of Partial Progress. We evaluate the throughput and latency of the CASSANDRA in a partitioned setting where no partition has the majority of replicas. We observe during the partition, CASSANDRA enters a speculative ordering and execution phase, where transactions are only softly committed and finality is only established once the partitions are recovered. Therefore, once the network is restored, we observe throughput peaks at $6\times$ of the steady state made possible by enabling partial progress. As expected, the finality latency is delayed proportionally to the length of the network outage. The results are shown in Figure 4.17.

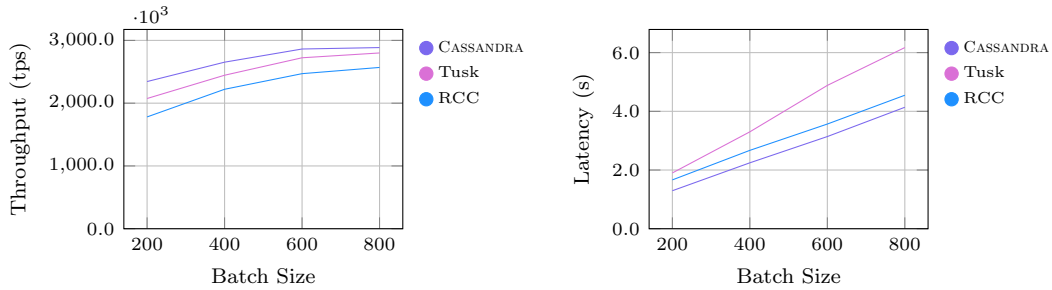


FIGURE 4.18. Evaluation of different batch sizes.

4.6.3. Tolerant Arbitrary Failure Cassandra. In this section, we evaluate the throughputs of CASSANDRA compared to RCC and Tusk. We implemented BFT CASSANDRA as well as RCC and Tusk on top of Apache ResilientDB (incubating), an open-source blockchain platform. We also evaluate the behavior when the cluster is partitioned. In this section, we will conduct an evaluation of the throughputs of CASSANDRA in comparison to RCC [117] and Tusk [26].

4.6.3.1. BaseLine. We compare CASSANDRA with two base line protocols: RCC and Tusk.

RCC. RCC is a multi-leader consensus protocol. RCC runs many instances, each of which runs an individual Byzantine commit algorithm (BCA) to commit transactions. Each instance is also a primary to propose transactions concurrently and does not require any coordination with other instances.

RCC operates in rounds. In each round, RCC proposes transactions in three phases: concurrent BCA, ordering phase and execution phase. The overview of RCC is shown in Figure ?? . In Concurrent BCA, m transactions are proposed by m instances running m lead-base consensus protocols in parallel in each round. These m instances do not need to coordinate with each other. Instance X is allowed to propose

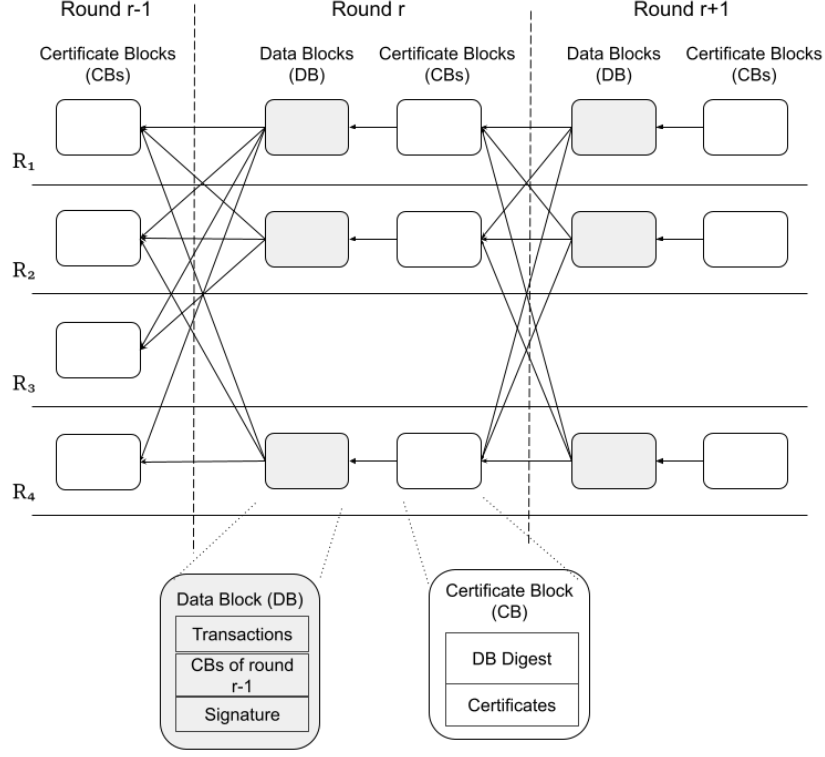


FIGURE 4.19. Overview of Tusk.

transactions for future round $i + 1$ while instance Y is still processing the transactions in round i . The wait-free proposing enables each replica to propose transactions at its maximum throughput by utilizing its entire outgoing network bandwidth. However, in ordering phase, each replica must wait for m transactions in each round to obtain a deterministic order using the same set of transactions and execute the transactions in that order in the Execution Phase.

The throughput of *RCC* is determined by the throughput of each BCA. By default, *RCC* leverages pbft [29] as the BCA which commits a transaction in three phases.

Tusk.

Tusk [26] is a DAG-based consensus protocol. Tusk runs in rounds and contains two types of blocks: Data Block and Certificate Block. In round r , each data block db_i of replica R_i contains at least $2f + 1$ certificate blocks of round $r - 1$ from different replicas. each certificate block cb_i of replica R_i contains the digest of the data block db_i and $2f + 1$ certificates signed from different replicas in round r . Figure 4.19 shows the overview of Tusk.

In round r , replica R_i generates the data block db_i including the new transactions and at least $2f + 1$ certificate blocks of round $r - 1$, then signs db_i and broadcasts to each replica. When a replica R_j receives a data block, R_j verifies the contents and responds R_i with the signatures. R_i generates the certificate block cb_i once receiving $2f + 1$ signatures for db_i and broadcasts cb_i to other replicas. When a replica receives at least $2f + 1$ certificate blocks of round r , it starts generating the data block in round $r + 1$.

Tusk leverages wave to commit blocks. When round r starts, one of the replicas will be selected as the primary P of round $r - 2$. The data block db_p^{r-2} proposed by P and all the blocks included by db_p^{r-2} will be committed. We say if block b_i includes block b_j , b_i can reach b_j . If block b_i can reach block b_j and block b_j can reach block b_k , then block b_i can reach block b_k . Then, all the data blocks in the previous rounds that can be reached by db_p^{r-2} will be committed if db_p^{r-2} is committed in round $r - 2$. Since all the replicas contain the same DAG graph, if two replicas commit the data block db_p^r in round r , they will commit the same blocks that can be reached from db_p^r .

However, since a data block only includes $2f + 1$ certificate blocks in the previous round, if a certificate block arrives after $2f + 1$ certificate blocks, it will be delayed which will increase the latency for the clients. On the other hand, broadcasting the data block will take time since the payload is large. However, before proposing a new data block, the replicas need to wait for the certificate blocks from the previous round, which will delay the data block.

4.6.3.2. Experiment Setup. Our experiments were conducted on an AWS c5.4xlarge instance, featuring 16 vCPUs and 32GB of DDR3 memory. Each experiment had a duration of 60 seconds, and the results were then averaged. Additionally, transactions are grouped into batches to optimize network utilization.

4.6.3.3. Impact of Batch Size. In our initial experiments, we conducted tests using various batch sizes on 64 nodes to determine a suitable base size we will use in the following experiments, and the outcomes are depicted in Figure 4.18. The findings indicate that augmenting the batch size results in increased throughput. However, it also reveals a corresponding increase in latency with larger batch sizes. The majority of protocols achieve peak throughputs, $2.8M$ tps, with a batch size of 600. Consequently, we have chosen to adopt 600 as the default batch size for our forthcoming experiments.

4.6.3.4. Performance Results. Next, we conducted experiments with various replica sizes, ranging from 16 to 64, without any partitions. As depicted in Figure 4.20, it is evident that CASSANDRA achieves comparable throughputs to Tusk and RCC, with all protocols reaching $3M$ transactions per second (TPS). This is because under a healthy and fast network, all proposals from the replicas in each round will be committed simultaneously across the three protocols. It is noteworthy that CASSANDRA exhibits a lower latency of 2

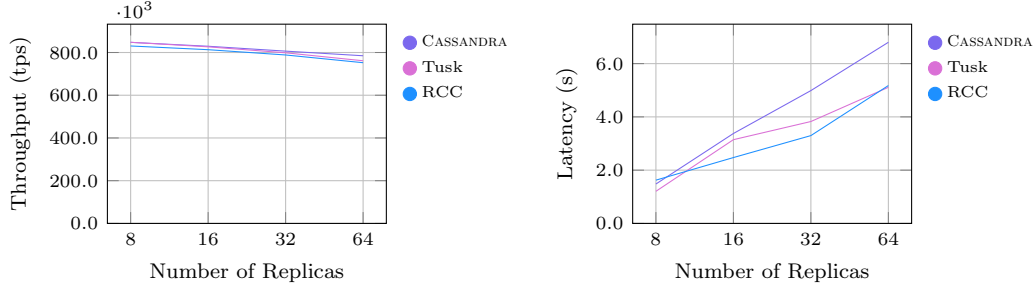


FIGURE 4.20. Evaluation of different protocols.

seconds with 32 replicas, while Tusk achieves 4 seconds, benefiting from the isolated data dissemination on our dissemination network. CASSANDRA demonstrates similar latency as RCC, as RCC benefits from its out-of-order processing, while Cassandra reduces latency by streaming the pipelines.

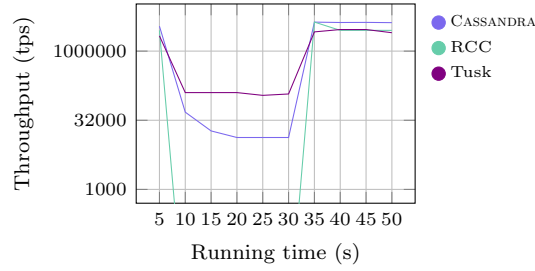


FIGURE 4.21. Recovery Behaviour with two partitions (23,9) of 32 replicas are partitioned during time 15-30.

4.6.3.5. Network Evaluation. Then, we demonstrate the behavior of two partitioned groups. Specifically, we partitioned 32 replicas into two smaller groups, with one containing 23 replicas and the other containing 9 replicas. At time 10, the connections between these two groups were deliberately terminated, and subsequently reestablished at time 30. The throughput timeline (each second) of each group is illustrated in Figure 4.21. A 5-second timer was set in CASSANDRA to wait for all the proposals, allowing the rounds to proceed further if the timer timed out.

It was noted that CASSANDRA and Tusk experienced a decline in throughput from $3M$ to $15K$ and $120K$ TPS respectively due to a decrease in proposed transactions, as the smaller group was unable to commit any transactions. However, RCC diminished to zero due to its global ordering of proposals required by all replicas.

Upon reconnection, both groups were able to recover within a timeframe of 10 seconds and reach a stable throughput of $3M$ TPS. Notably, Tusk exhibited higher throughputs than Cassandra during the partition,

as it required only $2f + 1$ replicas to process in each round, while Cassandra was constrained by waiting for a timeout.

4.7. Related work

Brewer’s CAP theorem [15] argues that while designing an application, the application designer can only select two out of the following three properties: consistency, availability and partition tolerance. Following this, several new models have appeared, which present fresh perspectives for an application designer. PACELC [217] captures the following double trade-off: if partitions occur, then trade between availability and consistency; else trade between latency and consistency. FIT [218] re-imagines the problem of a partitioned system as the problem of fairness and envisions fairness as a metric for latency. CAC [219] redefines consistency as causal consistency and compares it against availability and convergence. BASE [220] proposes a diametrically opposite model to ACID semantics. Unlike pessimistic nature of ACID transactions, BASE vouches for optimistic execution and accepts that the database consistency will be in a state of flux. CALM [221] proponates the ideas proposed by Conflict-replicated data types (CRDTs) [222]. Prior works on designing fault-tolerant database systems include Google’s Spanner [223] and Sequoia [224], which have been adopted in practice. None of these works discuss the possibility of partial progress under CAP model, which is the theme of this chapter, and we meet this aim with the help of our CASSANDRA protocol.

4.7.1. RCC. RCC is a multi-leader consensus protocol. RCC runs many instances, each of which runs an individual Byzantine commit algorithm (BCA) to commit transactions. Each instance is also a primary to propose transactions concurrently and does not require any coordination with other instances.

RCC operates in rounds. In each round, RCC proposes transactions in three phases: concurrent BCA, ordering phase and execution phase. In Concurrent BCA, m transactions are proposed by m instances running m lead-base consensus protocols in parallel in each round. These m instances do not need to coordinate with each other. Instance X is allowed to propose transactions for future round $i+1$ while instance Y is still processing the transactions in round i . The wait-free proposing enables each replica to propose transactions at its maximum throughput by utilizing its entire outgoing network bandwidth. However, in ordering phase, each replica must wait for m transactions in each round to obtain a deterministic order using the same set of transactions and execute the transactions in that order in the Execution Phase.

The throughput of *RCC* is determined by the throughput of each BCA. By default, *RCC* leverages pbft [29] as the BCA which commits a transaction in three phases.

4.7.2. Tusk. Tusk [26] is a DAG-based consensus protocol. Tusk runs in rounds and contains two types of blocks: Data Block and Certificate Block. In round r , each data block db_i of replica R_i contains at least $2f + 1$ certificate blocks of round $r - 1$ from different replicas. Each certificate block cb_i of replica R_i contains the digest of the data block db_i and $2f + 1$ certificates signed from different replicas in round r .

In round r , replica R_i generates the data block db_i including the new transactions and at least $2f + 1$ certificate blocks of round $r - 1$, then signs db_i and broadcasts to each replica. When a replica R_j receives a data block, R_j verifies the contents and responds R_i with the signatures. R_i generates the certificate block cb_i once receiving $2f + 1$ signatures for db_i and broadcasts cb_i to other replicas. When a replica receives at least $2f + 1$ certificate blocks of round r , it starts generating the data block in round $r + 1$.

Tusk leverages wave to commit blocks. When round r starts, one of the replicas will be selected as the primary P of round $r - 2$. The data block db_p^{r-2} proposed by P and all the blocks included by db_p^{r-2} will be committed. We say if block b_i includes block b_j , b_i can reach b_j . If block b_i can reach block b_j and block b_j can reach block b_k , then block b_i can reach block b_k . Then, all the data blocks in the previous rounds that can be reached by db_p^{r-2} will be committed if db_p^{r-2} is committed in round $r - 2$. Since all the replicas contain the same DAG graph, if two replicas commit the data block db_p^r in round r , they will commit the same blocks that can be reached from db_p^r .

However, since a data block only includes $2f + 1$ certificate blocks in the previous round, if a certificate block arrives after $2f + 1$ certificate blocks, it will be delayed which will increase the latency for the clients. On the other hand, broadcasting the data block will take time since the payload is large. However, before proposing a new data block, the replicas need to wait for the certificate blocks from the previous round, which will delay the data block.

4.8. Summary and Discussion

In this chapter, we illustrated that it is possible to design a system that is consistent, available, and makes partial progress under network partitions. We claim that this partial progress can help an application be responsive to a subset of its clients and allow it to achieve non-zero throughput. To prove our claim, we present the design of our novel CASSANDRA protocol that allows replicas to continue ordering client transactions even under failures. CASSANDRA achieves this goal by requiring each replica to vote for the strongest proposal among all the proposals. Selecting the strongest proposal requires comparing the rank of a proposal and observing the suffix of its log. Our experiments demonstrate that CASSANDRA works in the

wild and the partial progress during network partition helps to yield high throughputs once the network is restored.

Chapter 5: ResilientDB – A High-Throughput Fabric for Validating and Comparing BFT Protocols

5.1. Introduction: From Protocol Design to a Practical System

The preceding chapters have presented novel consensus and execution protocols—PoC, Thunderbolt, and CASSANDRA—each designed to address specific challenges in Byzantine fault-tolerant systems. However, the practical validation of such protocols requires more than theoretical analysis or isolated simulations. Researchers need a unified, high-performance platform that allows fair comparison, flexible integration of new protocols, and systematic evaluation under realistic conditions.

The preceding chapters have introduced three novel consensus and execution protocols—PoC, Thunderbolt, and CASSANDRA—each designed to address critical challenges in Byzantine fault-tolerant systems. PoC (Chapter 2) enhances the security of existing PoS/BFT systems against long-range attacks through collaborative mining. Thunderbolt (Chapter 3) improves smart contract execution scalability by integrating Order-Execute and Execute-Order-Validate models within a sharded DAG architecture. CASSANDRA (Chapter 4) ensures system availability during network partitions through a leaderless protocol that achieves partial progress.

However, the rigorous validation of such protocols demands more than theoretical analysis or isolated simulations. Researchers require a unified, high-performance platform that enables fair comparison across different consensus algorithms, supports flexible integration of new protocols, and provides systematic evaluation capabilities under realistic deployment conditions. This need is particularly acute in the blockchain research community, where inconsistent experimental setups and ad-hoc implementations often make it difficult to compare results across different studies.

ResilientDB was designed to address this gap. Founded in 2018 at UC Davis [212], ResilientDB was open-sourced in late 2019. Following a complete re-architecture in 2022, the project entered the Apache Incubation process in October 2023, establishing itself as a community-driven, open-source platform.

ResilientDB provides a modular, high-throughput permissioned blockchain fabric that serves as a common testbed for evaluating BFT protocols. Its system-centric architecture separates transaction ordering from execution, enabling deep pipelines and out-of-order processing. This design allows even slower consensus protocols—when deployed on a well-optimized system—to outperform less-optimized platforms running faster algorithms. This observation underscores a central theme of this dissertation: system-level optimizations are as critical to blockchain performance as protocol-level innovations.

This chapter presents ResilientDB as both a robust blockchain platform and, more importantly, a flexible research infrastructure. We describe its layered architecture, the key design decisions that underpin its performance, and its capabilities as a benchmarking framework. We demonstrate how researchers can implement their own consensus protocols atop ResilientDB without modifying underlying system components, ensuring fair and reproducible comparisons. Finally, we present performance benchmarks that characterize the platform’s capabilities and illustrate its utility for protocol evaluation.

5.2. System Architecture: A Modular Platform for Performance and Flexibility

The architecture of ResilientDB is composed of four principal layers, each designed with clear separation of concerns and optimized for high throughput.

Network Layer. The Network Layer, which manages all communication between replicas and clients, is implemented through two cooperating components: the RDBC Driver and the Network Substrate. The RDBC Driver (ResilientDB Database Connectivity) handles connections from client SDKs using a coroutine-based model built upon the Boost Asio library[???]. This design enables a single thread to manage thousands of concurrent connections with minimal context-switching overhead. When a worker coroutine receives a message, it places it into an input queue, from which it will later be consumed by the consensus manager. Decoupling message reception from processing ensures that the network layer continues to operate even when consensus processing is temporarily backlogged. The Network Substrate, on the other hand, manages replica-to-replica communication. It applies several key optimizations. First, batch processing aggregates multiple messages destined for the same replica into a single transmission, reducing the number of network round trips. Second, copy-on-write broadcasting ensures that when a message must be sent to all replicas, only a single copy is maintained in memory and referenced by each recipient, drastically reducing memory usage and copying overhead. Third, asynchronous delivery via coroutines permits non-blocking message transmission, so that a slow network operation never stalls other processing tasks. Together, these design



FIGURE 5.1. ResilientDB Architecture.

choices reduce CPU and memory overhead and allow ResilientDB to sustain high message rates under heavy load.

Consensus Manager. The Consensus Manager is responsible for establishing a global order of client transactions. Its most critical design property is that it is protocol-agnostic: any BFT consensus protocol can be integrated through a clean, minimal interface. That interface exposes only three essential operations: submit a batch of transactions for ordering, cast a vote for a proposed block, and finalize a block once consensus is reached. Researchers can implement protocols such as PBFT[?], HotStuff[?], RCC[?], or PoC[?] entirely within this interface, without modifying network, storage, or execution components. This isolation guarantees that performance differences observed between protocols reflect algorithmic choices rather than artifacts of implementation. Internally, the consensus manager deploys a pool of worker threads operating in a multi-threaded framework. Workers follow a pull strategy, obtaining requests from the queue maintained

by the RDBC Driver, and they sleep on condition variables when the queue is empty to avoid wasteful CPU spinning. As of the time of writing, ResilientDB has successfully integrated PBFT [29], PoE [118], PoC [225], HotStuff [11], RCC [117], GeoBFT [135], RingBFT [176], SpotLess [198], and several DAG-based protocols such as Tusk and Bullshark [26]. Each integration was accomplished without requiring any changes to other system components, demonstrating the effectiveness of the modular design.

Transaction Manager. Once transactions are totally ordered by the consensus manager, the Transaction Manager executes them against the application state. By default, execution follows the sequence determined by consensus, ensuring deterministic outcomes across all honest replicas. However, researchers can alternatively plug in their own execution engines to explore different concurrency control techniques. The transaction manager natively supports three execution models. The first is a high-performance Key-Value Store. The second model is Smart Contract Execution, which includes a built-in service supporting Solidity contracts that are compiled to EVM bytecode and executed on an embedded Ethereum Virtual Machine. Contract deployment and invocation can be performed through low-level JSON-based tools, the ResContract CLI, the ResVault wallet, or GraphQL services. The third model is the UTXO Model, intended for cryptocurrency-style applications, which supports transaction chaining and balance verification. Architecturally, the transaction manager consists of a set of background workers that execute transactions in parallel whenever dependencies permit. Because transaction ordering and execution are fully decoupled, replicas can continue ordering new transactions while previously ordered ones are still being executed, forming a deep pipeline that overlaps communication, computation, and I/O.

Chain State and Storage Manager. The Chain State and Storage Manager is the component that persists blockchain data, chain state, and metadata. It uses LevelDB [226] as the default storage engine, with RocksDB [227] available as an alternative. The chain state maintains the current world state after all committed transactions have been executed—for key-value applications, this is the latest value of each key; for smart contracts, it includes account balances and contract storage. The storage manager retains the complete transaction history, enabling audit, recovery, and re-execution. To support fast crash recovery, a restarting replica first loads the most recent checkpoint from the chain state, then replays any committed transactions that were executed after that checkpoint. This approach balances recovery speed with durability.

5.3. Ecosystem and Developer Tooling

Beyond the core node, ResilientDB ships with a comprehensive ecosystem of tools Table 5.1 that lower the barrier to application development and system evaluation.

Component	Description
GraphQL & REST APIs	HTTP services for committing and querying transactions, enabling easy integration with web backends and AI workflows.
SDKs	Python ORM (resdb-orm), Rust SDK, and ResVault SDK provide higher-level abstractions for building applications.
ResVault	Browser extension wallet for key management, user authentication, and transaction signing.
ResLens	Monitoring UI and middleware for visualizing cluster health, performance metrics, and profiling data.
Grafana Dashboards	Pre-configured dashboards for real-time performance monitoring.
Deployment Tools	Ansible playbooks, Docker containers, and Orbit desktop deployer for rapid cluster setup on local or cloud environments.
ResContract CLI	Command-line interface for creating accounts, deploying contracts, and invoking contract methods.

TABLE 5.1. Development Tools in ResilientDB

GraphQL and REST APIs serve as the primary HTTP interfaces for committing and querying transactions. The GraphQL endpoint allows clients to request precisely the data they need, avoiding over-fetching, while the REST API provides compatibility with conventional web backends. Together, these services facilitate integration with web applications, data services, and AI workflows without requiring developers to understand the internal consensus mechanisms.

A set of Software Development Kits (SDKs) provides higher-level abstractions for building applications. The Python ORM (resdb-orm) allows developers to model blockchain assets as Python classes and interact with ResilientDB through either GraphQL or REST interfaces. The Rust SDK targets performance-sensitive scenarios with memory safety and concurrency guarantees. The ResVault SDK is purpose-built for integration with the ResVault browser extension wallet. These language-specific libraries drastically reduce the effort required to build decentralized applications on top of ResilientDB.

ResVault itself is a browser extension wallet responsible for key management, user authentication, and transaction signing. It binds on-chain write operations to user-controlled identities and is an essential component for end-user-facing decentralized applications.

Monitoring and observability are covered by ResLens and pre-configured Grafana Dashboards. ResLens is a monitoring UI and middleware that visualizes cluster health, performance metrics, and profiling data. Grafana dashboards provide ready-to-use panels for real-time throughput, latency, and resource utilization monitoring, enabling operators to assess system behavior at a glance.

A variety of Deployment Tools enable rapid cluster setup. Ansible playbooks automate the configuration of ResilientDB nodes on bare-metal or cloud instances. Docker containers guarantee environment consistency across development, testing, and production. The Orbit desktop deployer offers a graphical interface for local or cloud deployment, further lowering the technical barrier.

For smart contract developers, the ResContract CLI provides a command-line interface to create accounts, deploy contracts, and invoke contract methods. These tools collectively form a complete pathway from user-facing applications down to the consensus layer, enabling researchers to deploy a ResilientDB cluster, submit transactions, and collect performance data with minimal setup overhead—capabilities that are essential for systematic protocol evaluation.

5.4. Key Design Decisions: System-Centric Optimizations

ResilientDB’s high performance stems from a set of foundational design principles that prioritize system-level efficiency over protocol-specific micro-optimizations.

Separation of Ordering from Execution. Traditional blockchain platforms often tightly couple transaction ordering and execution, causing the two phases to block each other. ResilientDB decouples them so that they proceed independently and asynchronously. Replicas can continue ordering new transactions while previously ordered ones are still executing, forming a deep pipeline where communication, computation, and I/O overlap. Moreover, the concurrency levels of ordering and execution can be tuned independently to match workload characteristics. This design was directly motivated by the observation that a well-optimized system running a slower consensus protocol can outperform a less-optimized system running a faster one.

Multi-Threaded Architecture with Coroutines. ResilientDB combines the concurrency benefits of asynchronous I/O with the simplicity of synchronous programming. Each thread runs multiple coroutines that yield control only when waiting for I/O, minimizing context-switch overhead. This allows a single thread to handle thousands of concurrent connections with far lower memory consumption than a thread-per-connection model. The code written in this style resembles synchronous code, which simplifies debugging and maintenance. Within the consensus and transaction managers, worker pools are employed with threads pinned to specific cores to maximize cache locality and reduce lock contention, while queue-based task distribution ensures load balancing.

Batched Message Processing. Network communication is often the primary bottleneck in distributed systems, and ResilientDB mitigates this through multiple complementary techniques. Messages destined for the same replica are aggregated into batches before transmission to reduce round trips. Copy-on-write broadcasting maintains a single in-memory message copy referenced by all recipients. Additionally, message payloads can be compressed before transmission, with configurable thresholds that allow operators to trade off CPU cycles for network bandwidth.

Modular Consensus Interface. The consensus manager’s protocol-agnostic interface is central to ResilientDB’s value as a research platform. The interface exposes only Propose, Vote, and Commit, giving implementers maximum flexibility. Because protocol implementations do not interact directly with the network, storage, or execution components, performance comparisons are fair: the underlying system remains constant, and only the consensus logic changes. To further ease development, the consensus manager provides

built-in utilities for common tasks such as leader election, quorum management, certificate storage, and timeout handling. Implementers may use these utilities or provide their own.

Support for Multiple Execution Models. By supporting key-value, smart contract, and UTXO models through a common execution interface, ResilientDB enables workload diversity without forcing researchers to reimplement consensus logic. All protocols are tested on identical execution engines, eliminating variability from execution-layer differences. New execution models can be added by implementing the execution interface, without any modification to the consensus or network layers.

5.5. Performance Benchmarking: Evaluating the Fabric’s Capabilities

ResilientDB includes a comprehensive benchmarking suite located under the `benchmark/` directory. The suite supports YCSB-style workloads with configurable read/write ratios, request distributions (uniform, Zipfian), and operation mixes; the SmallBank workload, which simulates banking transactions such as account transfers, balance queries, deposits, and withdrawals; and custom workloads that researchers can define by implementing a simple client interface.

All experiments reported in the dissertation were conducted on AWS c5.9xlarge instances (36 vCPUs, 72 GiB memory). A uniform experiment protocol was followed: a 60-second warm-up period was discarded from the results, measurement periods lasted 60 to 120 seconds depending on the experiment, and each experiment was repeated five times with results averaged. Unless otherwise noted, the client model was closed-loop, meaning a client waits for a response before sending its next transaction.

The performance of several consensus protocols implemented on ResilientDB is presented in Figure 5.2, with a batch size of 400 and replica counts ranging from 16 to 128. PoE outperforms PBFT because it eliminates one communication round, illustrating the impact of protocol-level reductions in message complexity. HotStuff exhibits lower throughput due to its three-phase commit latency, which limits how many rounds can execute in parallel. This aligns with the dissertation’s broader thesis: protocol speed alone does not determine system performance. All protocols degrade gracefully as the number of replicas increases, due to the quadratic growth in communication. Critically, ResilientDB’s PBFT implementation outperforms other permissioned blockchains that rely on faster consensus protocols (such as Zyzzyva), directly validating the system-centric design philosophy that a slower protocol on a well-optimized platform can surpass a faster protocol on a less-optimized one.

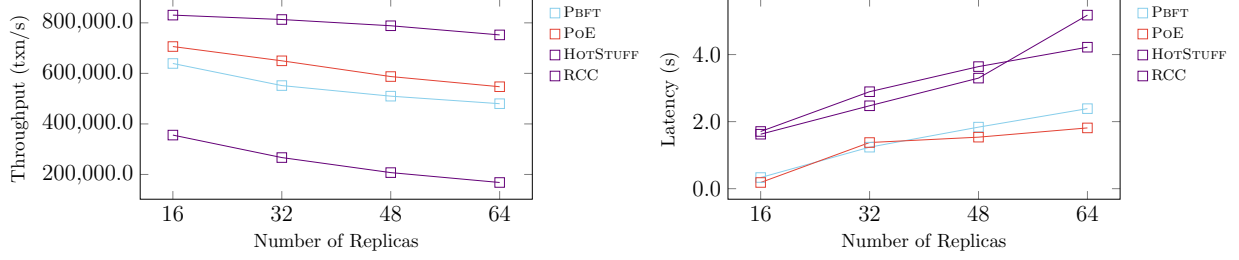


FIGURE 5.2. Peak throughput and latency of different consensus protocols.

5.6. Summary: ResilientDB as a Research and Development Platform

ResilientDB fulfills two essential roles within this dissertation. It serves as the validation platform for the protocols presented in preceding chapters: PoC was integrated as a consensus protocol module, Thunderbolt leveraged the concurrent executor and DAG integration, and CASSANDRA extended the consensus manager with leaderless semantics. At the same time, it constitutes a flexible research infrastructure that enables fair, reproducible comparison of BFT protocols through its modular architecture, high-performance core, and comprehensive benchmarking tooling.

The key contributions of ResilientDB can be summarized as follows. First, its system-centric design demonstrates that thoughtful system optimization can yield performance gains that exceed those achievable through protocol-level improvements alone. Second, its protocol-agnostic architecture allows new BFT protocols to be implemented and evaluated without modifying any underlying infrastructure, ensuring that performance comparisons are both fair and reproducible. Third, its rich ecosystem of tools—including GraphQL APIs, multiple SDKs, monitoring dashboards, and deployment tools—lowers adoption barriers for researchers and practitioners alike. Fourth, its open-source availability under the Apache Incubator ensures free access for academic and industrial use, with a community-driven development model that guarantees ongoing maintenance and feature evolution. Fifth, its support for multiple execution models—key-value, smart contract, and UTXO—enables workload diversity and extensibility.

Several limitations remain. Performance degrades significantly under high network latency, and protocols such as GeoBFT require further optimization. EVM execution remains a bottleneck for contract-intensive workloads; future versions could explore WebAssembly-based execution engines. Support for adding or removing replicas without service interruption is not yet complete. Expanding SDK support to additional languages such as Java and Go would also broaden the platform’s accessibility.

ResilientDB stands as a testament to the value of system-level thinking in blockchain research—a theme that runs throughout this dissertation. By providing a common, high-performance platform for protocol development and evaluation, it enables the kind of rigorous, reproducible research that advances the state of the art. Its ongoing evolution under the Apache Incubation process ensures that it will continue to serve the research community and industry for years to come.

Future Work – Leveraging RDMA for Scalable Transaction Execution

The preceding chapters have advanced the state of blockchain systems in security (PoC), scalability (Thunderbolt), and availability under asynchrony (CASSANDRA), culminating in a high-performance testbed (ResilientDB). Nevertheless, as blockchain networks continue to grow, the volume of on-chain data and the complexity of transaction execution pose new challenges. One promising direction to address these challenges is the integration of Remote Direct Memory Access (RDMA) technology. This chapter explores how RDMA can be harnessed to overcome memory bottlenecks and enable horizontal scaling of transaction execution in permissioned and permissionless blockchains.

6.1. Residual Bottlenecks in the Current Fabric

The preceding chapters have advanced the state of blockchain systems in security (PoC), scalability (Thunderbolt), availability (CASSANDRA), and system performance (ResilientDB). However, several fundamental bottlenecks remain that cannot be fully eliminated by protocol-level or system-level optimizations alone. These bottlenecks motivate the exploration of hardware-assisted networking technologies such as Remote Direct Memory Access (RDMA).

Communication overhead in consensus. Despite the optimizations in ResilientDB – batched message processing, copy-on-write broadcasting, and a multi-threaded coroutine architecture – the underlying TCP/IP stack still imposes significant CPU overhead. Each message transmission involves kernel crossings, context switches, and multiple memory copies. In a BFT system with, say, 128 replicas, each consensus round generates thousands of messages. For small transactions, the CPU cost of processing these messages often dominates the end-to-end latency. This overhead is inherent to the traditional network stack; eliminating it requires a kernel-bypass approach.

Memory bottleneck in stateful execution. Thunderbolt’s sharded execution reduces contention among transactions but does not reduce the total state size that each replica must store locally. As the ledger grows over time – an Ethereum-compatible state can exceed 500GB – replicas eventually resort to

disk accesses. Even with SSDs, random access to disk-resident Merkle tree nodes incurs latencies tens to hundreds of times higher than memory access. This memory wall limits the scalability of Thunderbolt in long-running deployments, as the working set of hot keys may exceed available RAM, causing frequent page faults.

State replication overhead across replicas. In a permissioned BFT system, fault tolerance requires each replica to maintain a full copy of the state. While this redundancy ensures resilience, it multiplies the total memory requirements by the number of replicas. For a state of several hundred gigabytes, deploying 16 or 32 replicas becomes prohibitively expensive in terms of memory hardware and energy consumption. This cost is particularly acute in geo-distributed deployments, where replicas are spread across multiple data centers.

These three bottlenecks – communication, memory capacity, and replication cost – represent the next frontier for improving permissioned blockchain performance. The remainder of this chapter examines whether RDMA can address them.

6.2. Overview of Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a networking technology that allows data to be transferred directly between the memory of two computers without involving the operating system, CPU caches, or buffers of either machine. Traditional network communication (e.g., TCP/IP) requires data to be copied from application memory to kernel buffers, then to network interface buffers, and the reverse process on the receiving side. These multiple copies and context switches introduce significant latency and consume CPU cycles.

RDMA bypasses the kernel entirely. Key features include:

- **Kernel bypass:** Data is transferred directly from the memory of one application to the memory of another, with the network interface controller (NIC) handling the transfer.
- **Zero-copy:** No intermediate buffer copies are required, reducing memory bandwidth consumption and latency.
- **CPU offload:** The remote CPU is not involved in the data movement; it only receives a completion notification. This frees CPU cores for other computation.
- **Low latency and high throughput:** RDMA achieves microsecond-scale latencies (as low as 1–2 μ s for small messages) and line-rate throughput (e.g., 100 Gb/s or higher).

RDMA is typically implemented over InfiniBand or RoCE (RDMA over Converged Ethernet). InfiniBand [228] is a dedicated high-speed interconnect common in high-performance computing (HPC) clusters, while RoCE [229] allows RDMA to run on standard Ethernet networks with lossless configuration. Today, RDMA is widely available in data centers through cloud providers (e.g., AWS EFA, Azure InfiniBand) and commodity NICs that support RoCE.

6.3. Benefits of RDMA for Blockchain Consensus and Execution

RDMA directly addresses the three bottlenecks identified in Section 6.1. This section details the specific performance benefits.

Overcoming communication overhead. By enabling kernel bypass and zero-copy transfers, RDMA can reduce the per-message latency from tens of microseconds (typical for TCP/IP) to 1–2 μ s for small messages. In a BFT protocol like PBFT or HotStuff, each consensus round involves multiple all-to-all message exchanges. Replacing TCP with RDMA has been shown to cut consensus latency by 50–80% [230] and increase throughput by 3–5 $\times$, while freeing CPU cores for other tasks. This directly eliminates the first bottleneck.

Eliminating the memory bottleneck via remote state access. RDMA supports one-sided reads and writes: a replica can fetch a key’s value or a Merkle tree node directly from a remote memory pool without interrupting the remote CPU. This enables a decoupled state architecture: the full state resides on a cluster of memory-rich storage nodes, while consensus replicas only keep a hot cache. Accesses that miss the cache are served via RDMA reads with microsecond latency – orders of magnitude faster than disk I/O. Consequently, the total state can grow far beyond the memory capacity of any single replica, and the working set no longer needs to fit entirely in RAM.

Reducing state replication overhead. Because RDMA allows replicas to access a shared remote state, the system no longer requires every replica to store a full copy of the state. Fault tolerance can be achieved by replicating state across a smaller set of storage nodes (using RDMA-based replication or erasure coding) rather than forcing each consensus replica to hold a complete copy. This reduces total memory requirements by a factor roughly equal to the number of consensus replicas, making large-scale deployments more cost-effective. Additionally, crash recovery becomes faster: a failed replica can directly pull the latest state via bulk RDMA transfers at line-rate throughput (e.g., 10Gb/s), reducing recovery time from minutes to seconds.

6.4. Enabling Scalable Execution via Remote Memory Access

This section outlines a concrete architecture for integrating RDMA into a permissioned blockchain fabric, extending the design of ResilientDB and Thunderbolt. We also discuss the key research challenges that remain to be solved.

Proposed architecture: disaggregated state with RDMA. The system consists of two types of nodes:

- Consensus replicas: run the BFT protocol (e.g., PBFT, HotStuff, or CASSANDRA) and execute transactions. They maintain a local cache of frequently accessed state but do not store the full state.
- Storage nodes: a cluster of memory-rich nodes that collectively store the entire state (e.g., key-value pairs, Merkle tree nodes). Each storage node exposes its memory regions for one-sided RDMA read and write access.

When a transaction executes, the replica issues RDMA reads to fetch required keys from storage nodes. Write operations are buffered locally and then applied via RDMA writes, possibly with atomic compare-and-swap (CAS) for single-location updates. Consensus ordering ensures that concurrent writes to the same key are serialized.

Research challenges. While RDMA offers clear benefits, its adoption in Byzantine environments raises several open problems:

- Atomicity across multiple remote writes: RDMA CAS only works on a single memory location. For transactions that update multiple keys, the system must ensure that a replica never observes a partially updated state. One solution is to use a distributed logging protocol over RDMA, but this adds latency.
- Crash consistency and recovery: If a storage node fails, its memory contents are lost. The system must replicate state across storage nodes (e.g., using RDMA-based replication or erasure coding) and provide crash-consistent checkpoints. Recovery protocols must avoid long blocking periods.
- Byzantine resilience of RDMA operations: A malicious replica could issue malformed RDMA writes to corrupt storage node memory. Protecting against this requires cryptographic integrity verification, but verifying every remote access may negate RDMA's performance gains. Lightweight, Byzantine-resilient RDMA protection mechanisms are an open research direction.

- Network partition handling: RDMA relies on lossless networks (e.g., InfiniBand, RoCE). Under partition, RDMA connections may time out. CASSANDRA’s leaderless partial-progress approach could be adapted, but RDMA’s lack of fine-grained failure detection makes this non-trivial.

These challenges are not insurmountable, but they require careful design. Existing RDMA-based systems (FaRM [230], FaSST [231]) are designed for crash-fault environments; extending them to tolerate Byzantine faults is the core contribution that future work must address.

6.5. Concluding Remarks on Future Directions

RDMA technology offers a compelling path to overcome the memory bottlenecks that constrain blockchain transaction execution. By enabling remote, one-sided access to state data, RDMA allows the state to be horizontally scaled across a cluster of memory-rich nodes, decoupling state size from the memory capacity of individual consensus replicas. This approach is particularly relevant for permissioned blockchains and enterprise applications, where state can grow into terabytes and performance requirements are stringent.

The integration of RDMA into blockchain systems is still in its infancy. We believe that combining RDMA with the protocols and platforms presented in this dissertation—PoC for security, Thunderbolt for scalable execution, CASSANDRA for availability, and ResilientDB as a testbed—could yield a new generation of high-performance, memory-scalable blockchain fabrics. Future research should address the concurrency control, fault tolerance, and security challenges outlined above, and evaluate the resulting system on realistic workloads. We hope that the ideas presented here will inspire both the academic community and industrial practitioners to explore RDMA as a cornerstone of next-generation blockchain infrastructure.

Conclusions

In this work, we presented protocols and design methodologies that can help scale large-scale distributed database systems. These large-scale distributed applications often face crash-failures and byzantine attacks. As a result, there has been a growing interest in designing resilient distributed applications, which is also the principal philosophy behind blockchain technology. The key focus of this work is to scale permissioned blockchain fabrics and to ensure they remain consistent.

The first work of this thesis focuses on scaling PoW protocols, which leverage distributed mining to eliminate long-range attacks. PoC is noninvasive as it works on the output of the underlying PoS/BFT consensus protocol and has minimal impact on the performance of existing decentralized systems. PoC advocates for collaborative mining, which, like PoW, requires miners to solve a compute-intensive puzzle, but all the miners are now working together (instead of competing) to solve the same puzzle.

The most closely related work, Bitcoin’s *centralized mining pools*, also attempts to reduce the costs associated with mining. As the name indicates, these mining pools are centralized and managed by an organization. The organization sets the rules for the mining pool, decides which node should receive a reward and how much reward, and controls which node can participate in the pool. Not only is the existence of centralized mining pools against the ethos of a decentralized system, but the managing organization charges fees for management without spending any computational resources. Further, attempts to create a decentralized mining pool have been unsuccessful due to nodes not doing designated tasks and a lack of accountability: the last block added by any decentralized mining pool in Bitcoin was in 2019.

PoC, in essence, functions as a single decentralized mining pool where all the nodes collaborate to find a solution for the compute-intensive puzzle. Like PoW, nodes are still spending their computational resources to find the nonce, which makes it computationally expensive for the adversary to create an adversarial ledger. However, we need to ensure that, like centralized mining pools, we reduce the wastage of computational resources while also guaranteeing decentralization and *fairness*. We do so by splitting the compute-intensive puzzle into a set of unique sub-problems, and each node works on a unique subset of these sub-problems; the solution to the original compute-intensive problem is present in these subsets. We also need to provide

accountability and deter malicious nodes from not doing work, as it can delay the discovery of the solution. PoC does so through our slice-shifting protocol, which identifies and penalizes a malicious miner and transfers its work to honest miners.

This additional phase is to introduce the Thunderbolt protocol to resolve the concurrency control issues during scaling and sharding the systems. Thunderbolt organizes transactions into distinct shards to mitigate potential conflicts. Most blockchain systems adopt the Order-Execute (*OE*) model, where transactions are ordered through consensus before execution. *OE*-based systems often employ deterministic concurrency controls by constructing transaction dependency graphs to optimize parallelism. However, platforms like Hyperledger Fabric utilize the Execute-Order-Validate (*EOV*) model, executing transactions before consensus, to enhance flexibility and Optimistic Concurrency Control (*OCC*) to improve the concurrency.

Scalability via DAG and Sharding: Another critical area for advancement in blockchain technology is scalability, particularly in supporting parallel execution. Recent advancements leverage Directed Acyclic Graph (DAG)-based consensus protocols to improve scalability. These protocols enable replicas to submit proposals concurrently by building a DAG that links new proposals to historical ones. This architecture has gained significant recognition in the industry due to its robust security features, exceptional scalability, and capability to support smart contracts. Complementary to DAG-based approaches, sharding techniques allow parallel transaction processing across each shard, reducing consensus overhead.

However, the above approaches do not effectively improve the execution of smart contracts. The challenges raise the question of whether it is possible to design a sharding system that does not depend on understanding the read and write sets of transactions, nor requires additional coordinators to manage cross-shard transactions.

We propose Thunderbolt, an innovative sharding architecture that seamlessly processes both single-shard (Single-shard TXs) and cross-shard transactions (Cross-shard TXs) without centralized coordinators. Thunderbolt effectively integrates the *OE* and *EOV* models, where the *EOV* model enhances parallelism in the execution of Single-shard TXs, while the *OE* model minimizes the abort rate when handling Cross-shard TXs across different shards. Furthermore, the *OE* model ensures a coordinated execution order between these two types of transactions, maintaining the overall correctness of the execution process.

Similar to conventional sharding systems, Thunderbolt organizes transactions into distinct shards to mitigate potential conflicts. In particular, each replica within Thunderbolt corresponds to a single shard and acts as a shard proposer, proposing transactions within that shard. Thunderbolt employs a DAG-based consensus protocol to reach an agreement on the execution results provided by each shard proposer.

Thunderbolt introduces a concurrent executor (*CE*) designed to improve the execution of Single-shard TXs before reaching consensus. Unlike traditional concurrency protocols that primarily manage conflicts based on the order of arrival, the *CE* utilizes a non-deterministic ordering system based on the execution run-time states of each transaction. This innovative approach minimizes the abort rate, thereby reducing transaction latency.

Finally, to evaluate our scalable BFT protocols, we designed ResilientDB, a high-throughput yielding permissioned blockchain fabric.

ResilientDB platform is an Apache Incubating open-source project providing a blockchain ecosystem with high performance. ResilientDB allows researchers to develop their own protocols much more conveniently without considering any system infrastructure so that they can have a fair comparison.

ResilientDB stands out as the first platform to deliver a high-performance blockchain ecosystem. It is designed to operate within private environments tailored for industrial applications.

Our ResilientDB fabric proves that designing such a well-crafted system is possible and even if such a system employs a three-phase protocol, It can outperform other systems utilizing a single-phase protocol. This endeavor requires us to dissect existing permissioned blockchain systems and highlight different factors affecting their performance.

References

- [1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2009).
URL <https://bitcoin.org/bitcoin.pdf>
- [2] L. Lamport, D. Malkhi, L. Zhou, Vertical paxos and primary-backup replication, in: Proceedings of the 28th ACM symposium on Principles of distributed computing, 2009, pp. 312–313.
- [3] N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg, The primary-backup approach, Distributed systems 2 (1993) 199–216.
- [4] B. W. Lampson, H. E. Sturgis, Crash recovery in a distributed data storage system, Xerox Palo Alto Research Center Palo Alto, California, 1979.
- [5] A. Miller, Permissioned and permissionless blockchains, Blockchain for distributed systems security (2019) 193–204.
- [6] S. Solat, P. Calvez, F. Naït-Abdesselam, Permissioned vs. permissionless blockchain: How and why there is only one right choice., J. Softw. 16 (3) (2021) 95–106.
- [7] C. V. Helliär, L. Crawford, L. Rocca, C. Teodori, M. Veneziani, Permissionless and permissioned blockchain diffusion, International Journal of Information Management 54 (2020) 102136.
- [8] Y. Guo, C. Liang, Blockchain application and outlook in the banking industry, Financial innovation 2 (2016) 1–12.
- [9] T. Shah, S. Jani, Applications of blockchain technology in banking & finance, Parul CUniversity, Vadodara, India (2018).
- [10] M. Castro, B. Liskov, Practical byzantine fault tolerance and proactive recovery, ACM Transactions on Computer Systems 20 (4) (2002) 398–461. doi:10.1145/571637.571640.
- [11] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, I. Abraham, HotStuff: BFT consensus with linearity and responsiveness, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, ACM, 2019, pp. 347–356. doi:10.1145/3293611.3331591.
- [12] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) (2014) 1–32.

- [13] S. Azouvi, G. Danezis, V. Nikolaenko, Winkle: Foiling long-range attacks in proof-of-stake systems, in: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies, Association for Computing Machinery, New York, NY, USA, 2020, p. 189–201. doi:10.1145/3419614.3423260.
- [14] M. K. Franklin, A survey of key evolving cryptosystems, *Int. J. Secur. Networks* 1 (1/2) (2006) 46–53. doi:10.1504/IJSN.2006.010822.
URL <https://doi.org/10.1504/IJSN.2006.010822>
- [15] E. A. Brewer, Towards robust distributed systems (abstract), in: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 7. doi:10.1145/343477.343502.
- [16] M. Kleppmann, A critique of the CAP theorem, *CoRR* abs/1509.05393 (2015). arXiv:1509.05393.
URL <http://arxiv.org/abs/1509.05393>
- [17] S. Gilbert, N. A. Lynch, Perspectives on the cap theorem, *Computer* 45 (02) (2012) 30–36. doi:10.1109/MC.2011.389.
- [18] P. A. Bernstein, N. Goodman, Concurrency control in distributed database systems, *ACM Computing Surveys (CSUR)* 13 (2) (1981) 185–221.
- [19] P. A. Bernstein, N. Goodman, An algorithm for concurrency control and recovery in replicated distributed databases, *ACM Transactions on Database Systems (TODS)* 9 (4) (1984) 596–615.
- [20] E. Androulaki et al., Hyperledger Fabric: A distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, ACM, 2018, pp. 30:1–30:15. doi:10.1145/3190508.3190538.
- [21] J. Gray, Notes on data base operating systems, in: Operating Systems, An Advanced Course, Springer-Verlag, 1978, pp. 393–481. doi:10.1007/3-540-08755-9_9.
- [22] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, G. Danezis, Chainspace: A sharded smart contracts platform, *arXiv preprint arXiv:1708.03778* (2017).
- [23] J. Hellings, D. P. Hughes, J. Primero, M. Sadoghi, Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing, *arXiv preprint arXiv:2008.04450* (2020).
- [24] L. Lamport, The part-time parliament (1998).
- [25] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: ATC, 2014.
- [26] G. Danezis, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, Narwhal and tusk: a dag-based mempool and efficient BFT consensus, in: EuroSys '22: Seventeenth European Conference on Computer Systems, ACM, 2022, pp. 34–50. doi:10.1145/3492321.3519594.

- [27] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, EIP-150 revision (2016).
URL <https://gavwood.com/paper.pdf>
- [28] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling byzantine agreements for cryptocurrencies, in: Proceedings of the 26th Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA, 2017, p. 51–68. doi:10.1145/3132747.3132757.
- [29] M. Castro, B. Liskov, Practical byzantine fault tolerance and proactive recovery, ACM Trans. Comput. Syst. 20 (4) (2002) 398–461. doi:10.1145/571637.571640.
- [30] S. King, S. Nadal, PPCoin: Peer-to-peer crypto-currency with Proof-of-Stake (2012).
URL <https://www.peercoin.net/whitepapers/peercoin-paper.pdf>
- [31] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, Zyzzyva: Speculative byzantine fault tolerance, ACM Transactions on Computer Systems 27 (4) (2009) 7:1–7:39. doi:10.1145/1658357.1658358.
- [32] M. K. Aguilera, I. Keidar, D. Malkhi, J. Martin, A. Shraer, Reconfiguring replicated atomic storage: A tutorial, Bull. EATCS 102 (2010) 84–108.
URL <http://eatcs.org/beatcs/index.php/beatcs/article/view/156>
- [33] Y. Wang, J. Sun, X. Wang, Y. Wei, H. Wu, Z. Yu, G. Chu, Sperax: An approach to defeat long range attacks in blockchains, in: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2020, pp. 574–579. doi:10.1109/INFOCOMWKSHPS50562.2020.9163036.
- [34] E. Tas, D. Tse, F. Gai, S. Kannan, M. Maddah-Ali, F. Yu, Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities, in: 2023 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 126–145. doi:10.1109/SP46215.2023.10179426.
- [35] Y. Yun, Lazarus group’s favorite exploit revealed — crypto hacks analysis (2024).
URL <https://cointelegraph.com/magazine/north-korean-hackers-private-keys-flash-loan-attacks/>
- [36] R. Nambampurath, K. Baird, 2,000 crypto private keys stolen from edge wallet (2023).
URL <https://beincrypto.com/2000-crypto-private-keys-stolen-edge-wallet/>
- [37] E. R, Private key breaches surge in 2024 with over \$239 million stolen! is your crypto safe? (2024).
- [38] I. Bentov, C. Lee, A. Mizrahi, M. Rosenfeld, Proof of activity: Extending bitcoin’s proof of work via proof of stake [extended abstract], SIGMETRICS Perform. Eval. Rev. 42 (3) (2014) 34–37. doi:10.1145/2695533.2695545.

- [39] M. Nijkerk, Ethereum Unstaking Requests Now Face About a 17-Day Wait (2023).
URL <https://www.coindesk.com/tech/2023/04/18/ethereum-unstaking-requests-now-face-about-a-17-day-wait/>
- [40] M. Sherman, Nodes on Bitcoin’s Lightning Network Double in 3 Months (2021).
URL <https://www.coindesk.com/tech/2021/07/14/nodes-on-bitcoins-lightning-network-double-in-3-months/>
- [41] N. Dana Troutman, A. Laszka, Poolparty: Efficient blockchain-agnostic decentralized mining pool, in: 2021 The 3rd International Conference on Blockchain Technology, Association for Computing Machinery, New York, NY, USA, 2021, p. 20–27. doi:10.1145/3460537.3460554.
- [42] L. Luu, Y. Velner, J. Teutsch, P. Saxena, SmartPool: Practical decentralized pooled mining, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 1409–1426.
- [43] P2Pool (2011).
URL <http://p2pool.in/>
- [44] Apache resilientdb (incubating) (2024).
URL <https://resilientdb.incubator.apache.org/>
- [45] V. Gramoli, R. Guerraoui, A. Lebedev, C. Natoli, G. Voron, Diablo-v2: A benchmark for blockchain systems (2012).
URL <https://infoscience.epfl.ch/record/294268>
- [46] Diem Association, Diem bft (2022).
URL <https://www.diem.com/en-us/>
- [47] J. Chase, Quorum whitepaper (2019).
URL <https://github.com/ConsenSys/quorum/blob/master/docs/Quorum%20Whitepaper%20v0.2.pdf>
- [48] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, A. Tomescu, Sbft: a scalable and decentralized trust infrastructure, in: 2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN), IEEE, 2019, pp. 568–580.
- [49] B. David, P. Gaži, A. Kiayias, A. Russell, Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain, in: J. B. Nielsen, V. Rijmen (Eds.), Advances in Cryptology – EUROCRYPT 2018, Springer International Publishing, Cham, 2018, pp. 66–98.

- [50] Eyal, A. E. Gencer, E. G. Sirer, R. V. Renesse, Bitcoin-NG: A scalable blockchain protocol, in: 13th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association, Santa Clara, CA, 2016, pp. 45–59.
- [51] E. N. Tas, D. Tse, F. Yu, S. Kannan, Babylon: Reusing bitcoin mining to enhance proof-of-stake security, arXiv preprint arXiv:2201.07946 (2022).
- [52] S. Azouvi, M. Vukolić, Pikachu: Securing pos blockchains from long-range attacks by checkpointing into bitcoin pow using taproot, arXiv preprint arXiv:2208.05408 (2022).
- [53] K. Okupski, Bitcoin developer reference (2016).
URL <https://github.com/minium/Bitcoin-Spec>
- [54] M. O. Rabin, Efficient dispersal of information for security, load balancing, and fault tolerance, Journal of the ACM (JACM) 36 (2) (1989) 335–348.
- [55] I. Abraham, K. Nayak, L. Ren, Z. Xiang, Good-case latency of byzantine broadcast: A complete categorization, in: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, PODC’21, Association for Computing Machinery, New York, NY, USA, 2021, p. 331–341. doi:10.1145/3465084.3467899.
- [56] A. Clement, E. Wong, L. Alvisi, M. Dahlin, M. Marchetti, Making byzantine fault tolerant systems tolerate byzantine faults, in: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, USENIX, 2009, pp. 153–168.
- [57] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, T. Riche, Upright cluster services, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 277–290. doi:10.1145/1629575.1629602.
- [58] Algorand Foundation, General FAQ (2022).
URL <https://www.algorand.foundation/general-faq>
- [59] Ethereum Foundation, Staking withdrawals (2023).
URL <https://ethereum.org/en/staking/withdrawals/>
- [60] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, A. Tomescu, SBFT: A scalable and decentralized trust infrastructure, in: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, 2019. doi:10.1109/DSN.2019.00063.
- [61] F. D’Amato, J. Neu, E. N. Tas, D. Tse, Goldfish: No more attacks on proof-of-stake ethereum, arXiv preprint arXiv:2209.03255 (2022).

- [62] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, 2010, pp. 143–154. doi:10.1145/1807128.1807152.
- [63] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, K.-L. Tan, BLOCKBENCH: A framework for analyzing private blockchains, in: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, 2017, pp. 1085–1100. doi:10.1145/3035918.3064033.
- [64] Blockchain.com, Hashrate Distribution: An estimation of hashrate distribution amongst the largest mining pools. (2023).
URL <https://www.blockchain.com/explorer/charts/pools>
- [65] C. Stathakopoulou, M. Pavlovic, M. Vukolić, State machine replication scalability made simple, in: Proceedings of the Seventeenth European Conference on Computer Systems, Association for Computing Machinery, New York, NY, USA, 2022, p. 17–33. doi:10.1145/3492321.3519579.
- [66] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, I. Zlotchi, Leaderless consensus, in: 41st IEEE International Conference on Distributed Computing Systems, IEEE, 2021, pp. 392–402. doi:10.1109/ICDCS51616.2021.00045.
- [67] A. Lekssays, G. Sirigu, B. Carminati, E. Ferrari, Malrec: A blockchain-based malware recovery framework for internet of things, in: Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22, Association for Computing Machinery, New York, NY, USA, 2022. doi:10.1145/3538969.3544446.
- [68] C. Rondanini, B. Carminati, F. Daidone, E. Ferrari, Blockchain-based controlled information sharing in inter-organizational workflows, in: 2020 IEEE International Conference on Services Computing (SCC), 2020, pp. 378–385. doi:10.1109/SCC49832.2020.00056.
- [69] C. Zhang, C. Xu, J. Xu, Y. Tang, B. Choi, Gem²-tree: A gas-efficient structure for authenticated range queries in blockchain, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 842–853. doi:10.1109/ICDE.2019.00080.
- [70] P. Aublin, R. Guerraoui, N. Knezevic, V. Quéma, M. Vukolic, The next 700 BFT protocols, ACM Trans. Comput. Syst. 32 (4) (2015) 12:1–12:45. doi:10.1145/2658994.
- [71] S. Liu, P. Viotti, C. Cachin, V. Quéma, M. Vukolic, Xft: Practical fault tolerance beyond crashes, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, USA, 2016, p. 485–500.

- [72] P. Sheng, G. Wang, K. Nayak, S. Kannan, P. Viswanath, BFT protocol forensics, in: CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2021, pp. 1722–1743. doi:10.1145/3460120.3484566.
- [73] P. Ruan, T. T. A. Dinh, Q. Lin, M. Zhang, G. Chen, B. C. Ooi, Lineagechain: a fine-grained, secure and efficient data provenance system for blockchains, VLDB J. 30 (1) (2021) 3–24. doi:10.1007/s00778-020-00646-1.
- [74] D. Loghin, T. T. A. Dinh, A. Maw, C. Gang, Y. M. Teo, B. C. Ooi, Blockchain goes green? part ii: Characterizing the performance and cost of blockchains on the cloud and at the edge, arXiv preprint arXiv:2205.06941 (2022).
- [75] J. A. Chacko, R. Mayer, H.-A. Jacobsen, How to optimize my blockchain? a multi-level recommendation approach, Proceedings of the ACM on Management of Data 1 (1) (2023) 1–27.
- [76] Z. Xiang, T. Wang, W. Lin, D. Wang, Practical differentially private and byzantine-resilient federated learning, Proceedings of the ACM on Management of Data 1 (2) (2023) 1–26.
- [77] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, B. Ford, Enhancing bitcoin security and performance with strong consistency via collective signing, in: 25th usenix security symposium (usenix security 16), 2016, pp. 279–296.
- [78] M. F. Madsen, M. Gaub, M. E. Kirkbro, S. Debois, Transforming byzantine faults using a trusted execution environment, in: 15th European Dependable Computing Conference, 2019, pp. 63–70. doi:10.1109/EDCC.2019.00022.
- [79] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, S. Yan, Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX, Association for Computing Machinery, New York, NY, USA, 2020, p. 955–970.
- [80] R. Yuan, Y. Xia, H. Chen, B. Zang, J. Xie, Shadoweth: Private smart contract on public blockchain, J. Comput. Sci. Technol. 33 (3) (2018) 542–556. doi:10.1007/s11390-018-1839-y.
- [81] V. A. Sartakov, S. Brenner, S. B. Mokhtar, S. Bouchenak, G. Thomas, R. Kapitza, Eactors: Fast and flexible trusted computing using SGX, in: P. Ferreira, L. Shriram (Eds.), Proceedings of the 19th International Middleware Conference, ACM, 2018, pp. 187–200. doi:10.1145/3274808.3274823.
- [82] M.-K. Sit, M. Bravo, Z. István, An experimental framework for improving the performance of bft consensus for future permissioned blockchains, in: Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, DEBS '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 55–65. doi:10.1145/3465480.3466922.

- [83] L. Kuhring, Z. István, A. Sorniotti, M. Vukolić, Streamchain: Building a low-latency permissioned blockchain for enterprise use-cases, in: 2021 IEEE International Conference on Blockchain (Blockchain), 2021, pp. 130–139. doi:10.1109/Blockchain53845.2021.00027.
- [84] E. Blass, F. Kerschbaum, BOREALIS: building block for sealed bid auctions on blockchains, in: ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, ACM, 2020, pp. 558–571. doi:10.1145/3320269.3384752.
- [85] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, D. Zage, Scaling byzantine fault-tolerant replication to wide area networks, in: International Conference on Dependable Systems and Networks (DSN'06), 2006, pp. 105–114. doi:10.1109/DSN.2006.63.
- [86] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, B. C. Ooi, Towards scaling blockchain systems via sharding, in: Proceedings of the 2019 International Conference on Management of Data, ACM, 2019, pp. 123–140. doi:10.1145/3299869.3319889.
- [87] M. J. Amiri, D. Agrawal, A. El Abbadi, SharPer: Sharding Permissioned Blockchains Over Network Clusters, Association for Computing Machinery, 2021, p. 76–88.
- [88] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, N. Crooks, Basil: Breaking up bft with acid (transactions), in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSOP '21, Association for Computing Machinery, 2021, p. 1–17. doi:10.1145/3477132.3483552.
- [89] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, R. Ramamurthy, Blockchaindb: A shared database on blockchains, Proceedings of the VLDB Endowment 12 (11) (2019) 1597–1609.
- [90] M. J. Amiri, D. Agrawal, A. E. Abbadi, Caper: a cross-application permissioned blockchain, Proceedings of the VLDB Endowment 12 (11) (2019) 1385–1398.
- [91] E. Deirmentzoglou, G. Papakyriakopoulos, C. Patsakis, A survey on long-range attacks for proof of stake protocols, IEEE Access 7 (2019) 28712–28725.
- [92] M. Kohlweiss, V. Madathil, K. Nayak, A. Scafuro, On the anonymity guarantees of anonymous proof-of-stake protocols, in: 42nd IEEE Symposium on Security and Privacy, IEEE, 2021, pp. 1818–1833. doi:10.1109/SP40001.2021.00107.
- [93] S. King, S. Nadal, Ppcoin: Peer-to-peer crypto-currency with proof-of-stake, self-published paper, August 19 (1) (2012).
- [94] A. Kiayias, A. Russell, B. David, R. Oliynykov, Ouroboros: A provably secure proof-of-stake blockchain protocol, in: Advances in Cryptology – CRYPTO 2017, Springer International Publishing, Cham, 2017, pp. 357–388.

- [95] S. Barber, X. Boyen, E. Shi, E. Uzun, Bitter to better — how to make bitcoin a better currency, in: *Financial Cryptography and Data Security*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 399–414.
- [96] V. Buterin, Proof of stake: How i learned to love weak subjectivity (2014).
URL <https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity>
- [97] P. Daian, R. Pass, E. Shi, Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake, in: *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, Springer-Verlag, Berlin, Heidelberg, 2019, p. 23–41. doi:10.1007/978-3-030-32101-7_2.
- [98] M. Drijvers, S. Gorbunov, G. Neven, H. Wee, Pixel: Multi-signatures for consensus, in: *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, 2020, pp. 2093–2110.
- [99] A. Yakovenko, Solana: A new architecture for a high performance blockchain v0. 8.13, Whitepaper (2018).
- [100] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, B. Ford, Enhancing bitcoin security and performance with strong consistency via collective signing, in: *Proceedings of the 25th USENIX Conference on Security Symposium, SEC’16*, USENIX Association, USA, 2016, p. 279–296.
- [101] D. Gupta, J. Saia, M. Young, Proof of Work Without All the Work, in: *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN ’18*, ACM, New York, NY, USA, 2018, pp. 6:1–6:10. doi:10.1145/3154273.3154333.
- [102] C. Gorenflo, S. Lee, L. Golab, S. Keshav, Fastfabric: Scaling hyperledger fabric to 20 000 transactions per second, *International Journal of Network Management* 30 (5) (2020) e2099.
- [103] S. Gupta, S. Rahnema, M. Sadoghi, Permissioned blockchain through the looking glass: Architectural and implementation lessons learned, in: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2020, pp. 754–764.
- [104] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, P. Jayachandran, Blockchain meets database: Design and implementation of a blockchain relational database, arXiv preprint arXiv:1903.01919 (2019).
- [105] M. Casey, J. Crane, G. Gensler, S. Johnson, N. Narula, The impact of blockchain technology on finance: A catalyst for change (2018).
- [106] L. Ge, C. Brewster, J. Spek, A. Smeenk, J. Top, F. Van Diepen, B. Klaase, C. Graumans, M. d. R. de Wildt, Blockchain for agriculture and food: Findings from the pilot study, no. 2017-112, Wageningen Economic Research, 2017.

- [107] M. Herlihy, Blockchains from a distributed computing perspective, *Communications of the ACM* 62 (2) (2019) 78–85.
- [108] M. N. Kamel Boulos, J. T. Wilson, K. A. Clauson, Geospatial blockchain: promises, challenges, and scenarios in health and healthcare (2018).
- [109] A. Narayanan, J. Clark, Bitcoin’s academic pedigree, *Communications of the ACM* 60 (12) (2017) 36–45.
- [110] F. Nawab, M. Sadoghi, Blockplane: A global-scale byzantizing middleware, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 124–135.
- [111] M. Pisa, M. Juden, Blockchain and economic development: Hype vs. reality, *Center for global development policy paper* 107 (150) (2017) 1–42.
- [112] N. Szabo, The idea of smart contracts, 1997.
URL <http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/idea.html>
- [113] N. Szabo, Formalizing and securing relationships on public networks, *First monday* (1997).
- [114] V. Buterin, et al., A next-generation smart contract and decentralized application platform, *white paper* 3 (37) (2014) 2–1.
- [115] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, H.-N. Lee, Ethereum smart contract analysis tools: A systematic review, *IEEE Access* 10 (2022) 57037–57062.
- [116] A. Aldweesh, M. Alharby, M. Mehrnezhad, A. Van Moorsel, Opbench: A cpu performance benchmark for ethereum smart contract operation code, in: 2019 IEEE International Conference on Blockchain (Blockchain), IEEE, 2019, pp. 274–281.
- [117] S. Gupta, J. Hellings, M. Sadoghi, RCC: resilient concurrent consensus for high-throughput secure transaction processing, in: 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, IEEE, 2021, pp. 1392–1403. doi:10.1109/ICDE51399.2021.00124. URL <https://doi.org/10.1109/ICDE51399.2021.00124>
- [118] S. Gupta, J. Hellings, S. Rahnema, M. Sadoghi, Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation *abs/1911.00838* (2019).
URL <https://arxiv.org/abs/1911.00838>
- [119] T. M. Qadah, M. Sadoghi, Quecc: A queue-oriented, control-free concurrency architecture, in: *Proceedings of the 19th International Middleware Conference*, 2018, pp. 13–25.

- [120] J. M. Faleiro, D. J. Abadi, J. M. Hellerstein, High performance transactions via early write visibility, *Proceedings of the VLDB Endowment* 10 (5) (2017).
- [121] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W.-F. Wong, M. Zhang, Exploiting single-threaded model in multi-core in-memory systems, *IEEE Transactions on Knowledge and Data Engineering* 28 (10) (2016) 2635–2650.
- [122] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, J. Li, Scaling multicore databases via constrained parallel execution, in: *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1643–1658.
- [123] H.-T. Kung, J. T. Robinson, On optimistic methods for concurrency control, *ACM Transactions on Database Systems (TODS)* 6 (2) (1981) 213–226.
- [124] G. Danezis, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, Narwhal and tusk: A dag-based mempool and efficient bft consensus, in: *Proceedings of the Seventeenth European Conference on Computer Systems*, Association for Computing Machinery, New York, NY, USA, 2022, p. 34–50. doi:10.1145/3492321.3519594.
- [125] A. Spiegelman, N. Giridharan, A. Sonnino, L. Kokoris-Kogias, Bullshark: Dag bft protocols made practical, in: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.
- [126] I. Keidar, E. Kokoris-Kogias, O. Naor, A. Spiegelman, All you need is dag, in: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.
- [127] I. Keidar, O. Naor, O. Poupko, E. Shapiro, Cordial miners: Fast and efficient consensus for every eventuality, *arXiv preprint arXiv:2205.09174* (2022).
- [128] C. Stathakopoulou, M. Wei, M. Yin, H. Zhang, D. Malkhi, Bbca-ledger: High throughput consensus meets low latency, *arXiv preprint arXiv:2306.14757* (2023).
- [129] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, A. Sonnino, Mysticeti: Low-latency dag consensus with fast commit path, *arXiv preprint arXiv:2310.14821* (2023).
- [130] B. Arun, Z. Li, F. Suri-Payer, S. Das, A. Spiegelman, Shoal++: High throughput dag bft can be fast!, *arXiv preprint arXiv:2405.20488* (2024).
- [131] A. Spiegelman, B. Aurn, R. Gelashvili, Z. Li, Shoal: Improving dag-bft latency and robustness, *arXiv preprint arXiv:2306.03058* (2023).
- [132] N. Shrestha, R. Shrothrium, A. Kate, K. Nayak, Sailfish: Towards improving latency of dag-based bft, *Cryptology ePrint Archive* (2024).

- [133] D. Malkhi, C. Stathakopoulou, M. Yin, Bbca-chain: One-message, low latency bft consensus on a dag, arXiv preprint arXiv:2310.06335 (2023).
- [134] M. Raikwar, N. Polyanskii, S. Müller, Sok: Dag-based consensus protocols, in: 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), IEEE, 2024, pp. 1–18.
- [135] S. Gupta, S. Rahnema, J. Hellings, M. Sadoghi, ResilientDB: Global scale resilient blockchain fabric, PVLDB 13 (6) (2020) 868–883. doi:10.14778/3380750.3380757.
- [136] J. Hellings, M. Sadoghi, ByShard: sharding in a byzantine environment, VLDB J. 32 (6) (2023) 1343–1367.
- [137] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, B. Ford, Omniledger: A secure, scale-out, decentralized ledger via sharding, in: 2018 IEEE symposium on security and privacy (SP), IEEE, 2018, pp. 583–598.
- [138] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, B. C. Ooi, Towards scaling blockchain systems via sharding, in: Proceedings of the 2019 international conference on management of data, 2019, pp. 123–140.
- [139] M. Zamani, M. Movahedi, M. Raykova, Rapidchain: Scaling blockchain via full sharding, in: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 931–948.
- [140] A. Sonnino, S. Bano, M. Al-Bassam, G. Danezis, Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers, in: 2020 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2020, pp. 294–308.
- [141] L. Zhang, Y. Wang, Y. Ding, H. Liang, C. Yang, C. Li, Sharding technologies in blockchain: Basics, state of the art, and challenges, in: International Conference on Blockchain and Trustworthy Systems, Springer, 2023, pp. 242–255.
- [142] D. Yang, C. Long, H. Xu, S. Peng, A review on scalability of blockchain, in: Proceedings of the 2020 2nd International Conference on Blockchain Technology, 2020, pp. 1–6.
- [143] M. J. Amiri, D. Agrawal, A. El Abbadi, Sharper: Sharding permissioned blockchains over network clusters, in: Proceedings of the 2021 international conference on management of data, 2021, pp. 76–88.
- [144] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, S. Guo, Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding, in: IEEE INFOCOM 2022-IEEE Conference on Computer Communications, IEEE, 2022, pp. 1968–1977.

- [145] F. Cheng, J. Xiao, C. Liu, S. Zhang, Y. Zhou, B. Li, B. Li, H. Jin, Shardag: Scaling dag-based blockchains via adaptive sharding, in: 2024 IEEE 40th International Conference on Data Engineering (ICDE), IEEE, 2024, pp. 2068–2081.
- [146] Z. Hong, S. Guo, P. Li, Scaling blockchain via layered sharding, IEEE Journal on Selected Areas in Communications 40 (12) (2022) 3575–3588.
- [147] N. Qi, Y. Yuan, F.-Y. Wang, Dag-block: A novel architecture for scaling blockchain-enabled cryptocurrencies, IEEE Transactions on Computational Social Systems 11 (1) (2022) 378–388.
- [148] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino, et al., Sui lutris: A blockchain combining broadcast and consensus, arXiv preprint arXiv:2310.18042 (2023).
- [149] R. V. Rasmussen, M. A. Trick, Round robin scheduling—a survey, European Journal of Operational Research 188 (3) (2008) 617–636.
- [150] R. Agrawal, M. J. Carey, M. Livny, Concurrency control performance modeling: Alternatives and implications, ACM Transactions on Database Systems (TODS) 12 (4) (1987) 609–654.
- [151] P. Franaszek, J. T. Robinson, Limitations of concurrency in transaction processing, ACM Transactions on Database Systems (TODS) 10 (1) (1985) 1–28.
- [152] Y. C. Tay, N. Goodman, R. Suri, Locking performance in centralized databases, ACM Transactions on Database Systems (TODS) 10 (4) (1985) 415–462.
- [153] Microsoft, eevm.
URL <https://github.com/microsoft/eEVM>
- [154] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, A. Sonnino, Mysticeti: Low-latency dag consensus with fast commit path, arXiv preprint arXiv:2310.14821 (2023).
- [155] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, Journal of the ACM (JACM) 35 (2) (1988) 288–323.
- [156] R. Cattell, Scalable sql and nosql data stores, Acm Sigmod Record 39 (4) (2011) 12–27.
- [157] X. Chen, M.-g. Xie, A split-and-conquer approach for analysis of extraordinarily large data, Statistica Sinica (2014) 1655–1684.
- [158] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., Apache spark: a unified engine for big data processing, Communications of the ACM 59 (11) (2016) 56–65.

- [159] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, Scope: easy and efficient parallel processing of massive data sets, *Proceedings of the VLDB Endowment* 1 (2) (2008) 1265–1276.
- [160] S. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical partitioning algorithms for database design, *ACM Transactions on Database Systems (TODS)* 9 (4) (1984) 680–710.
- [161] M. Shreedhar, G. Varghese, Efficient fair queueing using deficit round robin, in: *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1995, pp. 231–242.
- [162] J. Mirkovic, P. Reiher, A taxonomy of ddos attack and ddos defense mechanisms, *ACM SIGCOMM Computer Communication Review* 34 (2) (2004) 39–53.
- [163] C. Douligeris, A. Mitrokotsa, Ddos attacks and defense mechanisms: classification and state-of-the-art, *Computer networks* 44 (5) (2004) 643–666.
- [164] E. Soisalon-Soininen, T. Ylönen, Partial strictness in two-phase locking, in: *International Conference on Database Theory*, Springer, 1995, pp. 139–147.
- [165] smallbank benchmark (2019).
URL <http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/>
- [166] M. Alomari, M. Cahill, A. Fekete, U. Rohm, The cost of serializability on platforms that use snapshot isolation, in: *2008 IEEE 24th International Conference on Data Engineering*, IEEE, 2008, pp. 576–585.
- [167] P. Thakkar, S. Natarajan, Scaling hyperledger fabric using pipelined execution and sparse peers, *arXiv preprint arXiv:2003.05113* (2020).
- [168] M. Li, Y. Wang, S. Ma, C. Liu, D. Huo, Y. Wang, Z. Xu, Auto-tuning with reinforcement learning for permissioned blockchain systems, *Proceedings of the VLDB Endowment* 16 (5) (2023) 1000–1012.
- [169] Z. Lai, C. Liu, E. Lo, When private blockchain meets deterministic database, *Proceedings of the ACM on Management of Data* 1 (1) (2023) 1–28.
- [170] Y. Zhang, J. K. Wang, Y. J. Han, Cchain: a high throughput blockchain system, in: *Second International Conference on Digital Society and Intelligent Systems (DSInS 2022)*, Vol. 12599, SPIE, 2023, pp. 129–136.
- [171] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, B. C. Ooi, A transactional perspective on execute-order-validate blockchains, in: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

- [172] Z. Peng, Y. Zhang, Q. Xu, H. Liu, Y. Gao, X. Li, G. Yu, Neuchain: a fast permissioned blockchain system with deterministic ordering, *Proceedings of the VLDB Endowment* 15 (11) (2022) 2585–2598.
- [173] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, M. Stonebraker, Staring into the abyss: An evaluation of concurrency control with one thousand cores (2014).
- [174] X. Yu, A. Pavlo, D. Sanchez, S. Devadas, Tictoc: Time traveling optimistic concurrency control, in: *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1629–1642.
- [175] G. Wang, Z. J. Shi, M. Nixon, S. Han, Sok: Sharding on blockchain, in: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 41–61.
- [176] S. Rahnema, S. Gupta, R. Sogani, D. Krishnan, M. Sadoghi, RingBFT: Resilient Consensus over Sharded Ring Topology, in: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT, OpenProceedings.org*, 2022, pp. 2:298–2:311. doi:10.48786/edbt.2022.17.
- [177] A. Churyumov, Byteball: A decentralized system for storage and transfer of value, URL <https://byteball.org/Byteball.pdf> (2016) 11.
- [178] L. Baird, The swirls hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance, *Swirls Tech Reports SWIRLDS-TR-2016-01*, Tech. Rep 34 (2016) 9–11.
- [179] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, A. C.-C. Yao, A decentralized blockchain with high throughput and fast confirmation, in: *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 515–528.
- [180] H. Yu, I. Nikolić, R. Hou, P. Saxena, Ohie: Blockchain scaling made simple, in: *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 90–105.
- [181] J. Xu, Y. Cheng, C. Wang, X. Jia, Occam: A secure and adaptive scaling scheme for permissionless blockchain, in: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2021, pp. 618–628.
- [182] J. Xiao, S. Zhang, Z. Zhang, B. Li, X. Dai, H. Jin, Nezha: Exploiting concurrency for transaction processing in dag-based blockchains, in: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2022, pp. 269–279.
- [183] P. Thakkar, S. Nathan, B. Viswanathan, Performance benchmarking and optimizing hyperledger fabric blockchain platform, in: *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*, IEEE, 2018, pp. 264–276.
- [184] P. Thakkar, S. Natarajan, Scaling blockchains using pipelined execution and sparse peers, in: *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 489–502.

- [185] A. Sharma, F. M. Schuhknecht, D. Agrawal, J. 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*, 2019, pp. 105–122.
- [186] C. Gorenflo, L. Golab, S. Keshav, Xox fabric: A hybrid approach to blockchain transaction execution, in: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2020, pp. 1–9.
- [187] D. Shasha, F. Llirbat, E. Simon, P. Valduriez, Transaction chopping: Algorithms and performance studies, *ACM Transactions on Database Systems (TODS)* 20 (3) (1995) 325–363.
- [188] D. Shasha, E. Simon, P. Valduriez, Simple rational guidance for chopping up transactions, in: *Proceedings of the 1992 ACM SIGMOD International Conference on management of Data*, 1992, pp. 298–307.
- [189] Z. Chen, H. Zhuo, Q. Xu, X. Qi, C. Zhu, Z. Zhang, C. Jin, A. Zhou, Y. Yan, H. Zhang, Schain: a scalable consortium blockchain exploiting intra-and inter-block concurrency, *Proceedings of the VLDB Endowment* 14 (12) (2021) 2799–2802.
- [190] D. Qin, A. D. Brown, A. Goel, Caracal: Contention management with deterministic concurrency control, in: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 180–194.
- [191] R. Neiheiser, A. Babaei, G. Alexopoulos, M. Kogias, E. K. Kogias, Chiron: Accelerating node synchronization without security trade-offs in distributed ledgers, *arXiv preprint arXiv:2401.14278* (2024).
- [192] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, R. 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*, 2023, pp. 232–244.
- [193] S. Gupta, J. Hellings, M. Sadoghi, *Fault-Tolerant Distributed Transactions on Blockchain*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers. (2021), 2021.
- [194] M. J. Amiri, C. Wu, D. Agrawal, A. E. Abbadi, B. T. Loo, M. Sadoghi, The bedrock of BFT: A unified platform for BFT protocol design and implementation, *CoRR abs/2205.04534* (2022). **arXiv:** 2205.04534, doi:10.48550/arXiv.2205.04534.
- [195] S. Gupta, M. J. Amiri, M. Sadoghi, Chemistry behind agreement, in: *13th Conference on Innovative Data Systems Research, CIDR 2023*, Amsterdam, The Netherlands, January 8-11, 2023, www.cidrdb.org, 2023.

- [196] M. Sadoghi, S. Blanas, Transaction Processing on Modern Hardware, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2019. doi:10.2200/S00896ED1V01Y201901DTM058.
URL <https://doi.org/10.2200/S00896ED1V01Y201901DTM058>
- [197] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, M. Sadoghi, Dissecting BFT consensus: In trusted components we trust!, in: G. A. D. Luna, L. Querzoni, A. Fedorova, D. Narayanan (Eds.), Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023, ACM, 2023, pp. 521–539.
- [198] D. Kang, S. Rahnama, J. Hellings, M. Sadoghi, SpotLess: Concurrent rotational consensus made practical through rapid view synchronization, in: 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024, IEEE, 2024.
- [199] E. A. Brewer, Towards robust distributed systems, in: PODC, Vol. 7, Portland, OR, 2000, pp. 343–477.
- [200] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, J. Yang, Overload control for scaling wechat microservices, in: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 149–161.
- [201] Forbes, Bitcoin2017failure (2017).
URL <https://www.forbes.com/sites/ktorpey/2017/11/30/2017-was-the-year-when-everyone-finally-learned-bitcoin-isnt-controlled-by-miners/>
- [202] CoinMarketCap, Ethereum2020failure (2020).
URL <https://coinmarketcap.com/academy/article/ethereums-infuriating-outage-revives-decentralization-concerns>
- [203] CoinMarketCap, Solana2021failure (2021).
URL <https://cryptonary.com/solana-network-went-down-again-mainnet-beta-restarts-after-7-hours-of-outage/>
- [204] Y. Alabdulkarim, A review of blockchain use cases and recent failures in the enterprise, in: 2023 3rd International Conference on Computing and Information Technology (ICCIT), IEEE, 2023, pp. 641–646.
- [205] J. A. Chacko, R. Mayer, H.-A. Jacobsen, Why do my blockchain transactions fail? a study of hyperledger fabric, in: Proceedings of the 2021 international conference on management of data, 2021, pp. 221–234.
- [206] P. Bailis, K. Kingsbury, The network is reliable: An informal survey of real-world communications failures, ACM Queue 12 (7) (2014). doi:10.1145/2639988.2639988.

- [207] M. Burrows, The chubby lock service for loosely-coupled distributed systems, in: Proceedings of the 7th symposium on Operating systems design and implementation, 2006, pp. 335–350.
- [208] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, L. Zhou, Boxwood: Abstractions as the foundation for storage infrastructure., in: OSDI, Vol. 4, 2004, pp. 8–8.
- [209] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, {ZooKeeper}: Wait-free coordination for internet-scale systems, in: 2010 USENIX Annual Technical Conference (USENIX ATC 10), 2010.
- [210] L. Lamport, The part-time parliament, in: Concurrency: the Works of Leslie Lamport, 2019, pp. 277–317.
- [211] I. Moraru, D. G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 358–372.
- [212] S. Gupta, S. Rahnema, M. Sadoghi, Permissioned blockchain through the looking glass: Architectural and implementation lessons learned, in: 40th International Conference on Distributed Computing Systems, IEEE, 2020.
- [213] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX annual technical conference (USENIX ATC 14), 2014, pp. 305–319.
- [214] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al., Spanner: Google’s globally distributed database, ACM Transactions on Computer Systems (TOCS) 31 (3) (2013) 1–22.
- [215] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al., Windows azure storage: a highly available cloud storage service with strong consistency, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 143–157.
- [216] M. Elhemali, N. Gallagher, B. Tang, N. Gordon, H. Huang, H. Chen, J. Idziorek, M. Wang, R. Krog, Z. Zhu, et al., Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service, in: 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022, pp. 1037–1048.
- [217] D. Abadi, Consistency tradeoffs in modern distributed database system design: Cap is only part of the story, Computer 45 (2) (2012) 37–42. doi:10.1109/MC.2012.33.
- [218] J. M. Faleiro, D. J. Abadi, FIT: A distributed database performance tradeoff, IEEE Data Eng. Bull. 38 (1) (2015) 10–17.

- [219] R. Guerraoui, M. Pavlovic, D. Seredinschi, Trade-offs in replicated systems, *IEEE Data Eng. Bull.* 39 (1) (2016) 14–26.
- [220] D. Pritchett, Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability., *Queue* 6 (3) (2008) 48–55. doi:10.1145/1394127.1394128.
- [221] J. M. Hellerstein, P. Alvaro, Keeping calm: When distributed consistency is easy, *Commun. ACM* 63 (9) (2020) 72–81. doi:10.1145/3369736.
- [222] M. Shapiro, N. M. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: X. Défago, F. Petit, V. Villain (Eds.), *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, Vol. 6976 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 386–400. doi:10.1007/978-3-642-24550-3_29.
- [223] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford, Spanner: Google’s Globally-Distributed Database, in: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, USENIX Association, 2012, pp. 261–264.
- [224] P. A. Bernstein, Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing, *Computer* 21 (2) (1988) 37–45. doi:10.1109/2.17.
- [225] J. Chen, S. Gupta, A. Sonnino, L. Kokoris-Kogias, M. Sadoghi, Resilient consensus sustained collaboratively (2024). arXiv:2302.02325.
URL <https://arxiv.org/abs/2302.02325>
- [226] Leveldb.
URL <https://dbdb.io/db/leveldb>
- [227] S. Dong, A. Kryczka, Y. Jin, M. Stumm, Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications, *ACM Trans. Storage* 17 (4) (Oct. 2021). doi:10.1145/3483840.
URL <https://doi.org/10.1145/3483840>
- [228] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, D. K. Panda, High performance rdma-based mpi implementation over infiniband, in: *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 295–304.
- [229] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, S. Shenker, Revisiting network support for rdma, in: *Proceedings of the 2018 Conference of the ACM Special Interest Group*

on Data Communication, 2018, pp. 313–326.

- [230] A. Dragojević, D. Narayanan, M. Castro, O. Hodson, {FaRM}: Fast remote memory, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014, pp. 401–414.
- [231] A. Kalia, M. Kaminsky, D. G. Andersen, {FaSST}: Fast, scalable and simple distributed transactions with {Two-Sided}({{{{RDMA}}}}) datagram {RPCs}, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 185–201.