High-Performance Coordination with Weaker Protocols: From Shared Registers to Data Feed Processing in Blockchains

by

Hao Tan

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

Waterloo, Ontario, Canada, 2025
© Hao Tan 2025

Examining Committee Membership

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

External Examiner:	Andreas G Veneris Professor, Dept. of Electrical and Computer Engineering University of Toronto
Supervisor(s):	Wojciech Golab Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal Member:	Anwar Hasan Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal Member:	Mahesh Tripunitara Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal-External Member:	Samer Al-Kiswany Associate Professor, David R. Cheriton School of Computer Science University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

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

Abstract

Blockchains are decentralized ledger technologies that provide secure, transparent, and tamper-resistant transaction records. Permissioned blockchains restrict participation to authorized entities and employ Byzantine Fault Tolerance (BFT) protocols for consensus. As blockchain systems evolve, the need for exchanging data and assets across heterogeneous networks has grown. To support such functionality, platforms increasingly integrate with external systems, enabling information flow between decentralized and conventional infrastructures. Blockchain oracles serve a key role in this integration by injecting real-world data—such as asset prices or event outcomes—into smart contracts through structured oracle data feeds.

Conventional blockchain systems typically employ BFT consensus for oracle data integration, which introduces substantial latency and computational overhead. Although consensus protocols offer robust consistency guarantees, they also incur significant computational and communication overhead, thereby constraining system responsiveness and scalability in blockchain environments. To address these challenges, this thesis investigates weaker coordination protocols as lightweight alternatives that relax strict consistency requirements while preserving sufficient reliability for practical deployment. This thesis evaluates the effectiveness of these protocols in enhancing the performance of read-write operations—a common access pattern in oracle data feed processing—across three distinct projects. First, this thesis assesses the overhead of consensus in key-value processing by comparing consensus protocols with shared register protocols. The comparison reveals two previously overlooked limitations that arise under workload-driven access patterns. Second, it introduces a shared register protocol tailored for wide-area networks, which employs loosely synchronized clocks and asymmetric quorum techniques to reduce coordination latency across heterogeneous client loads. Third, it applies weaker coordination techniques to oracle data feed processing—a fundamental component of many smart contract applications. By decoupling data feed processing from BFT consensus using a probabilistic quorum protocol and a censorship-resistant broadcast primitive, the proposed design improves both throughput and data freshness for on-chain transactions dependent on timely off-chain inputs.

Collectively, these contributions demonstrate how workload-specific weaker protocols can enable more efficient distributed systems. By reducing coordination overhead without sacrificing practical correctness, the proposed approaches support the development of scalable, responsive, and application-aware decentralized infrastructures.

Acknowledgements

First, to my parents and my wife—thank you for your unwavering support and for standing beside me during challenging times. Your constant love, encouragement, and belief in me have been a continuous source of strength throughout this journey. This accomplishment would not have been possible without your sacrifices, patience, and unconditional support.

I would like to express my sincere gratitude to my advisor, Prof. Wojciech Golab, for his unwavering guidance and support throughout my PhD studies. His insightful advice, generous mentorship, and commitment to research excellence have had a lasting influence on the direction and quality of this thesis.

I am also sincerely grateful to the members of the thesis committee: Prof. Andreas G. Veneris, Prof. Samer Al-Kiswany, Prof. Anwar Hasan, and Prof. Mahesh Tripunitara. Their insightful feedback, thoughtful questions, and constructive suggestions greatly contributed to the clarity and depth of this work.

I gratefully acknowledge my colleagues in the Distributed Algorithms and Systems Lab and at Alibaba DAMO Academy for their support, collaboration, and the intellectually stimulating research environment they provided. The insightful discussions, exchange of ideas, and strong sense of community within these groups have greatly enriched my academic experience and contributed significantly to the development of this thesis.

Finally, I am thankful for the opportunity to pursue this research journey and for everyone who has supported me along the way—through guidance, mentorship, or quiet encouragement. Each contribution, no matter how small, has helped bring this work to completion.

Dedication

This thesis is dedicated to my family.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Abstract iv

Acknowledgements v

Dedication vi

List of Figures xii

List of Tables xiv

List of Symbols xv

1 Introduction 1

1.1 Motivation 2

1.1.1 Conventional Approach 3

1.1.2 Research Question 5

1.2 Contributions 6

1.3 Thesis Organization 7

2 Background **8**
2.1 System Model . . . 8
2.2 Failure Model . . . 8
2.2.1 Channel Reliability . . . 9
2.2.2 Node Failures . . . 9
2.3 The Consensus Problem . . . 10
2.4 State Machine Replication . . . 11
2.5 Shared Objects . . . 12
2.5.1 Linearizability . . . 12
2.5.2 Shared Register . . . 13
2.5.3 Consensus Numbers . . . 14
2.6 Quorum Systems . . . 14
2.6.1 Types of Quorum Systems . . . 15
2.7 Relaxation of Quorum Systems . . . 16
2.7.1 Asymmetric Quorum . . . 16
2.7.2 Probabilistic Quorum . . . 17
2.8 Blockchain Technology . . . 17
2.8.1 Consensus in Blockchain Systems . . . 19

3 Related Works **22**
3.1 Shared Register Protocols and Distributed Storage Systems . . . 22
3.2 Consensus in Partially-synchronous Systems . . . 23
3.2.1 Crash Fault-Tolerant Consensus . . . 23
3.2.2 BFT Consensus . . . 24
3.2.3 Censorship-resilient BFT . . . 25
3.3 Consensus in Asynchronous Systems . . . 26
3.4 Benchmarking Consensus . . . 26
3.5 Blockchain Interoperability . . . 27
3.6 Blockchain Oracles . . . 28

4 Comparison Between Consensus and Shared Register **29**
4.1 Background . . . 30
4.1.1 Consensus Protocols . . . 30
4.1.2 Shared Register Protocols . . . 31
4.2 The Tale of Two Protocols . . . 31
4.2.1 ABD Algorithm . . . 32
4.2.2 EPaxos Algorithm . . . 33
4.2.3 A First Glance . . . 35
4.2.4 Diminishing Fast-Path Commit . . . 37
4.2.5 Processing Granularity . . . 38
4.2.6 The Sub-effect of Writes . . . 39
4.3 Chapter Conclusion . . . 41

5 Enhancing Shared Registers for WAN Environments **42**
5.1 Model . . . 43
5.2 Consistency Properties . . . 44
5.2.1 Monotonicity Properties . . . 44
5.3 Fast Write via Loosely Synchronized Clocks . . . 45
5.3.1 Clock Optimized ABD . . . 47
5.3.2 Analysis for CABD . . . 47
5.3.3 Discussion . . . 53
5.4 Watermark Integrated ABD . . . 54
5.4.1 Watermark-based Read/Write Algorithm . . . 55
5.4.2 Combining Watermark-based Algorithm with CABD . . . 57
5.4.3 Correctness of WABD . . . 59
5.5 Evaluation . . . 68
5.5.1 Experiment Setup . . . 68
5.5.2 Throughput . . . 69

5.5.3 Latency . . . 70
5.5.4 The Effectiveness of Mode Switching In WABD . . . 71
5.5.5 The Impact of Maximum Clock Offset . . . 72
5.6 Chapter Conclusion . . . 74

6 Dolphin: Efficient Processing of Oracle Data Feeds for Chained BFT Protocols 75

6.1 Background . . . 77
6.1.1 Chained BFT Consensus . . . 77
6.1.2 Blockchain Oracles . . . 79
6.2 Motivation . . . 79
6.2.1 Issues with Conventional Approach . . . 80
6.2.2 Data Staleness . . . 80
6.2.3 Unnecessary Replication . . . 82
6.3 Model . . . 83
6.4 Dolphin Overview . . . 84
6.5 Data Feed Processing . . . 85
6.5.1 Simulation of Data Staleness . . . 86
6.5.2 Discussion . . . 87
6.6 DDT Processing . . . 89
6.6.1 Data Structures and Utilities . . . 90
6.6.2 Block Processing . . . 93
6.6.3 Verifiably Recent Broadcast . . . 94
6.6.4 Collect Phase . . . 97
6.6.5 Consensus Phase . . . 97
6.6.6 VRB View Change . . . 98
6.6.7 Transaction Execution . . . 100
6.7 Correctness . . . 101

6.8 Evaluation 104
6.8.1 Commit Latency 105
6.8.2 Data Staleness 106
6.8.3 Censorship Attack Scenario 106
6.8.4 Transaction Processing Throughput 109
6.9 Chapter Conclusion 111

7 Conclusion and Future Works **112**
7.1 Future Works 113

References **114**

List of Figures

1.1 The conventional approach for processing oracle data feed and DDTs . . . 4

2.1 Proof of Work: Linking Blocks With Hash Pointers 19

4.1 Message Types and Parameters in the ABD Algorithm 32
4.2 Percentile Latency 36
4.3 Rate of Slow Read vs Throughput 37
4.4 Different processing granularity when handling a batch of two read operations. In ABD, `Read A` returns after one round-trip of communication while both `Read A` and `Read B` return after two round-trips of communication in EPaxos. 38
4.5 (a) In EPaxos, a write operation on object A processed by p_2 immediately updates the command log before any other replica is aware of it. As a result, a subsequent read commits on the slow path due to a conflict with the write processed by p_2; (b) In ABD, the same write operation processed by p_2 updates object A only after completing one round-trip of communication with other replicas. Consequently, a read operation coordinated by p_1 can commit on the fast path, as the write operation has not yet taken effect on p_2 when p_2 replies to p_1. 40

5.1 Local state maintained by each process for Algorithm 4 and Algorithm 5 . 54
5.2 Throughput versus mean latency, $n = 5$, $o_{max} = 10$ms. 69
5.3 Latency distribution varied by client load, $n = 5$, $o_{max} = 10$ms. 71
5.4 Impact of Clock Offset on Latency for Replication Factor $N = 3$. 72
5.5 Impact of Clock Offset on Latency for Replication Factor $N = 5$. 73

6.1 HotStuff: block chaining via quorum certificates 78
6.2 The decoupled processing of data feeds and DDTs in Dolphin 84
6.3 CDFs of data staleness from 1000 runs of Algorithm 9 with $n = 64$, $q_r = 43$, varying q_w; $\lambda_d = 100$/sec, $\lambda_q = 1$/sec. 87
6.4 2-Phase cBFT: DDT tx and data record v are committed together in block B_1 88
6.5 Dolphin: data record v is broadcast only when required by DDT tx 89
6.6 Local state maintained by each replica 90
6.7 Impact of VRB on commit latency of DDTs 105
6.8 Average data staleness (ms) versus data feed replication factor in a 64-replica system. `CA` denotes results under censorship attacks. 107
6.9 Box-and-whisker plot illustrating data staleness across different scenarios under a fully replicated data feed. 108
6.10 Saturation throughput (txn/s) and average staleness (ms) for 2K data feeds issued every 100ms by oracles 110

List of Tables

4.1 Comparing ABD with EPaxos 35

List of Symbols

M the state machine

R the set of all processes or servers

f the maximum number of faulty processes tolerated by the system

GST the global stabilization time

Δ the communication delay bound after global stabilization time

$\mathcal{Q}$ a finite set of all possible quorums in a quorum system

Q_r the quorum associated with the read operation

Q_w the quorum associated with the write operation

$inv(o)$ the invocation event of operation o

$resp(o)$ the response event of operation o

o_{max} the maximum clock offset across processes

$c(t, p)$ the local clock value at real time t on process p

$ts(o)$ the timestamp associated with operation o

$clock(ts)$ the clock component of timestamp ts

Π the proof of recency

$H(v)$ the hash value of some input v

Chapter 1

Introduction

Blockchain technology, introduced in 2008 with the launch of Bitcoin [123], emerged as the foundational infrastructure for cryptocurrencies. It enables fund transfers between mutually untrusting parties and facilitates a consistent, shared view of account balances. By ensuring data integrity, transparency, and a degree of decentralization, blockchain fosters cooperation among entities without necessitating full trust. Consequently, it is widely envisioned as a core building block for future financial and commercial infrastructure. While blockchain's mainstream recognition emerged through cryptocurrencies, its underlying foundation—Distributed Ledger Technology (DLT)—has deeper roots. Earlier research in cryptographic hash chains [79, 80, 112] and Byzantine Fault Tolerance (BFT) [100, 34, 93, 3] laid the theoretical groundwork for secure, decentralized consensus. Bitcoin popularized these concepts by combining cryptographic primitives and distributed computing techniques into a trustless, peer-to-peer system that prioritizes user autonomy and eliminates the need for central control. Alongside this decentralized model, many central banks are actively exploring Central Bank Digital Currencies (CBDCs) [62, 59, 129], a state-backed alternative that employs managed DLT to issue and regulate digital fiat currencies. CBDCs aim to enhance payment efficiency and offer new tools for implementing monetary policy, while introducing important considerations around privacy, surveillance, and the evolving role of commercial banks.

As the conceptual foundations of blockchain matured, research and industry efforts expanded rapidly to meet growing demand for more scalable, secure, and application-flexible infrastructures. A surge of work has focused on improving protocol performance [92, 75, 60, 82, 155, 77, 37, 131] and strengthening security against various attack vectors [92, 52, 83, 73, 103, 159, 139, 140]. These developments accelerated the emergence of blockchain-inspired systems that incorporate modularity, programmability, and data man-

agement techniques from modern distributed databases [69, 14, 165, 70, 15, 57, 136, 128, 138, 148, 164]. Such platforms blend the auditability and decentralization guarantees of blockchain with the performance and usability expected of enterprise data systems. This architectural evolution has shown particular promise in application domains that benefit from secure, tamper-resistant, and auditable data exchange. Prominent examples include the trade of high-value commodities [105, 160], agricultural production and supply chain tracking [120, 137], land and property registration [78], digital identity and credential management [110, 10, 74], health care data protection and interoperability [32, 47, 121], insurance claims and fraud detection [135], and decentralized energy production and peer-to-peer trading [147, 81, 11]. These applications underscore the broader utility of blockchain-based systems beyond financial use cases, supporting trustless coordination across diverse sectors that require transparency, traceability, and resilience.

Building upon these foundational innovations, Ethereum's [31] emergence extended the utility of blockchain beyond monetary transactions by introducing *smart contracts*—self-executing programs deployed on the blockchain that automatically carry out predefined instructions when specified conditions are met. Smart contracts enable decentralized applications (dApps) to operate without centralized intermediaries, facilitating transparent, verifiable, and trustless interactions across the network. This evolution gave rise to the broader vision of *Web3*, a paradigm that reimagines internet infrastructure as decentralized and user-controlled. Under Web3, dApps run on blockchain platforms or peer-to-peer networks, leveraging smart contracts and DLT to deliver services ranging from decentralized finance to content sharing, all without reliance on centralized authorities.

1.1 Motivation

A major limitation of early blockchain systems [123, 31] that hinders the development of advanced dApps is their restricted ability to obtain reliable input from external environments. These systems were originally designed as closed platforms, where interactions occur solely among participants within the same blockchain network. As a result, applications are confined to accessing only on-chain resources, such as the distributed ledger and execution engine. This architectural isolation significantly limits the expressiveness and utility of dApps, particularly for domains such as decentralized finance (DeFi), where applications replicate and extend traditional financial services—such as lending, borrowing, and derivatives—without centralized intermediaries. DeFi systems require timely, trustworthy access to external information, including asset prices, interest rates, and market conditions, which are not available through the blockchain alone.

To address these limitations, blockchain *oracles* [28, 162, 5] have been introduced as trusted services that supply smart contracts with authenticated external data. Oracles serve as essential interfaces between the blockchain and its surrounding environment, enabling smart contracts to operate on real-time, verifiable information. They retrieve, verify, and deliver data such as financial prices, weather updates, or sensor readings from off-chain sources to on-chain applications. By doing so, oracles significantly enhance the functionality and automation potential of smart contracts.

In DeFi specifically, oracles are critical for enabling core functionalities, such as accurate pricing, dynamic interest rate adjustments, and the management of liquidation thresholds [132]. Beyond finance, oracles also play important roles in domains such as supply chain tracking [95], where they validate product provenance and logistics, and in insurance [122], where they trigger claim settlements based on predefined external events.

1.1.1 Conventional Approach

Oracles provide a continuous data feed to on-chain smart contracts. To make the latest data records from external feeds accessible to on-chain transactions, the conventional method involves deploying a proxy smart contract managed by the oracle [104]. This proxy maintains a key-value store D, which maps data feed identifiers to their most recent data records. The key-value store D is integrated into the state machine M (also referred to as the world state), which is consistently synchronized across the blockchain network via the underlying BFT consensus protocol [34, 3, 93, 114, 161]. This architecture ensures that smart contracts can reliably access and process up-to-date external data within a secure and decentralized environment.

Figure 1.1 demonstrates the typical workflow:

1. The oracle retrieves the most recent data record v_1 for the data feed identified by key k_1. It then issues a transaction tx_1 that updates the data store entry $D[k_1]$ to the new value v_1, where $D[k_1]$ denotes the current on-chain value associated with data feed key k_1 in D. This transaction tx_1 is submitted to the blockchain network and processed through the BFT consensus mechanism.

2. Another client issues a data-dependent transaction (DDT) tx_2, which updates $M[k_2]$ to $g(D[k_1])$, where g denotes an arbitrary computation applied to the value stored at $D[k_1]$. Here, $M[k_2]$ represents an on-chain application state entry associated with key k_2 in M.

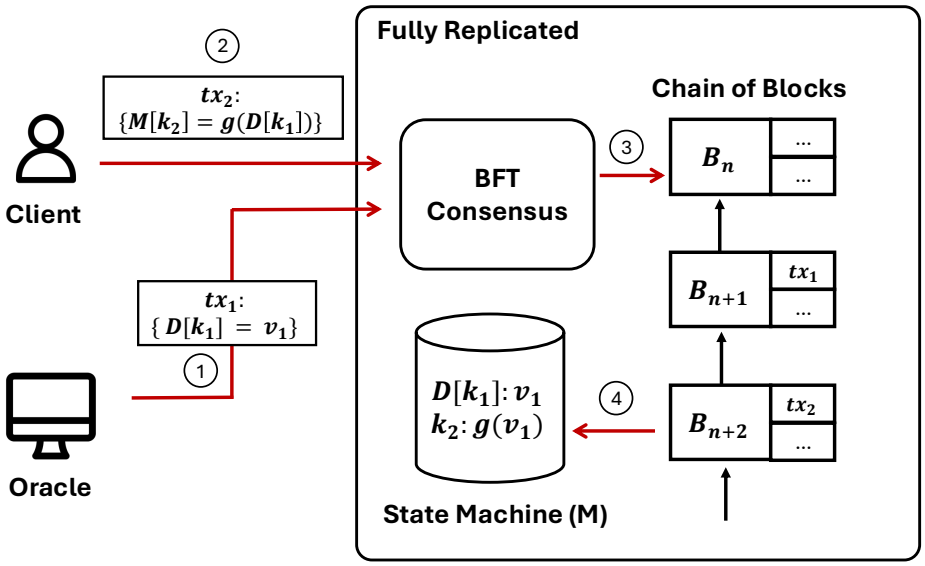


Figure 1.1: The conventional approach for processing oracle data feed and DDTs

3. Both tx_1 and tx_2 are ordered by the BFT consensus, which forms the same chain of blocks on each replica. In the example, tx_1 is included in block B_{n+1} and tx_2 is included in B_{n+2}, the successor block of B_{n+1}.

4. Each node then applies each block in the chain sequentially to the local state machine. In our example, $D[k_1]$ is updated to v_1 after B_{n+1} is applied and $M[k_2]$ is updated to the result of $g(v_1)$ after B_{n+2} is applied.

While BFT consensus protocols provide strong consistency and fault tolerance, they introduce significant performance bottlenecks when processing high-frequency data feeds in large geo-distributed networks.

1. In traditional financial applications, data feed update rates can reach up to 1 million events per second during peak periods [39]. However, conventional BFT consensus protocols typically do not meet required throughput levels. Empirical benchmarks conducted by Amiri et al. [13] show that widely used BFT systems generally sustain fewer than 100,000 transactions per second. Consequently, when integrating high-frequency oracle data (e.g., real-time price feeds or sensor updates), these limitations become increasingly problematic.

2. Another challenge arises from the fundamental latency-throughput trade-off in BFT protocol design [13]. To enhance throughput, some modern BFT variants [161, 111] reduce the number of messages per communication round, but this optimization requires additional rounds of communication among validators to achieve consensus. The increased number of rounds contributes to higher latency and greater data staleness—an unacceptable trade-off for latency-sensitive applications that demand real-time data freshness, such as financial systems.

These two challenges create a dilemma for blockchain platforms designed for handling DeFi applications. Achieving support for high volumes of data updates requires prioritizing throughput, but doing so often incurs higher latency. The resulting increase in data staleness undermines the reliability of time-sensitive oracle-driven operations.

1.1.2 Research Question

This thesis argues that the synchronous, deterministic design of BFT consensus protocols is ill-suited to the dynamic, high-frequency update patterns characteristic of oracle-driven applications. Frequent submissions from oracle services can overload the consensus mechanism, leading to increased transaction commit latency and impeding the scalability of smart contract platforms that rely on BFT protocols.

While consensus protocols deliver strong safety and liveness guarantees, they do so at the cost of increased complexity, latency, and resource consumption [13, 12, 51]. These protocols offer a general-purpose, flexible interface that supports a wide range of transactional and coordination workloads, while enforcing strong consistency models such as linearizability, ensuring all participants observe a globally agreed-upon order of operations.

In contrast, weaker coordination mechanisms, such as shared registers, provide a more lightweight and efficient alternative, often achieving higher throughput and lower latency under benign conditions [30]. However, these protocols typically expose limited interfaces—such as `get` and `put`—restricting their applicability to specialized scenarios. Moreover, weaker protocols may relax consistency guarantees to improve performance; for example, probabilistic quorum systems (PQS) can introduce inconsistencies and may return stale or outdated data to clients, particularly under certain failure conditions.

These trade-offs motivate a central research question: *Can a more efficient blockchain be architected by selectively leveraging workload-specific, weaker coordination primitives, while achieving a favorable trade-off among consistency, fault tolerance, and performance?*

1.2 Contributions

This thesis investigates how workload-specific weaker coordination protocols can enhance blockchain performance. The main contributions of this thesis are summarized as follows:

Comparison Between Consensus and Shared Register:
The study begins by analyzing the performance overhead of consensus protocols in key-value (KV) processing and contrasting them with shared register protocols. It identifies two commonly overlooked limitations of consensus mechanisms: coarse-grained processing, which serializes unrelated operations, and the immediate side effects of write operations, which propagate state changes across the network. These inefficiencies lead to shared register protocols achieving half the latency and up to 25× the throughput of consensus protocols, particularly under high client loads. This analysis offers deeper insight into how workload-agnostic consensus protocols can hinder system performance when applied to workload-specific access patterns.

Enhancing Shared Registers for WAN Environments:
Subsequently, the thesis introduces a more efficient shared register protocol tailored for wide-area network (WAN) environments. Leveraging loosely synchronized clocks and asymmetric quorum techniques, the proposed design reduces communication overhead and improves performance across varied client-load scenarios. Specifically, the work presents an optimization that exploits bounded clock offsets to eliminate a coordination round in write operations by waiting through a calculated uncertainty period. This optimization is implemented in WABD, a protocol that enhances the classic Attiya-Bar-Noy-Dolev (ABD) algorithm [18] by integrating local read techniques inspired by prior research. Through adaptive mode switching, WABD exhibits latency comparable to ABD under low client workloads, while achieving throughput up to 1.6 times higher than ABD under heavy workloads, thereby balancing system efficiency and robustness.

Efficient Oracle Data Processing for Blockchain:
Finally, this thesis extends the application of weaker protocols to the domain of oracle data feed processing. To allow on-chain transactions to access timely and up-to-date data from external oracles, the thesis introduces *Dolphin*, a novel blockchain system that decouples oracle data handling from the chained BFT consensus (cBFT) layer. A key feature of Dolphin is its censorship resilience: the ability to ensure that valid data records from oracle sources are not indefinitely suppressed or ignored by adversarial nodes. To achieve this, Dolphin combines a probabilistic quorum protocol with a censorship-resistant broadcast mechanism, significantly improving both the throughput and freshness of on-chain transactions that rely on external data feeds. Compared to the state-of-the-art cBFT protocol,

Dolphin attains up to twice the throughput of existing protocols, simultaneously reducing data staleness by 40% during the processing of transactions dependent on data feeds.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides background on consensus protocols, quorum systems, and blockchain technology. Chapter 3 reviews related work pertinent to this thesis. Chapter 4 compares consensus and shared register protocols, analyzing their respective strengths and weaknesses in managing key-value workloads. Chapter 5 introduces WABD, an efficient and adaptive shared register protocol designed to handle KV workloads in wide-area network environments. Chapter 6 details the design of Dolphin, a system for efficient oracle data feed processing within chained BFT protocols. Finally, Chapter 7 presents the conclusion and outlines directions for future research.

Chapter 2

Background

This chapter introduces foundational concepts essential to this thesis, including consensus protocols, state machine replication, shared objects, quorum systems, and blockchain technology. These concepts form the theoretical basis for understanding how distributed systems achieve agreement, preserve consistency, tolerate faults, and maintain reliability in decentralized settings.

2.1 System Model

A distributed system comprises a set of computing nodes, each representing a distinct process or server, communicating via message exchanges over communication channels, which may be reliable or unreliable. The system model relies on assumptions about message delivery times and process execution speeds, classifying the system as **synchronous**, **partially synchronous**, or **asynchronous**. In synchronous systems, upper bounds on message delays, denoted Δ, and process speed variations are known. Asynchronous systems, in contrast, assume no such bounds. The partially synchronous model represents an intermediate case, where such bounds exist but are either unknown or hold only after an unknown global stabilization time (GST).

2.2 Failure Model

In distributed systems, failures affect both *communication channels* and *computing nodes*, impacting consistency, coordination, and fault tolerance.

2.2.1 Channel Reliability

A *communication channel* enables message exchange between distributed processes. However, network conditions, protocol limitations, and system failures can introduce unreliability. A channel is *unreliable* if it fails to guarantee message integrity, delivery, or order. Key failure types:

1. **Message Loss**: Messages may be dropped due to network congestion, hardware faults, or routing issues, common in protocols like UDP.
2. **Message Corruption**: Transmission errors—caused by bit flips, signal interference, or faulty hardware components—can corrupt messages, potentially resulting in
3. **Message Duplication**: Retransmission mechanisms may cause duplicate messages, leading to redundant computations or incorrect state transitions.
4. **Message Reordering**: Variable latency due to congestion, queuing, or routing can delay some messages more than others.
5. **Spurious Messages**: Erroneous or adversarially injected messages may disrupt system behavior, particularly in security-critical environments.

2.2.2 Node Failures

According to [56], node failures can be classified into the following types:

1. **Fail-stop failure**: A process crashes and ceases to send or receive messages indefinitely.
2. **Omission failure**: A process fails to send or receive messages when required.
3. **Byzantine failure**: Also known as arbitrary failure, this occurs when a node deviates from its assigned algorithm. A Byzantine node may either halt prematurely (crash) or exhibit unpredictable behavior, including sending incorrect or misleading messages. If the deviation is intentional, the failure is termed *malicious*.

2.3 The Consensus Problem

The consensus problem is a core challenge in distributed computing, especially in systems requiring strong consistency and availability. It plays a central role in replication, where multiple processes must agree on a single value to ensure a consistent system state. Informally, consensus refers to the ability of a group of processes to agree on one of the values proposed by participants, despite failures or message delays. The Byzantine consensus problem is formally defined as follows:

Definition 1 (Consensus Problem)**.** Let V be a set of input values. The consensus problem is defined by a partial function

$$c : V^n \rightharpoonup V$$

which maps a vector of n input values to a unique, agreed-upon output value. Let $R = \{p_1, \ldots, p_n\}$ be a set of n processes, where each process $p_i \in R$ is initially assigned an input value $x_i \in V$. A process is considered *correct* if it participates in the execution without experiencing a crash or exhibiting Byzantine behavior; otherwise, the process is classified as faulty. Each correct process must irrevocably decide on an output value $y_i \in V$, consistent with the result of $c(x_1, \ldots, x_n)$, subject to the following conditions:

1. **Termination (Liveness)**: Every correct process $p_i \in R$ eventually decides on some value $y_i \in V$.
2. **Agreement (Safety)**: If any two correct processes $p_i, p_j \in R$ with $i \neq j$ both decide, then their outputs must be equal: $y_i = y_j$.
3. **Validity**: If all correct processes have the same input $v \in V$, and at least one correct process decides, then the decision value must be v.

The consensus problem has been extensively studied in prior work [98, 119, 124, 34, 3, 113, 161, 93, 27] under various assumptions about failure models and system synchrony. Let f denote the maximum number of faulty processes that the system is required to tolerate. Dwork et al. [56] proved that, in partially synchronous systems, solving the consensus problem requires at least $n \geq 2f + 1$ processes to tolerate f crash (non-Byzantine) failures, and at least $n \geq 3f + 1$ to tolerate f Byzantine failures. According to the impossibility result by Fischer, Lynch, and Paterson [63], no deterministic algorithm can solve the consensus problem in a purely asynchronous system, even when processes are subject only to crash failures.

2.4 State Machine Replication

Definition 2 (State Machine)**.** A *state machine* is a tuple $M = (S, s_0, \mathcal{C}, \mathcal{O}, A)$, where S is a set of states, $s_0 \in S$ is the initial state, $\mathcal{C}$ is a set of input commands, $\mathcal{O}$ is a set of outputs, and $A : S \times \mathcal{C} \to S \times \mathcal{O}$ is a deterministic transition function mapping a state and a command to the next state and corresponding output.

Definition 3 (Replicated State Machine)**.** A *Replicated State Machine* (RSM) consists of n identical deterministic state machines $\{M_1, \ldots, M_n\}$, where each $M_i = (S, s_0, \mathcal{C}, \mathcal{O}, A)$ is a state machine. All correct replicas reach agreement on a function

$$\ell : \mathbb{N} \to \mathcal{C}$$

that specifies a global sequence of input commands. Agreement on each entry $\ell(i)$, for every $i \in \mathbb{N}$, is established consecutively via repeated invocations of a consensus protocol. For each index $i \in \mathbb{N}$, every correct replica applies the command $\ell(i)$ using the transition function A, updates its local state accordingly, and produces the corresponding output. The RSM abstraction guarantees that all correct replicas apply the same sequence of commands in the same order, resulting in identical state transitions and outputs.

State Machine Replication (SMR) is a foundational technique in distributed systems for achieving fault tolerance, high availability, and strong consistency. An SMR protocol ensures that all correct replicas apply the same sequence of commands in a consistent order, thereby maintaining system-wide consistency, despite the presence of up to f faulty replicas exhibiting crash or Byzantine behavior in a system of n replicas. The system's externally observable behavior must remain indistinguishable from that of a single, fault-free centralized server processing client requests. An SMR protocol satisfies the following properties:

1. **Safety**: For every index $i \in \mathbb{N}$, if a correct replica decides the assignment $\ell(i) = \mathsf{cmd}$ for some $\mathsf{cmd} \in \mathcal{C}$, then no other correct replica decides $\ell(i) = \mathsf{cmd}'$ for any $\mathsf{cmd}' \in \mathcal{C}$ such that $\mathsf{cmd}' \neq \mathsf{cmd}$.
2. **Liveness**:
   (a) **Progress**: If a command $\mathsf{cmd} \in \mathcal{C}$ is proposed by any correct replica, then there exists an index $i \in \mathbb{N}$ such that $\ell(i)$ is eventually decided as cmd by all correct replicas.

(b) **Prefix Completion**: If $\ell(i)$ is decided by any correct replica, then every $\ell(j)$ with $j < i$ is eventually decided by all correct replicas.

Consensus protocols [98, 124, 34] are fundamental to implementing SMR protocols, as they coordinate multiple processes to agree on the order of client-issued commands, even in the presence of failures.

2.5 Shared Objects

In distributed computing, processes may interact by accessing *shared objects* within a *shared-memory model* [19]. Shared objects simplify coordination by providing a uniform access interface to shared state. They allow multiple processes to read, update, or synchronize on a common resource. Examples of shared objects include registers, queues, stacks, and more advanced data types.

A *shared object* is an abstraction that encapsulates a data structure or service, providing a set of operations that may be invoked by concurrent processes. Each *operation* is initiated by a single process and is associated with an *invocation* event and, optionally, a matching *response* event, arising from the process's interaction with the shared object. The invocation marks the moment the process issues a request to execute a specific command (e.g., `read`, `write`, `enqueue`), along with any required input arguments. If the operation completes, the response marks the moment the shared object returns the outcome of the command to the same process. In some cases, an operation may remain incomplete and lack a corresponding response event, due to failure, crash, or indefinite delay.

2.5.1 Linearizability

Linearizability [86] is a fundamental correctness condition for concurrent objects in distributed and shared-memory systems. It requires that each operation on a shared object appear to take effect instantaneously at some point between its invocation and response, regardless of concurrent executions.

Let an operation op consist of an invocation event $inv(\mathsf{op})$ and, if completed, a corresponding response event $resp(\mathsf{op})$, each associated with real-time timestamps. Given two operations op and op', define the real-time precedence relation $\mathsf{op} \prec_H \mathsf{op}'$ if $resp(\mathsf{op}) < inv(\mathsf{op}')$, indicating that op completes before op' begins in real time. Operations op and

op' are said to be *concurrent*, denoted as $\mathsf{op} \parallel_H \mathsf{op}'$, if neither $\mathsf{op} \prec_H \mathsf{op}'$ nor $\mathsf{op}' \prec_H \mathsf{op}$ holds.

A *history* H is a finite sequence of invocation and response events, capturing the execution timeline of one or more processes accessing shared objects. A history H is said to be *sequential* if it consists of alternating invocation and matching response events, such that each invocation is immediately followed by its corresponding response and no two operations are concurrent.

Definition 4 (Linearizability)**.** A history H is *linearizable* if there exists an extension H' of H formed by appending response events to pending operations or removing incomplete ones, such that H' can be transformed, by reordering events, into a sequential history H_S that satisfies the following two properties:

1. **Real-time order preservation**: Let $\mathsf{op}, \mathsf{op}'$ be complete operations whose invocation and response events appear in H'. If $resp(\mathsf{op}) < inv(\mathsf{op}')$, then $\mathsf{op} \prec_{H_S} \mathsf{op}'$ in the sequential history H_S.
2. **Sequential specification compliance**: The sequential history H_S must be legal according to the object's sequential specification, meaning it represents a valid execution in which operations occur one at a time, starting from a well-defined initial state.

Linearizability provides a strong and intuitive consistency model: it allows clients to reason about concurrent operations as if they were executed in some global, total order that respects real-time constraints. This property is particularly important for ensuring correctness and safety in distributed systems, as it enables atomic reasoning about shared state, even in the presence of concurrency and partial failures.

2.5.2 Shared Register

A *shared register* is an abstract object that supports two operations: `Read()` and `Write(`v`)`. It is accessed concurrently by multiple processes either in a shared-memory model or through emulation over a message-passing network [19].

Definition 5 (Atomic Shared Register)**.** A *shared register* maintains a value $v \in V$, where V is a predefined domain of values. It supports the following operations:

- `Write(`v`)`: updates the register's value to the input $v \in V$.

- `Read()`: returns the value currently stored in the register.

An *atomic shared register*, also called a *linearizable register*, is a shared register whose operations appear to take effect atomically at some instant between their invocation and response. This implies that all operations can be totally ordered in a manner that respects the real-time order of non-concurrent operations. Each `Read()` returns the value written by the most recent preceding `Write()` in this total order, or the initial value if no such `Write()` exists.

In distributed settings, emulating a shared register over unreliable or asynchronous networks requires carefully designed algorithms to ensure consistency and availability. A notable example is the ABD algorithm [18], which implements a linearizable read/write register in an asynchronous message-passing system with crash failures.

2.5.3 Consensus Numbers

To characterize the synchronization power of shared objects, Herlihy introduced the notion of *consensus number* [84].

Definition 6 (Consensus Number)**.** The *consensus number* of a shared object type `T` is the largest number of processes n for which it is possible to solve the consensus problem in an asynchronous system, using any number of objects of type `T` along with atomic shared registers. If consensus can be solved among any number of processes using `T`, then `T` is said to have consensus number ∞ and is referred to as *universal*.

The consensus number establishes a hierarchy of shared objects based on their coordination power. Atomic read/write registers have consensus number 1: they can solve consensus for only a single process. More expressive synchronization primitives differ in strength. For instance, test-and-set (TAS) has consensus number 2, enabling consensus among up to two processes. In contrast, compare-and-swap (CAS) is universal—it has consensus number ∞ and can solve consensus among any number of processes.

2.6 Quorum Systems

Quorum systems are fundamental in distributed computing, providing a fault-tolerant mechanism for coordination in consensus protocols (e.g., Paxos [98], PBFT [34]) and distributed storage systems. A quorum system is a collection of subsets, called quorums, over

a set of servers (or processes), such that the quorums satisfy certain intersection properties. These properties are designed to ensure consistency and availability, even in the presence of failures.

Definition 7 (Quorum System)**.** A *quorum system* is defined over a finite set of servers (or processes) R. It consists of a non-empty collection $\mathcal{Q} \subseteq 2^R \setminus \{\emptyset\}$, where each element $Q \in \mathcal{Q}$ is called a *quorum*.

1. **Intersection Property**: For all quorums $Q_1, Q_2 \in \mathcal{Q}$,

$$Q_1 \cap Q_2 \neq \emptyset.$$

   This ensures that any two quorums share at least one common server, enabling consistent coordination.

2. **Availability Condition** (for fault tolerance): The quorum system should be *resilient* to failures, meaning that despite some faulty servers, at least one quorum remains functional.

3. **Minimality (Optional)**: No quorum in $\mathcal{Q}$ should be a strict superset of another. That is, removing any element from a quorum should violate the quorum property.

Note that in systems supporting shared data access, such as shared registers, quorum systems often distinguish between *read quorums* and *write quorums*. This separation imposes more specialized intersection requirements than the uniform quorum model commonly used in consensus protocols. Further discussion is provided in Section 2.7.1.

2.6.1 Types of Quorum Systems

- **Majority Quorum:** A widely used quorum mechanism in crash fault-tolerant systems, where each quorum contains at least $f + 1$ servers:

$$|Q| \geq f + 1$$

  This guarantees that any two quorums intersect in at least one server, provided the system satisfies $|R| \geq 2f + 1$, where f is the maximum number of crash faults tolerated. This intersection property is essential for preserving consistency in the presence of failures. In crash fault-tolerant protocols such as Paxos, majority quorums ensure that decisions by non-faulty processes remain visible and accessible to future operations, thereby ensuring both safety and liveness.

- **Byzantine Quorum:** A quorum system designed to tolerate Byzantine faults, ensuring consistency despite the presence of arbitrary or malicious behaviors. Byzantine quorum systems are fundamental to BFT consensus protocols, where quorum intersections must include enough correct replicas to prevent equivocation and support agreement. In standard BFT systems, such as PBFT [34] and HotStuff [161], the system consists of $|R|$ replicas and tolerates up to f Byzantine faults, provided that $|R| \geq 3f + 1$. To meet this requirement, quorums are formed with size:

  $$|Q| \geq 2f + 1$$

  This guarantees that any two quorums intersect in at least $f + 1$ replicas, ensuring that at least one correct replica is always present in the overlap. This intersection is crucial for agreement on committed values and for defending against conflicting actions by faulty nodes.

- **Grid Quorum:** Organizes servers in an $n \times n$ grid where each quorum consists of a full row and a full column. Consensus protocol that uses this type of quorum include CMT Paxos [156].

2.7 Relaxation of Quorum Systems

Traditionally, quorum systems require strict intersection properties for correctness. However, to optimize performance and scalability, variants such as asymmetric and probabilistic quorum systems have been introduced. These approaches relax traditional quorum constraints, enabling better efficiency while maintaining essential correctness properties.

2.7.1 Asymmetric Quorum

An *asymmetric quorum* system is a generalization of quorum systems where different operations, such as read and write, use distinct quorum sets with potentially different sizes. Asymmetric quorum systems enable optimized performance by allowing smaller read quorums while ensuring consistency through larger write quorums. They are widely used in distributed storage systems such as quorum-based replicated databases.

Definition 8 (Asymmetric Quorum System)**.** Let $\mathcal{Q}_r$ and $\mathcal{Q}_w$ be the sets of read and write quorums, respectively. The asymmetric quorum system satisfies:

1. **Read-Write Intersection** Any read quorum $Q_r \in \mathcal{Q}_r$ and any write quorum $Q_w \in \mathcal{Q}_w$ must intersect to ensure consistency:

$$\forall Q_r \in \mathcal{Q}_r, Q_w \in \mathcal{Q}_w, \quad Q_r \cap Q_w \neq \emptyset.$$

2. **Write-Write Intersection** Any two write quorums must intersect to prevent conflicting updates:

$$\forall Q_{w_1}, Q_{w_2} \in \mathcal{Q}_w, \quad Q_{w_1} \cap Q_{w_2} \neq \emptyset.$$

2.7.2 Probabilistic Quorum

A *probabilistic quorum system* is a relaxation of the classical quorum system model, allowing quorum intersection to hold with high probability rather than deterministically. Probabilistic quorum systems are used in large-scale distributed systems to reduce the overhead of ensuring deterministic quorum intersections.

Definition 9 (Probabilistic Quorum System)**.** A probabilistic quorum system satisfies the *probabilistic intersection* property, which requires that for any two quorums $Q_1, Q_2 \in \mathcal{Q}$, the probability that they intersect in at least one correct server is at least $1 - \epsilon$, for some $\epsilon \in \mathbb{R}^+ \cap (0, 1)$:

$$\forall Q_1, Q_2 \in \mathcal{Q}, \quad \Pr[Q_1 \cap Q_2 \cap R_c \neq \emptyset] \geq 1 - \epsilon,$$

where $R_c \subseteq R$ denotes the set of correct (non-faulty) servers.

2.8 Blockchain Technology

The term "blockchain" was initially introduced to describe the technology underlying Bitcoin [123]. Fundamentally, a blockchain is an append-only, tamper-resistant ledger structured as a linked list of blocks, where each link is secured via a cryptographic hash pointer. Each block contains a batch of transactions that document the transfer of digital assets. While blockchain technology originated with cryptocurrencies, its application has since expanded to encompass a broader class of distributed systems. These systems allow mutually untrusting parties to agree on a shared transaction history without depending on centralized authorities, thereby enabling decentralized trust.

Bitcoin [123] is widely acknowledged as the first practical implementation of a blockchain system. In Bitcoin, transactions transfer tokens from a sender to one or more recipients.

Each transaction includes inputs and outputs: inputs reference unspent transaction outputs (UTXOs) from previous transactions, while outputs designate token amounts assigned to recipients. A valid transaction must preserve balance by ensuring that the total value of inputs equals the sum of outputs and an transaction fee, which incentivizes miners to include the transaction in a block. *Miners* validate pending transactions, group them into blocks, and append these blocks to the blockchain using Nakamoto consensus, which relies on a proof-of-work (PoW) mechanism. This protocol incentivizes miners to extend the longest valid chain and maintain safety, under the assumption that honest participants control the majority of the network's computational power.

Ethereum [31] builds upon Bitcoin's architecture by supporting *smart contracts*—Turing-complete programs that define arbitrary state transition functions and execute autonomously on-chain. Contracts are written in high-level languages such as Solidity [157], and compiled into Ethereum bytecode executed by the Ethereum Virtual Machine (EVM)—a deterministic, stack-based runtime with 1024 slots, each capable of storing 256-bit words. The EVM enforces gas metering by assigning a predefined cost to each opcode, and deducting gas from the transaction sender's balance to prevent resource exhaustion. Ethereum employs an account/balance model in which contract states and account balances are globally replicated across all full nodes. Smart contracts enable automated fund transfers, conditional execution, and verifiable coordination among parties without relying on trust. Ethereum originally utilized Nakamoto consensus with proof-of-work (PoW), but transitioned to a proof-of-stake (PoS) mechanism to enhance energy efficiency and finality [31].

Hyperledger Fabric [14], developed by IBM and the Linux Foundation, is a modular permissioned blockchain that adopts an execute-order-validate architecture. It supports smart contracts in Java and C++, and achieves consensus using BFT protocols such as PBFT [34]. Fabric is widely adopted for its scalability and flexible trust configuration. R3 Corda [133] targets the financial sector with a point-to-point architecture that avoids global broadcast. It uses a UTXO-like model and relies on notary nodes to guarantee uniqueness and prevent double-spending. Contracts written in Kotlin or Java support workflows that align with legal and regulatory requirements. ConsenSys Quorum [41], an enterprise variant of Ethereum, adds permissioned access, privacy controls, and support for PoA-based consensus such as IBFT [118] and Clique [127]. Tools like Tessera enable confidential smart contracts and private transactions, making Quorum well suited for sensitive enterprise applications. FISCO BCOS [64], developed under WeBank's leadership, is a widely adopted platform in China. It offers pluggable consensus mechanisms, high transaction throughput, and robust privacy features, addressing the needs of financial and public-sector deployments.

2.8.1 Consensus in Blockchain Systems

The primary function of consensus protocols in blockchain systems is to ensure that all participating nodes agree on a common history of transactions.

Nakamoto Consensus

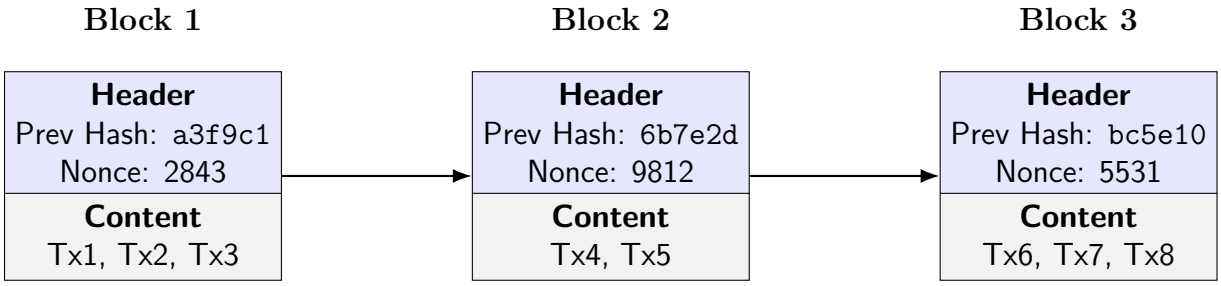


Figure 2.1: Proof of Work: Linking Blocks With Hash Pointers

The Nakamoto consensus protocol underpins the Bitcoin cryptocurrency. It organizes transactions into blocks, each cryptographically linked to its predecessor via a Proof-of-Work (PoW) mechanism, as illustrated in Figure 2.1. PoW involves solving a computational puzzle: finding a nonce such that, when hashed together with the block's transactions and the previous bloc's hash, the resulting output (e.g., using SHA-256) contains a specified number of leading zero bits. The difficulty of the puzzle increases exponentially with the number of required zero bits, ensuring security through computational effort. However, block validity remains easy to verify via a single hash computation.

Network delays and adversarial behavior can cause forks in the blockchain, where multiple blocks reference the same parent. To resolve such divergence, nodes adopt the longest valid chain they observe as the canonical ledger. While temporary forks may arise from asynchronous communication or adversarial mining, the protocol ensures eventual consistency, provided that honest nodes control a majority of the network's total hashing power. This fork-resolution mechanism is a defining feature of Nakamoto consensus, which relies on Proof-of-Work (PoW) to establish a probabilistic notion of agreement. Unlike classical Byzantine Fault Tolerant (BFT) protocols [34, 23, 116, 21, 53], which assume a known set of participants and use deterministic voting to maintain safety under up to one-third Byzantine faults, Nakamoto consensus reframes the fault threshold in terms of computational power. Safety is preserved as long as honest nodes possess more than 50% of the

total PoW capacity, making the system resilient without requiring fixed membership or direct coordination among participants.

Alternative Consensus Mechanisms

Nakamoto consensus presents two key limitations: 1) High energy consumption resulting from Proof-of-Work (PoW) mining, and 2) An approximate 10-minute block interval, designed to minimize forking but inherently limiting transaction throughput. To address these issues, several alternative consensus mechanisms have been developed.

Proof of Stake (PoS), as adopted by Ethereum [31], eliminates the need for energy-intensive mining by selecting validators based on their cryptocurrency holdings. Participants with larger stakes are more likely to be chosen by the protocol to propose blocks, under the assumption that significant financial investment aligns their incentives with the network's integrity. PoS protocols typically divide time into slots, each assigning a randomly selected validator the responsibility to propose a block. A committee of validators subsequently verifies the proposed block to ensure its validity.

Proof of Elapsed Time (PoET) [106], introduced by Intel in 2016, simulates PoW mining using a Trusted Execution Environment (TEE) [45]. Rather than performing actual computation, PoET relies on trusted hardware to enforce randomized wait times, thereby reducing energy consumption.

Proof of Retrievability (PoR) [91], also known as proof of space, was initially developed for secure peer-to-peer file storage. It uses challenge-response protocols to verify that a node retains a retrievable copy of a designated file. PoR later served as the consensus mechanism in systems like Permacoin [115], where storage replaces computation as the resource of value.

Classical BFT Consensus Protocols

Proof-of-X consensus schemes are widely adopted in permissionless blockchains due to their resilience against open participation and adversarial behavior. In contrast, permissioned blockchains often employ classical BFT protocols for their consensus layers, leveraging the performance benefits made possible by a known set of participants. Although classical BFT protocols are well-understood and effective in small, controlled environments, they face limitations in scalability due to high communication complexity. As the number of participants increases, message overhead becomes a significant performance bottleneck.

To mitigate this, many blockchain systems restrict the consensus process to a smaller committee. For instance, Algorand [75] employs verifiable random functions (VRFs) to select a temporary committee that executes classical BFT consensus, enhancing scalability without compromising security. Similarly, recent platforms [92, 96] adopt hybrid designs that combine Proof-of-X schemes with BFT protocols. These approaches typically use Proof-of-X to elect a rotating subset of representative nodes that participate in each BFT consensus round, striking a balance between security, efficiency, and decentralization.

Chapter 3

Related Works

This chapter presents additional related works that complement the core concepts and literature reviewed in the preceding chapter.

3.1 Shared Register Protocols and Distributed Storage Systems

The ABD algorithm [18, 108] establishes that fault-tolerant atomic shared read/write registers can be implemented in asynchronous systems. Dutta et al. [55] prove that achieving atomicity with both fast reads and fast writes (in a single round-trip time) is impossible. Their follow-up work [54] explores conditions on the number of concurrent readers or writers under which fast operations may still be feasible. Additionally, Aguilera et al. [6] demonstrate that even for read-write registers with dynamic participant sets, consensus can be avoided.

Attiya et al. [20] establish a lower bound on the worst-case latency for read/write operations needed to guarantee linearizability, assuming bounded message delays. They also introduce a fast write/slow read shared register protocol that differs from ours in two important respects: (1) Their algorithm implicitly assumes that a process can always complete a broadcast to all non-failed processes before crashing, an assumption our protocol does not make. (2) Their solution depends on perfectly synchronized clocks, while our approach operates under looser assumptions with loosely synchronized clocks and tolerance for unknown message delays.

Gryff [30] combines shared register protocols with state machine replication to implement shared objects. It employs the multi-writer ABD algorithm for handling reads and writes, and EPaxos for read-modify-write operations. This hybrid approach invokes consensus only when necessary.

Spanner [43] is a geo-distributed storage system that provides external consistency for read-write transactions, ensuring that transaction effects respect commit-time ordering. Spanner requires clocks within a cluster to be synchronized to within a few milliseconds. Each read-write transaction waits for an uncertainty period to ensure that subsequent transactions receive larger commit timestamps. WABD applies a similar principle to enforce monotonicity of timestamps assigned to values for writes on the same key. Compared to Spanner, WABD is limited to single-key read and write operations, whereas Spanner supports multi-key read-write transactions. Due to interface differences, Spanner assigns timestamps via Paxos-based consensus, enabling coordination-free read-only transactions once timestamps are established. In contrast, WABD assigns timestamps locally but still requires a two-round coordination process for read operations.

3.2 Consensus in Partially-synchronous Systems

State Machine Replication (SMR) [97] is a well-established technique to build fault-tolerant distributed systems. Consensus protocols realize replicated state machines by coordinating correct replicas to execute the same sequence of deterministic commands.

3.2.1 Crash Fault-Tolerant Consensus

Paxos [98], proposed by Leslie Lamport, is one of the most influential and widely used protocols. Paxos tolerates up to f non-Byzantine failures with $2f + 1$ processes. Basic Paxos, also known as single-decree Paxos, coordinates a collection of processes to choose a single value such as a command proposed by a client. Although being able to choose a single value can hardly be useful in practice, basic Paxos is the building block of Multi Paxos, which can be used to implement an RSM in a production environment. In basic Paxos, there are three types of processes: proposer, acceptor, and learner. The proposer sends a proposed value v to a quorum of acceptors. An acceptor may accept v if certain conditions are met. A learner learns the decided value. Paxos is a two-phase algorithm for its normal operations (no failures). In the first phase, the proposer contacts the acceptors to (a) learn of any value that may have already been chosen in earlier rounds and (b) prevent any new

values from being chosen in later rounds. In the second phase, the proposer propagates its value or the value learned from the first phase to all acceptors.

Many works have been proposed to optimize Paxos from different aspects. Mencius [113] is a variant of Paxos that rotates the leader for each command to distribute the load evenly across replicas. Mencius also addresses the issue of unevenly distributed client load across replicas. It allows a replica with a low client load to voluntarily skip its leader term to favor replicas with a higher client load. Mencius is not completely leaderless and skipping a leader term cannot help a replica with a high client load to utilize network resources at a replica with a low client load. EPaxos [119] further improves scalability and reduces commit latency by removing the role of the leader. Each client can send the request to the nearest replica, which is referred to as the command leader. Committing non-conflicting commands requires a single round-trip between the command leader and a super-majority of all replicas. A conflicting command requires an extra round-trip of communication for replicating dependencies of conflicting commands to a majority of replicas.

Raft [124] is another consensus protocol used by many real-world systems [44, 130]. It is proposed to address the difficulty of implementing and understanding Paxos protocols. Raft is a leader-based protocol, where a process acts as the leader while others are followers. During the leader's term, all client requests are ordered by the leader and consistently replicated to followers. The novelty of the Raft algorithm mostly lies in its leader election mechanism. A leader election is triggered when a follower times out after waiting for a heartbeat from the leader. The follower becomes a candidate and starts collecting votes from other followers. The follower becomes the new leader when it collects votes from a majority of replicas.

3.2.2 BFT Consensus

The Byzantine agreement problem, introduced by Lamport [100], has been widely studied, leading to the development of numerous Byzantine Fault Tolerant (BFT) protocols [34, 93, 3]. Among BFT protocols, PBFT [34] is one of the most widely recognized and has become almost synonymous with Byzantine fault tolerance. PBFT follows a leader-based approach, consisting of two core components: normal operation and view change. In normal operation, a client submits a request to the primary replica, which broadcasts a `Pre-Prepare` message to all replicas. Upon receiving the `Pre-Prepare`, replicas broadcast `Prepare` messages to one another. After collecting $2f + 1$ matching `Prepare` messages, a replica is ready to commit and issues `Commit` messages. A replica executes the request once it receives $2f + 1$ matching `Commit` messages and replies to the client. This process

incurs a message complexity of $O(n^2)$ in the normal operation phase for n replicas. In the event of a primary failure, PBFT triggers a view change to restore liveness. Backup replicas that detect the failure via timeout broadcast `View-Change` messages, which include information about pending requests and their current view number. When a replica receives $2f$ matching `View-Change` messages, it designates the next pre-assigned replica, based on a round-robin schedule, as the new primary. The new primary then broadcasts a `New-View` message to all replicas, containing the collected `View-Change` messages and any outstanding requests. Once replicas receive the `New-View` message, they resume normal operation under the new primary.

Numerous subsequent BFT protocols have been proposed to improve the efficiency of PBFT, including Quorum/Update (QU) [3], Hybrid Quorum (HQ) [46], Zyzzyva [94], FaB [114], Spinning [154], Robust BFT SMR [38], and Aliph [21]. To overcome the scalability, complexity, and fairness limitations of classical BFT protocols, a new class of chained BFT (cBFT) protocols has emerged, including HotStuff [161, 111] and its extensions [48, 66], enabling more scalable and efficient blockchain systems.

HotStuff [161] is a pioneering cBFT protocol that achieves optimal responsiveness, where message delays are determined solely by network latency rather than adversarial behavior. It incorporates cryptographic techniques such as threshold signatures to reduce the message complexity of view changes from quadratic to linear. In addition, HotStuff introduces a pipelined architecture that allows multiple protocol phases to share quorum certificates, leading to improved throughput and responsiveness. The Dolphin system proposed in this thesis builds upon this cBFT-based design, with a specific focus on enhancing the efficiency of oracle data feed replication and reducing data staleness in data-driven transactions (DDTs).

3.2.3 Censorship-resilient BFT

To address censorship attacks, cBFT protocols employ rotating leaders to ensure fairness [161, 111]. Narwhal [48] introduces additional communication rounds before cBFT consensus to ensure a portion of proposed transactions originate from correct replicas, but this technique increases data staleness for DDTs. Dolphin overcomes these limitations by leveraging on-demand VRB, which limits the power of malicious leaders without requiring additional communication rounds beyond those needed by cBFT. Some approaches address censorship attacks using cryptographic encryption and decryption schemes [116, 17]. However, a faulty leader can still selectively propose transactions based on accessible transaction metadata [85].

3.3 Consensus in Asynchronous Systems

FLP impossibility [63] proves that consensus cannot guarantee termination in an asynchronous environment with even one crash failure. However, this limitation can be practically circumvented by randomization and probabilistic termination. The basic idea is that a process makes a random choice when there are not enough messages received for making a final decision. Bracha [26] proposed a reliable broadcast (RBC) primitive and an asynchronous binary consensus protocol, in which all non-faulty processes eventually make the same binary decision. RBC guarantees that non-faulty processes will never accept contradicting messages from any process and forces the faulty ones to output either nothing or the correct value. Agreement on a common subset (ACS) was introduced by Ben-Or [23] to serve as a consensus primitive for secure and efficient multi-party computation in an asynchronous system. Similar to [26], Ben-Or's ACS protocol is built with two primitives: RBC and binary asynchronous Byzantine agreement (ABA) which allows players to agree on the value of a single bit. HoneyBadgerBFT [116] is the first asynchronous BFT consensus protocol specifically designed for blockchains. It coordinates n players with different sets of transactions to agree on a common set of sorted transactions that will be included in a block. HoneyBadgerBFT's better performance comes from its optimized implementation of ACS using erasure codes [33] and the reduction from ACS to atomic reliable broadcast introduced by Ben-Or. BEAT [53] further optimizes HoneyBadgerBFT by using more efficient threshold encryption implementation and erasure-coding schemes. BEAT also incorporates a more efficient broadcast primitive—bandwidth-efficient asynchronous verifiable information dispersal (AVID-FP), which significantly improves its throughput in bandwidth scarce environment.

3.4 Benchmarking Consensus

Consensus protocols are typically benchmarked using three categories of workloads: null applications, key-value store (KV) workloads, and other application-specific workloads.

Null applications consist of generic commands with configurable request and response sizes, and optionally tunable contention levels. These workloads lack execution logic and isolate the coordination overhead of consensus mechanisms. They are employed in benchmarking tools such as BedRock [13] and serve as the default evaluation model for workload-oblivious protocols including MultiPaxos [152], Fast Paxos [99], NoPaxos [102], Nezha [71], Raft [124], HotStuff [161], PBFT [34], and Zyzzyva [94]. This setup enables controlled measurements of coordination costs independent of application semantics.

Key-value store workloads, typically composed of read and write operations, such as YCSB [42], are widely adopted in recent benchmarking frameworks like Paxi [8] and Bamboo [65]. KV workloads reflect practical use cases in coordination services such as etcd [44] and ZooKeeper [87], and support evaluations across varied dimensions, including read-write ratios and key popularity skew. Several protocols leverage these workloads to demonstrate performance benefits under specific access patterns. For example, EPaxos [119], EPaxosTOQ [151], and Atlas [58] exploit command commutativity to reduce coordination for non-conflicting operations. Adaptive protocols such as WPaxos [9] and Mencius [113] dynamically remap leaders to keys based on access frequency. Other protocols—including RMW-Paxos [143], Paxos-Quorum-Read [36], and Compartmentalized Paxos [156]— decouple read processing from consensus using leaderless quorum-based approaches, similar to shared register protoocls [108]. However, benchmarking with key-value workloads raises a fundamental concern: whether such workloads are suitable for evaluating consensus protocols. Alternative techniques, such as shared registers, may offer greater efficiency.

Finally, some protocols are evaluated using other application-specific workloads alongside null applications to capture realistic execution behavior. For instance, PBFT [34] and QU [3] have been tested using NFS file system workloads that emulate filesystem operations. In addition, HQ Replication [46] and QU [3] have been evaluated using atomic counter applications designed to stress state machine replication under frequent, increment-heavy updates.

3.5 Blockchain Interoperability

Interoperability [153] refers to the ability to execute operations across distinct systems or administrative domains, including software platforms, processes, or business entities. In the blockchain context, interoperability enables on-chain components to exchange data and trigger interactions with blockchain networks.

Public chain connectors focus on interoperability between cryptocurrency systems and encompass techniques such as sidechains, notaries, and hashed timelock contracts (HTLCs). In the sidechain model [22, 88, 67, 68], assets move between a main chain and a sidechain via *two-way peg* mechanisms [22]. Notary schemes use centralized or decentralized intermediaries to monitor multiple chains and trigger cross-chain transactions based on external events. Centralized notaries, such as traditional exchanges [40, 25], and decentralized ones, like automated market makers [158], help match counterparties. However, centralized exchanges expose users to risks, including key custody vulnerabilities and potential fraud.

The *blockchain of blockchains* (BoB) paradigm enables interoperability by offering reusable consensus, networking, storage, incentive, and smart contract layers. Notable BoB systems include Polkadot and Cosmos. Polkadot facilitates interoperability through the Cross-chain Message Passing Protocol (XCMP), whereas Cosmos employs the Inter-Blockchain Communication (IBC) protocol to coordinate block and transaction routing between chains.

3.6 Blockchain Oracles

Blockchain oracles deliver diverse data types, including asset prices, weather updates, event results, and more, enabling blockchains to execute sophisticated, data-driven logic while maintaining decentralization. To guarantee data integrity and authenticity, oracle-supplied data feeds are typically signed with the oracle's private key. Oracles are already integral to several blockchain applications, such as stablecoins [150, 109] and cross-chain asset swaps [29, 35]. Additionally, new use cases for oracle-provided data feeds continue to emerge across various domains [126]. GRuB [104] introduces adaptive replication for oracle data feeds, replicating data records on-chain selectively based on historical access patterns. However, this approach assumes infrequent updates from data feeds and may not suit DDTs that require fresh, low-latency data. In contrast, Dolphin adopts partial replication, replicating oracle data on-demand when DDTs are proposed. This design reduces data staleness and avoids wasting on-chain resources by limiting unnecessary replication of data feeds not immediately needed by executing DDTs.

Chapter 4

Comparison Between Consensus and Shared Register

In today's large-scale and geo-distributed web services, ensuring high availability has become essential. To achieve strong consistency despite failures, consensus protocols are commonly employed in web service infrastructures. These protocols offer the capability to implement arbitrary deterministic state machines through the abstraction of a shared command log. However, the impossibility result established by Fischer, Lynch, and Paterson [63] proves that no deterministic protocol can solve consensus in a purely asynchronous system if even a single process may crash. In contrast, shared register protocols [18] can implement linearizable shared registers, supporting basic read and write operations while guaranteeing termination in failure-prone asynchronous environments. The consensus hierarchy [84], which ranks the power of synchronization primitives based on how many processes they can solve consensus for, places shared registers as the weakest synchronization primitive, with a consensus number of one. Consequently, using an arbitrary number of shared registers to solve consensus for multiple processes is impossible. Interestingly, as discussed in Section 3.4, many works in consensus protocols [119, 134, 9, 58, 113, 156] evaluate their performance using key-value (KV) workloads consisting solely of read and write operations. This raises a fundamental question: Is consensus an overly strong primitive for achieving linearizability in the context of simple read and write operations? More specifically, can a weaker synchronization primitive outperform a stronger one when both are applied to the same workload in a message-passing environment? Additionally, are the commonly used key-value workloads well-suited for consensus protocols?

While consensus and shared register protocols have been extensively studied, their performance in Wide Area Network (WAN) environments is rarely compared directly. Previous

research [30] indicates that shared register protocols can outperform consensus protocols in terms of tail latency, primarily due to the absence of the *stable ordering constraint*. This constraint, fundamental to consensus, requires that each operation be executed only after all preceding operations in the command log have been committed. However, such evaluations have typically focused on low-throughput settings and overlook the impact of command batching—a widely adopted technique to enhance throughput and amortize communication overhead.

To address this gap, this work presents a performance comparison between two representative protocols: ABD [108], a foundational algorithm for implementing linearizable shared registers, and EPaxos [119], a leaderless consensus protocol optimized for WAN latency. The evaluation considers both protocols with command batching enabled. The results expose two often-overlooked limitations of consensus protocols when applied to key-value workloads: coarse-grained processing and the immediate sub-effects of write operations. These factors significantly increase the coordination overhead of consensus protocols as client load scales.

4.1 Background

4.1.1 Consensus Protocols

Consensus protocols can implement a replicated state machine (RSM), enabling fault-tolerant emulation of arbitrary data types by agreeing on a sequence of operations over shared state variables. This sequence is typically represented as a command log, with each entry containing one or more operations. Due to the deterministic nature of state machines, each replica produces identical state transitions and output sequences. Thus, an operation is executed only after all prior log entries have completed. This requirement is referred to as *log-dependent* execution. Some previous work [30] also describes it as *stable ordering*.

Paxos [98] is widely regarded as one of the most influential consensus protocols for achieving agreement in distributed systems. Paxos typically requires two round-trips of communication to reach agreement among replicas. In the first round-trip, a proposer learns about any concurrent proposals that may exist. In the second round-trip, the proposer replicates either its own value or a value learned from the first round to the other replicas. Paxos inspires a series of subsequent work and many variants have been developed, including Multi-Paxos [152], Fast Paxos [99], and EPaxos [119]. Multi-Paxos introduces

a stable leader, allowing the protocol to bypass the initial prepare phase in subsequent consensus rounds, and optimizes throughput by reusing the same leader across multiple rounds, reducing the overhead of repeated prepare phases. Fast Paxos reduces latency by allowing clients to communicate directly with any replicas, bypassing the leader when possible, though it requires larger quorum sizes. This approach is also beneficial in WAN environments, as it enables clients to interact with the nearest replicas. EPaxos adopts a similar leaderless design, allowing clients to send requests to any replica. Compared to Fast Paxos, EPaxos further improves performance by exploiting the commutativity of proposed commands and coordinating with a smaller quorum size. However, it introduces a more complex execution mechanism based on topological sorting of command dependencies, which may become a performance bottleneck under high contention.

4.1.2 Shared Register Protocols

The shared register is a fundamental shared object, offering two basic interfaces: **Read()** and **Write(v)**. It can be emulated in asynchronous message-passing systems that tolerate crashes of a minority of processes, using protocols such as ABD [18]. Despite their simplicity, shared registers have limited synchronization power, as formalized by the concept of *consensus numbers* introduced by Herlihy [84]. Shared registers have a consensus number of 1, the weakest level in Herlihy's hierarchy, meaning they cannot be used to implement consensus among two or more processes. Herlihy further proves that no collection of objects with consensus numbers less than n can be used to implement an object with consensus number n. Nevertheless, shared registers offer more efficient coordination due to their simple interface semantics. In particular, they do not require *log-dependent* execution; for example, a **Write(v)** operation can commit without observing the current state of the register. This enables *logless* execution, where each operation completes immediately after its coordination phase, without waiting for unrelated operations to finish. In contrast, consensus requires *log-dependent* execution to achieve stronger symmetry-breaking power, making it theoretically more difficult to implement than the coordination mechanisms used in shared registers.

4.2 The Tale of Two Protocols

This section compares two candidate protocols: ABD, a classic shared-register protocol, and EPaxos, a leaderless consensus protocol. ABD is selected as the representative shared-register protocol due to its foundational role in distributed computing literature and its

well-established correctness under asynchronous, crash-prone environments. EPaxos is selected as the consensus protocol candidate because its coordination mechanism resembles that of ABD, while offering stronger synchronization guarantees for arbitrary commands beyond simple reads and writes. The comparison reveals trade-offs involved in achieving stronger synchronization within distributed environments. The analysis uncovers two previously overlooked limitations of consensus protocols in the context of processing key-value workloads: coarse *processing granularity* and immediate *sub-effects of writes*.

4.2.1 ABD Algorithm

ABD [108] emulates a multi-writer multi-reader (MWMR) shared register in a message-passing environment. Algorithm 1 presents the procedures for read and write operations, while Algorithm 2 outlines the message-handling logic executed by each replica process. The message types used in the protocol are illustrated in Figure 4.1.

Message Type	**Description**
Read1	Initial read request
Read1Resp(ts, v)	Response to Read1 containing timestamp ts and value v
Read2(ts, v)	Write-back request carrying timestamp ts and value v
Read2Resp	Acknowledgment in response to Read2
Write1	Initial write request
Write1Resp(ts)	Response to Write1 with the current timestamp ts
Write2(ts, v)	Second-round write request with timestamp ts and value v
Write2Resp	Acknowledgment in response to Write2

Figure 4.1: Message Types and Parameters in the ABD Algorithm

Unlike the single-writer-multi-reader (SWMR) shared register, the writer does not know the latest version of the register. Hence, both the read and write procedures require one round-trip to retrieve the value associated with the latest version, represented as a tuple containing a monotonically increasing sequence number and a process identifier. In the second round-trip, the writer process creates a new version by incrementing the largest sequence number retrieved from the first round of responses. It then propagates the new value associated with the updated version to a majority of servers. If other concurrent writers exist, multiple writers can propagate versions with the same sequence number. In such a scenario, the id field will determine the rank of those versions. The reader process also performs a second round-trip, known as the write-back phase, because the latest value

obtained in the first round-trip may be stored by only a minority of processes. This second round-trip ensures that subsequent read operations can return the same value, provided no concurrent write is present. As a result, ABD guarantees the linearizability of concurrent operations on the register. A common optimization for the reader is to execute a fast-path read when the first round request receives replies from a *consistent quorum* [72]. A consistent quorum is a quorum of replicas that yield the same response. Since the purpose of the second round-trip is to prevent old-new inversion through writing the read value to a quorum of processes, given the presence of a consistent quorum, the reader process can bypass the second round-trip and return immediately.

Algorithm 1: ABD Algorithm

1 **Procedure** `Read()`**:**
2 **broadcast** Read1
3 **wait until** Read1Resp(ts_q, v_q) received from each process q in a quorum Q
4 $ts \leftarrow \max_{q \in Q} ts_q$
5 $v \leftarrow v_q$ **with** $ts_q = ts$
6 **if** $\exists q \in Q, ts_q \neq ts$ **then**
7 **broadcast** Read2(ts, v)
8 **wait until** Read2Resp received from some quorum Q'
9 **return** v
10 **Procedure** `Write`(v)**:**
11 **broadcast** Write1
12 **wait until** Write1Resp(ts_q) received from each process q in a quorum Q
13 $ts \leftarrow 1 + \max_{q \in Q} ts_q$
14 **broadcast** Write2(ts, v)
15 **wait until** Write2Resp received from some quorum Q'
16 **return**

4.2.2 EPaxos Algorithm

EPaxos is a leaderless consensus protocol that allows clients to issue commands to the nearest replica, making it particularly well-suited for WAN environments. This feature distinguishes it from leader-based protocols [152, 124], which require clients to interact with a designated leader, often incurring higher latency in geographically distributed systems. EPaxos guarantees liveness only under partially synchronous conditions, as dictated by the FLP result [63].

Algorithm 2: Message Handling

```
   ▷ State variables on a process p
1  Var ts_p: register version
2  Var v_p: register value
   ▷ Upon receiving a message m from a process q
3  Procedure HandleMessage(m):
4      switch m do
5          case m is Read1 do
6              send Read1Resp(ts_p, v_p) to q
7          case m is Read2(ts_q, v_q) do
8              Update(ts_q, v_q)
9              send Read2Resp to q
10         case m is Write1 do
11             send Write1Resp(ts_p) to q
12         case m is Write2(ts_q, v_q) do
13             Update(ts_q, v_q)
14             send Write2Resp to q
15 Procedure Update(ts', v'):
16     if ts_p < ts' then
17         ts_p ← ts'
18         v_p ← v'
```

The protocol has three distinct phases, namely, **PreAccept**, **Accept**, and **Commit**, in failure-free executions. When a command is committed on the fast path, the coordinator returns to the client immediately after the PreAccept phase and skips the Accept phase. However, if the command is committed on the slow path, it proceeds to the Accept phase before entering the Commit phase.

The PreAccept and Accept phases in EPaxos are designed to establish the dependencies of a command c, which represent the set of commands that must execute prior to c. The purpose of the Commit phase is for the coordinator to notify other replicas about the finalized dependencies. During the PreAccept phase, the coordinator of a command constructs an initial dependency set, consisting of all other commands that conflict with the given command. The coordinator then sends the command along with this preliminary set of dependencies to a fast quorum of replicas. The replicas update the received dependency set with any conflicting commands they are aware of that are not already included, and then respond to the coordinator with the modified dependencies. If the coordinator receives

responses from a fast quorum where the dependencies are consistent across all replies, it proceeds directly to the Commit phase. Otherwise, it transitions to the Accept phase to resolve the discrepancies, which involves an additional round-trip of communication to establish a consensus on the updated dependencies.

4.2.3 A First Glance

Table 4.1 provides a summary of the essential features of the two protocols, ABD and EPaxos. Although they are distinct types of protocols, they exhibit several shared characteristics.

	ABD	**EPaxos**
Quorum Sizes	$f + 1$	**Slow-path:** $f + 1$ **Fast-path:** $f + \lfloor \frac{f+1}{2} \rfloor$
Fast Read	✓	✓
Fast Write	✗	✓
Execution	Logless	Log-dependent
Processing Granularity	Operation	Log Entry
Sub-effects of Writes	Delayed	Immediate

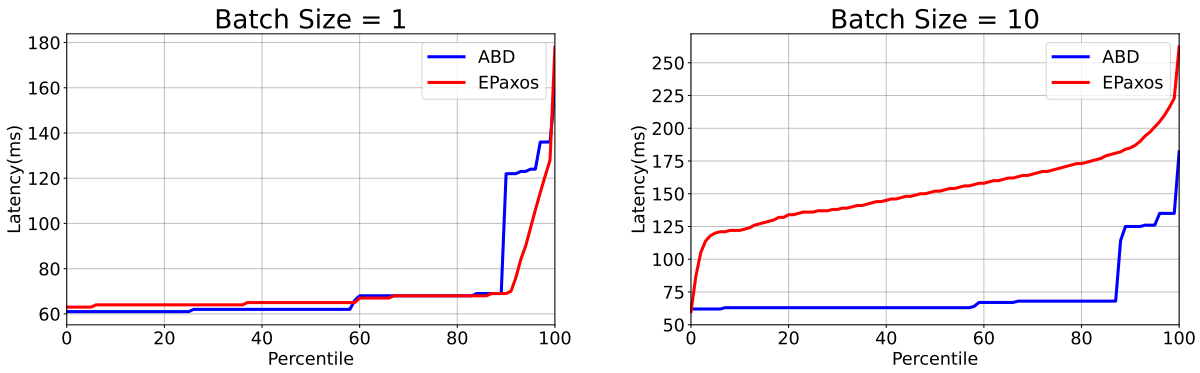
Table 4.1: Comparing ABD with EPaxos

First, both ABD and EPaxos are leaderless, enabling clients to send requests to the closest replica, which is favourable for WAN environments.[1] Second, both protocols require at most two communication round-trips and implement a fast-path commit mechanism that requires only a single communication round-trip within a quorum of replicas, thereby reducing commit latency. It is important to note, however, that EPaxos facilitates fast-path commit for both read and write operations, while ABD only supports it for read operations. Lastly, ABD and EPaxos exhibit similar quorum sizes. Despite EPaxos having different quorum sizes for fast-path commits, in the most frequently encountered deployment scenarios where the number of replicas is either 3 or 5, the fast-path quorum of EPaxos consists of a simple majority of replicas.

[1] In conventional settings, client processes invoke read and write procedures directly. However, in a proxy server configuration, clients can send requests to a replica that then executes the read/write operations on their behalf.

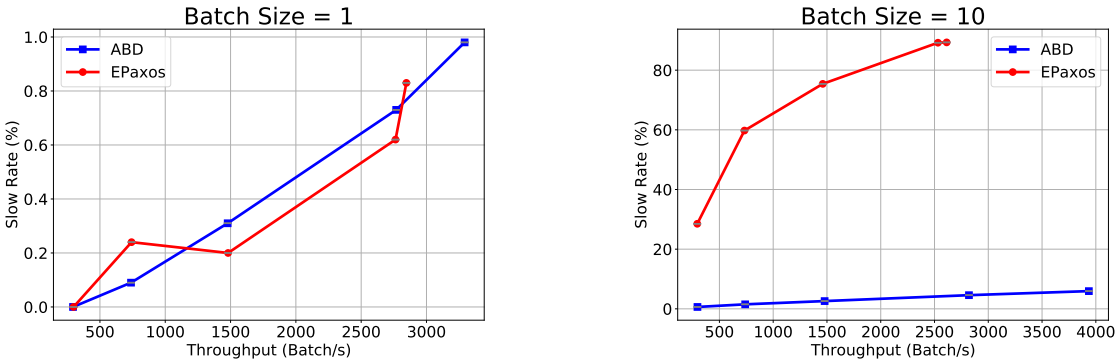
A superficial comparison of quorum sizes and communication round-trips may suggest that EPaxos outperforms ABD when processing key-value workloads in wide-area network (WAN) environments, where communication costs are high. However, prior work such as Gryff [30] shows that shared register protocols achieve better tail latency, primarily because they avoid log-dependent execution. This observation leads to a key question regarding the comparative capabilities of shared register and consensus protocols for processing key-value workloads in WAN settings: *Is the absence of log-dependent execution the only advantage shared register protocols possess?* To examine this question, EPaxos and ABD are evaluated using three geo-distributed replicas located in Virginia, Oregon, and Ireland. A client is deployed in each region, issuing requests to its co-located replica server at varying request rates.

In both protocols, the likelihood of fast-path commit increases when conflicting operations are infrequent. Two operations are considered *conflicting* if they access the same data item (e.g., the same key in a key-value store) and at least one operation updates the data item. As a result, the fast-path commit achieves superior performance in read-heavy workloads, where most operations are non-conflicting. Based on this observation, a read-heavy workload is used, comprising 10% write and 90% read operations. The key access pattern follows a Zipfian distribution with a theta parameter of 0.6, indicating moderate skew. The key space consists of 1,000 unique keys. End-to-end operation latency is measured to identify two previously overlooked weaknesses in consensus protocols when handling key-value workloads, compared to shared-register protocols such as ABD.



(a) No Batching (b) With Batching

Figure 4.2: Percentile Latency



(a) No Batching (b) With Batching

Figure 4.3: Rate of Slow Read vs Throughput

4.2.4 Diminishing Fast-Path Commit

Figure 4.2 presents the percentile latency of the two protocols under configurations with and without batching. Batching refers to the technique of aggregating multiple client requests into a single protocol message or processing round, thereby reducing coordination overhead and improving throughput. When batching is disabled, Figure 4.2a shows that EPaxos and ABD exhibit similar latency performance. EPaxos even achieves better P95 latency than ABD, due to its ability to commit both read and write operations on the fast path. Figure 4.3 demonstrates how the slow-path commit rate for read operations changes with respect to throughput with/without batching. Figure 4.3a demonstrates that there are no fundamental differences between EPaxos and ABD when batching is disabled: 99% of read operations are committed on fast-path for both protocols.

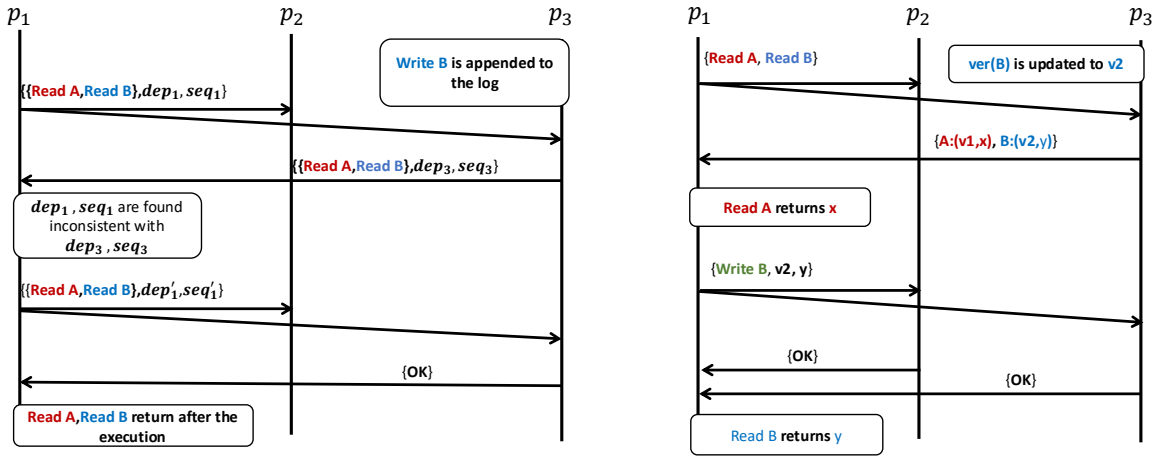
Further analysis reveals that the fast-path commit rate of EPaxos drops substantially under batching, which contributes to its higher latency. When batching is enabled, client requests are grouped into batches, each containing 10 independent operations. Under this configuration, ABD significantly outperforms EPaxos, which suffers from increased latency. As shown in Figure 4.2b, ABD maintains a high fast-path commit rate for reads, even under high throughput (Figure 4.3b). At 40K operations per second, over 85% of reads complete with just one round-trip of communication. In contrast, EPaxos shows a sharp drop in fast-path commit rate, with over 90% of reads requiring two round-trips of communication. A similar pattern is also observed when EPaxos's execution phase is disabled, removing the effects of log-dependent execution. Why does batching become a game changer? The key lies in the subtle differences in how the two protocols detect and handle conflicts arising

from concurrent operations.

4.2.5 Processing Granularity

Shared register protocols operate at the per-operation level. According to Algorithm 1, a read operation can bypass the second round of coordination when it receives a consistent quorum of responses. This rule also applies when coordinating a batch of read operations accessing different keys. In contrast, consensus protocols typically maintain a consistently replicated command log and operate at the granularity of a log entry. On a given replica, an entire batch of operations associated with a log entry is committed and executed together.

In EPaxos, when a replica p receives a batch of client commands, it constructs a log entry and coordinates with other replicas to determine a sequence number ***seq*** and a dependency list ***dep*** containing all other log entries that the current entry depends on. A log entry can be committed via the fast path if all conflicting entries on a fast-path quorum of replicas are already known to p. Two log entries are considered conflicting if their associated command batches contain conflicting commands.



(a) EPaxos (b) ABD

Figure 4.4: Different processing granularity when handling a batch of two read operations. In ABD, `Read A` returns after one round-trip of communication while both `Read A` and `Read B` return after two round-trips of communication in EPaxos.

Figure 4.4 demonstrates a scenario where two read operations `Read A` and `Read B` are received and handled differently in EPaxos and ABD. In EPaxos, p_1 broadcasts a

PreAccept request including the log entry $\boldsymbol{inst_1}$ = {`Read A, Read B`}, its corresponding sequence number $\boldsymbol{seq_1}$ and its dependency list $\boldsymbol{dep_1}$ computed based on the local view of the command log on p_1. Upon receiving the **PreAccept** request, p_3 replies with $\boldsymbol{seq_3}$ and $\boldsymbol{dep_3}$, which are respectively the sequence number and dependency list of $\boldsymbol{inst_1}$ computed based on p_3's view of the command log. Since p_3 has an instance in its command log that contains a write operation on key `B`, which is not known to p_1, $\boldsymbol{seq_3}$ and $\boldsymbol{dep_3}$ are inconsistent with $\boldsymbol{seq_1}$ and $\boldsymbol{dep_1}$. As a result, all operations associated with $\boldsymbol{inst_1}$ require a second round of coordination. When a similar situation happens in ABD, where p_3 has the latest version v_2 for key `B`. Due to a finer processing granularity, p_1 will receive responses from a consistent quorum (including p_1 itself) for `Read A` and returns the value to clients without going through the second round-trip for coordination.

4.2.6 The Sub-effect of Writes

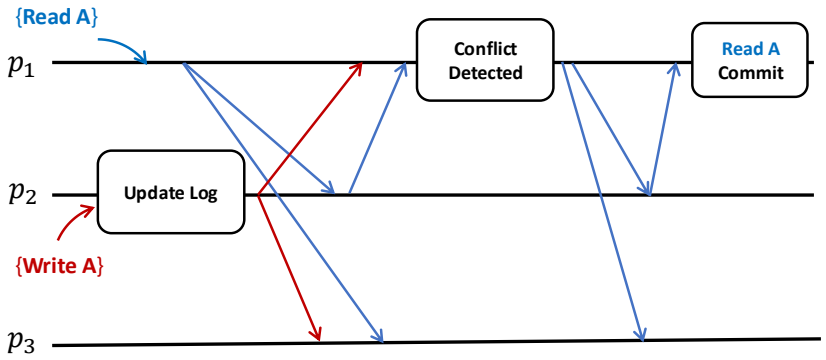
Another subtle difference between shared register protocol and consensus protocol lies in the timing of the sub-effects of write operations on each replica process. The following definitions are introduced first:

Definition 10 (Core Variables)**.** Core variables are local variables maintained by each process whose value determines the communication round-trips for coordinating operations.

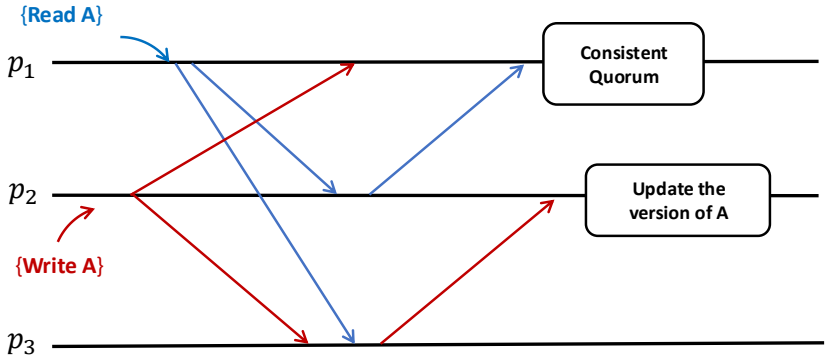
For ABD, the version of the register is the core variable. For EPaxos, the local view of the command log is the core variable.

Definition 11 (Sub-effects)**.** The updates to core variables of process p that result from coordinating an operation o, where p serves as the coordinator of o.

In the context of key-value workloads, the sub-effect of a write operation $w = \texttt{put}(k, v)$ refers to any update made by w to the core internal variables of the coordinator replica (i.e., the replica that receives w from a client), excluding the value of key k. In ABD, the sub-effect of w simply updates the version associated with key k, and this update does not take effect until w completes a full round-trip of communication with a quorum of replicas. In contrast, in EPaxos, the sub-effect of w updates the coordinator replica's local view of the command log—including incrementing the maximum log entry identifier and appending a new log entry that contains w. These sub-effects occur immediately upon receipt of the write, prior to any coordination with other replicas. Similar behavior is observed in other leaderless consensus protocols [58, 99].



(a) EPaxos



(b) ABD

Figure 4.5: (a) In EPaxos, a write operation on object A processed by p_2 immediately updates the command log before any other replica is aware of it. As a result, a subsequent read commits on the slow path due to a conflict with the write processed by p_2; (b) In ABD, the same write operation processed by p_2 updates object A only after completing one round-trip of communication with other replicas. Consequently, a read operation coordinated by p_1 can commit on the fast path, as the write operation has not yet taken effect on p_2 when p_2 replies to p_1.

As illustrated in Figure 4.5a, EPaxos permits replica p_2 to update its local log prior to coordinating with other replicas. This behavior may cause a concurrent read from another replica to observe an inconsistent log state, resulting in fallback to the slow path. In contrast, ABD updates the version of an object A at p_2 only after completing a communication round with a quorum, enabling reads on A to complete in a single round-trip—a key performance advantage in WAN environments.

4.3 Chapter Conclusion

This study shows that log-dependent execution is not the sole performance-limiting factor of consensus protocols in the context of key-value workloads. Coarse processing granularity and the immediate sub-effects of write operations also contribute to a higher incidence of slow-path commits, which significantly amplifies coordination overhead. These inefficiencies stem from the replicated command log abstraction, which, although designed for general-purpose state machine replication, is poorly suited to the fine-grained, high-frequency access patterns typical of key-value workloads.

However, these challenges are not necessarily unavoidable in practice. For example, some system designs [107, 90] propose assigning a separate replicated state machine (RSM) to each key or partition, which reduces contention and minimizes coordination granularity. Additionally, some approaches [36, 30] mitigate the impact of log-dependent execution by decoupling the processing of reads and writes, employing quorum-based read or shared register protocols while reserving consensus for more complex operations such as read-modify-write. These workload-aware strategies show that consensus can be effectively combined with weaker primitives to better match the access patterns and performance requirements of key-value workloads.

Chapter 5

Enhancing Shared Registers for WAN Environments

Many distributed systems rely on a shared read/write abstraction to coordinate concurrent access to replicated state. This abstraction is particularly relevant in replicated storage systems, metadata services, and distributed coordination frameworks [44, 87], where linearizable read and write operations form the basis of consistency guarantees. In emerging blockchain-based infrastructures, the read/write interface also aligns closely with the requirements of decentralized oracle networks [104]. In such systems, off-chain data must be written to the chain through reliable, time-sensitive updates, while smart contracts issue reads to access the most recent values. Supporting efficient and consistent read/write access is therefore critical for both traditional storage applications and modern oracle-driven use cases, where performance and availability directly impact data freshness and transaction completion.

As discussed in the previous chapter, when client load is low, the primary performance bottleneck for coordination in a WAN environment is the latency associated with communication round-trips. In such settings, ABD performs suboptimally under write-heavy workloads due to its inherent requirement of two communication round-trips for write operations. To mitigate this limitation, we explore optimizations for ABD and introduce WABD (Watermark-integrated ABD) as a solution. WABD leverages clock-based techniques to reduce ABD's communication overhead by enabling single-round-trip writes. Each replica process is assumed to have access to a local clock synchronized via hardware or software methods, such as the NTP algorithm [117]. By imposing a waiting period based on the maximum clock offset, WABD guarantees linearizable single-round-trip write operations. WABD also supports linearizable local reads through a watermark gossiping

mechanism [61], allowing reads to complete locally in failure-free cases, albeit with added latency from watermark advancement. To further optimize performance, WABD adaptively switches between quorum and local reads based on client load. In low-load scenarios, it adopts CABD's quorum read strategy for optimal latency, while under high load, it prioritizes coordination-free local reads, reverting to quorum reads when necessary. We implemented WABD and evaluated its performance against EPaxos and EPaxosTOQ [151] on Amazon EC2. Our results show that WABD outperforms both EPaxos variants in throughput and latency, particularly under high-throughput workloads.

5.1 Model

The system comprises a set of n processes, where each pair is connected via bidirectional point-to-point channels. We assume a crash failure model, allowing up to f failures, with $n = 2f+1$. The replicated store is modeled as a collection of shared registers, each uniquely identified by a key from a predefined key domain K, and storing values drawn from a value domain V. Each process p functions as a replica of a key-value store and is uniquely identified within the system. Client processes operate externally to the system, interacting with it via $\mathbf{get}(k)$ and $\mathbf{put}(k, v)$ operations, where $k \in K$ and $v \in V$. Clients issue requests to a designated process p, which coordinates with other processes as necessary and responds to the client upon operation completion. If the designated process p fails during an operation, the client is responsible for retrying the request with an alternative replica. A $\mathbf{get}(k)$ operation, when issued to process p, triggers a **read** operation on the shared register associated with key k. Similarly, a $\mathbf{put}(k, v)$ operation initiates a **write** operation of value v to the same register. Each write operation w is assigned a timestamp by its coordinator process p at the start of coordination. This timestamp is defined as $ts(w) = \langle lt_p, p\rangle$, where lt_p denotes the value of process p's local clock at the time $ts(w)$ is issued. To formalize this construction, let $c(t, p)$ denote the value of process p's local clock at real time t; The notation $clock(\langle lt_p, p\rangle)$ is used to denote lt_p, i.e., the clock component of the timestamp. It is assumed that process p issues at most one operation within the interval $[lt_p,\ lt_p + 1)$.

All processes are equipped with local clocks synchronized via hardware (e.g., atomic clocks) or software-based time protocols such as NTP [117]. Clocks are assumed to have bounded drift, to advance monotonically, and never to move backward. Synchronization occurs over a separate synchronous channel or via trusted hardware, while inter-replica communication and processing are otherwise asynchronous. The maximum deviation between any local clock and a reference time source is bounded by a known constant $o_{\max}$,

formalized by the following assumption:

Definition 12 (Bounded Clock Offset Assumption)**.** There exists a constant $o_{\max} > 0$ such that, for all processes $p \in R$ and all real times $t \in \mathbb{R}^+$, the clock offset is bounded as follows:

$$\left|c(t,p) - t\right| < o_{\max}.$$

5.2 Consistency Properties

Linearizability [86], also referred to as atomicity in the context of read/write registers, is a widely accepted correctness condition for the interaction between a set of processes and shared registers through the invocation of their exposed interfaces. Section 2.5.1 provides a general definition for a collection of arbitrary shared data types. This chapter focuses on linearizability for a single shared register object, motivated by the fact that linearizability is compositional: a collection of individually linearizable objects is also linearizable as a whole [86].

The definition of linearizability for a single shared register object is specified as follows:

Definition 13 (Register Linearizability)**.** A history H of a shared register is said to be *linearizable* if it can be extended to a history H' (by appending zero or more matching response events or removing incomplete operations) such that there exists a sequential history S that preserves the order of events in H' and satisfies the following two conditions:

1. **Real-time constraint**: If $o_1 \prec_H o_2$, then $o_1 \prec_S o_2$.
2. **Read constraint**: Each read operation returns either the value written by the most recent preceding write operation in S, or an initial value if no such write exists.

5.2.1 Monotonicity Properties

Our analysis of consistency will focus on four monotonicity properties defined by [125] which are suffcient conditions for linearizability. We define the notation $ts(o)$ as follows:

1. If o is a write operation that writes a value v, then $ts(o)$ is the unique timestamp assigned to v by the process that issued o.

2. If o is a read operation that returns a value v, then $ts(o)$ is the timestamp associated with v. In other words, $ts(r) = ts(w)$, where w is the write operation that wrote the value v returned by the read operation r.

For each history H, and any two operations, the following four properties that must be guaranteed by a linearizable shared register:

Property 5.2.1. ***Write-write Monotonicity***: $w_1 \prec_H w_2 \implies ts(w_1) < ts(w_2)$.

Property 5.2.2. ***Write-read Monotonicity***: $w \prec_H r \implies ts(w) \leq ts(r)$.

Property 5.2.3. ***Read-write Monotonicity***: $r \prec_H w \implies ts(r) < ts(w)$.

Property 5.2.4. ***Read-read Monotonicity***: $r_1 \prec_H r_2 \implies ts(r_1) \leq ts(r_2)$.

Lemma 1. *For a shared register object, if a history H satisfies all four monotonicity properties, then H is linearizable.*

Proof. Let H be a history that satisfies all required monotonicity properties. To construct an extended history H', append the response event $resp(w)$ to H for each pending write operation $w \in H$, and remove any pending read operation r from H.

We then construct a sequential history S by sorting operations o in increasing order of $ts(o)$, with each invocation immediately followed by its corresponding response. Ties are resolved as follows: 1. Write operations precede read operations. 2. Otherwise, ties are broken based on the order of invocation in H. To satisfy the real-time constraint, if $o_1 \prec_H o_2$, then $o_1 \prec_S o_2$ must hold. By the monotonicity property, $o_1 \prec_H o_2$ implies $ts(o_1) \leq ts(o_2)$. If $ts(o_1) < ts(o_2)$, then $o_1 \prec_S o_2$ holds. If $ts(o_1) = ts(o_2)$, the tie-breaking rule ensures $o_1 \prec_S o_2$. In S, any read operation r and its closest preceding write operations w have the same timestamp $ts(w)$. As a result, read constraint also holds. □

5.3 Fast Write via Loosely Synchronized Clocks

The original ABD protocol uses monotonically increasing sequence numbers in timestamps to label register versions. Computing the timestamp for a write operation requires an extra round to retrieve the latest version from a majority quorum of replicas. If processes had access to perfectly synchronized clocks, this initial round-trip could be replaced by reading the local clock, allowing write operations to complete with a single round-trip without compromising consistency guarantees—an advantage in WAN settings. However,

perfectly synchronized clocks are unattainable in practice, and prior work [54] has shown that fast (i.e., single-round) algorithms are impossible without additional assumptions about timing. Some work [125] proves that single-round-trip algorithms can guarantee no more than write-read monotonicity.

To address this, the present work investigates single-round-trip read and write algorithms that ensure strong consistency under the bounded clock offset assumption formalized in Definition 12. Under this model, clocks are only loosely synchronized, but the bounded skew assumption enables precise reasoning about the timing of operations. The remainder of this section investigates the following question: Can fast algorithms be achieved under the assumption of access to such clocks?

Algorithm 3: Clock-Optimized ABD (CABD)

▷ State variables on process p
Var ts_p: register version
Var v_p: register value
Procedure `DeltaRead`(δ)**:**
  broadcast Read1
  wait until Read1Resp(ts_q, v_q) received from all q in some quorum Q
  $ts \leftarrow \max_{q \in Q} ts_q$
  $v \leftarrow v_q$ **with** $ts_q = ts$
  `WriteBack`(Q, ts, v)
  wait until `GetTime`$() - clock(ts) > \delta$
  return v
Procedure `WriteBack`$(Q,\ ts,\ v)$**:**
  if $\exists q \in Q, ts_q \neq ts$ **then**
    broadcast Read2(ts, v)
    wait until Read2Resp(ts) received from all q in some quorum Q'
Procedure `DeltaWrite`(v, δ)**:**
  $ts \leftarrow \langle$`GetTime`$(), p\rangle$
  broadcast Write(ts, v)
  wait until `GetTime`$() - clock(ts) > \delta \wedge$ WriteResp(ts) received from some quorum Q
Procedure `Update`(ts', v')**:**
  if $ts_p < ts'$ **then**
    $ts_p \leftarrow ts'$
    $v_p \leftarrow v'$

```
   ▷ Upon receiving a message m from process q
23 Procedure HandleMessageCABD(m):
24     switch m do
25         case m is Read1 do
26             send Read1Resp(ts_p, v_p) to q
27         case m is Read2(ts_q, v_q) do
28             Update(ts_q, v_q)
29             send Read2Resp(ts_q) to q
30         case m is Write(ts_q, v_q) do
31             Update(ts_q, v_q)
32             send WriteResp(ts_q) to q
```

5.3.1 Clock Optimized ABD

Algorithm 3 presents the resulting algorithm, referred to as Clock-Optimized ABD (CABD), with its message-handling logic detailed in line 23. The most prominent difference compared to ABD is that CABD achieves single-round-trip write coordination. However, in addition to message exchanges for quorum-based coordination, both read and write operations enforce timing constraints to ensure linearizability.

Let δ denote the delay parameter that determines when coordination may terminate at a process, ensuring consistent operation ordering in the presence of bounded clock skew, as defined by the assumption in Definition 12. A read operation terminates only after the local clock time exceeds the maximum timestamp received from a quorum by at least δ, as enforced on line 9; a write operation completes only after the local clock time exceeds its locally measured invocation time by δ units, as shown on line 18. As a result, the latency of coordination is influenced not only by the number of communication round-trips but also by the value of δ. The algorithm achieves improved coordination efficiency when δ is smaller than the duration of one communication round-trip.

5.3.2 Analysis for CABD

This section analyzes the choice of δ required to ensure linearizability in Algorithm 3.

Lemma 2. *For $p_1, p_2 \in R$, if $c(t_1, p_1) - c(t_2, p_2) > \delta$, then it follows that $t_1 - t_2 > \delta - 2o_{max}$.*

Proof. According to Definition 12, we have $t_1 > c(t_1, p_1) - o_{max}$ and $t_2 < c(t_2, p_2) + o_{max}$. We then have:

$$t_1 - t_2 > c(t_1, p_1) - o_{max} - (c(t_2, p_2) + o_{max}) \tag{5.1}$$
$$= (c(t_1, p_1) - c(t_2, p_2)) - 2o_{max} \tag{5.2}$$
$$= \delta - 2o_{max} \tag{5.3}$$

□

Lemma 3. *Let o_1 and o_2 denote the local clock offsets of processes p_1 and p_2, respectively. The difference between these offsets satisfies:*

$$|o_2 - o_1| < 2o_{\max}.$$

Proof. Under the bounded clock offset condition specified in Definition 12, the local clock at any process $p \in R$ satisfies:

$$\left|c(t, p) - t\right| < o_{\max}$$

for all real times t. This implies the local clock offset $o_p = c(t, p) - t$ is bounded such that:

$$-o_{\max} < o_p < o_{\max}.$$

Applying this to processes p_1 and p_2, it holds that:

$$-o_{\max} < o_1 < o_{\max}, \quad -o_{\max} < o_2 < o_{\max}.$$

Therefore, the difference $o_2 - o_1$ is bounded as follows:

$$o_2 - o_1 < o_{\max} - (-o_{\max}) = 2o_{\max},$$
$$o_2 - o_1 > -o_{\max} - o_{\max} = -2o_{\max}.$$

Combining both bounds:

$$-2o_{\max} < o_2 - o_1 < 2o_{\max},$$

which implies:

$$|o_2 - o_1| < 2o_{\max}.$$

□

Lemma 4. *For real times t_1, t_2, t_3 and processes $p_1, p_2, p_3 \in R$, if $c(t_2, p_2) - c(t_1, p_1) > 4o_{\max}$ and $t_2 < t_3$, then $c(t_3, p_3) > c(t_1, p_1)$.*

Proof. By Lemma 2, it follows that:

$$t_2 - t_1 > 4o_{\max} - 2o_{\max} = 2o_{\max}.$$

Therefore:

$$t_2 > t_1 + 2o_{\max}.$$

Since $t_3 > t_2$, one obtains:

$$t_3 > t_2 > t_1 + 2o_{\max}.$$

Let $t_3 = t_1 + 2o_{\max} + \epsilon$ for some $\epsilon > 0$. Let o_1 and o_3 denote the clock offsets of p_1 and p_3, respectively. Then:

$$\begin{aligned} c(t_3, p_3) - c(t_1, p_1) &= (t_1 + 2o_{\max} + \epsilon + o_3) - (t_1 + o_1) \\ &= 2o_{\max} + \epsilon + (o_3 - o_1). \end{aligned}$$

By Lemma 3, $o_3 - o_1 > -2o_{\max}$, so:

$$c(t_3, p_3) - c(t_1, p_1) > \epsilon > 0.$$

Hence:

$$c(t_3, p_3) > c(t_1, p_1).$$

□

Theorem 1. *Algorithm 3 guarantees Property 5.2.1, i.e.,* $w_1 \prec_H w_2 \implies ts(w_1) < ts(w_2)$, *if and only if* $\delta \geq 4o_{\max}$.

Proof. To prove that Algorithm 3 guarantees Property 5.2.1, i.e., $w_1 \prec_H w_2 \Rightarrow ts(w_1) < ts(w_2)$, if and only if $\delta \geq 4o_{\max}$, both directions are established below.

Let p_1 and p_2 denote the coordinators of w_1 and w_2, respectively. Let t_1^{inv} and t_1^{resp} denote the real times at which w_1 is invoked and completed, respectively. Let t_2^{inv} denote the real time at which w_2 is invoked.

($\Leftarrow$) Assume $\delta \geq 4o_{\max}$. Since the algorithm enforces a local wait of at least δ at p_1 (line 9), it follows that:

$$c(t_1^{resp}, p_1) - c(t_1^{inv}, p_1) > \delta \geq 4o_{\max}.$$

Given $w_1 \prec_H w_2$, the invocation of w_2 at real time t_2^{inv} occurs after the response of w_1, i.e., $t_1^{resp} < t_2^{inv}$. By Lemma 4, it follows that:

$$c(t_2^{inv}, p_2) > c(t_1^{inv}, p_1).$$

Given that $clock(ts(w_2)) = c(t_2^{inv}, p_2)$ and $clock(ts(w_1)) = c(t_1^{inv}, p_1)$, it follows that:

$$clock(ts(w_2)) > clock(ts(w_1)).$$

Thus, $ts(w_1) < ts(w_2)$.

($\Rightarrow$) Suppose $\delta < 4o_{\max}$, and let $\delta = 4o_{\max} - \epsilon_1$ for some $\epsilon_1 > 0$. The proof constructs a counterexample to Property 5.2.1, in which $w_1 \prec_H w_2$, but $ts(w_1) > ts(w_2)$ holds.

Let t_1^{inv} and t_1^{resp} denote the real times at which w_1 is invoked and completed, respectively. Since the algorithm enforces a local wait of δ, we have:

$$c(t_1^{resp}, p_1) - c(t_1^{inv}, p_1) > \delta.$$

By Lemma 2, this implies:

$$t_1^{resp} - t_1^{inv} > \delta - 2o_{\max}.$$

Assume w_2 is invoked at time t_2^{inv}, such that:

$$t_2^{inv} - t_1^{inv} = \delta - 2o_{\max} + \epsilon_2,$$

for some $\epsilon_2 > 0$. Substituting $\delta = 4o_{\max} - \epsilon_1$, we obtain:

$$t_2^{inv} = t_1^{inv} + 2o_{\max} + \epsilon_2 - \epsilon_1.$$

Let o_1 and o_2 denote the clock offsets of p_1 and p_2. Then:

$$\begin{aligned} clock(ts(w_2)) - clock(ts(w_1)) &= c(t_2^{inv}, p_2) - c(t_1^{inv}, p_1) \\ &= (t_1^{inv} + 2o_{\max} + \epsilon_2 - \epsilon_1 + o_2) - (t_1^{inv} + o_1) \\ &= 2o_{\max} + \epsilon_2 - \epsilon_1 + (o_2 - o_1). \end{aligned}$$

Choose $\epsilon_2 = \frac{\epsilon_1}{4}$ and $o_2 - o_1 = -2o_{\max} + \epsilon_2$, then:

$$clock(ts(w_2)) - clock(ts(w_1)) = \frac{-\epsilon_1}{2} < 0.$$

Thus, $ts(w_2) < ts(w_1)$ even though $w_1 \prec_H w_2$, violating the monotonicity property.

From both directions of the proof, it follows that $w_1 \prec_H w_2 \Rightarrow ts(w_1) < ts(w_2)$ holds if and only if $\delta > 4o_{\max}$. □

Theorem 2. *Algorithm 3 guarantees Property 5.2.3, i.e., $r \prec_H w \implies ts(r) < ts(w)$, if and only if $\delta \geq 4o_{\max}$.*

Proof. Let p_r and p_w denote the coordinator processes for the read operation r and the write operation w, respectively. Let p_v denote the process that assigns a timestamp to the value returned by r, with the assignment occurring at real time t, such that $clock(ts(r)) = c(t, p_v)$, as determined in line 6. Let t_r^{resp} denote the real time at which r completes, and let t_w^{inv} denote the real time at which w is invoked.

($\Leftarrow$) Suppose $\delta \geq 4o_{\max}$. Line 9 of Algorithm 3 enforces waiting at p_r until the local clock exceeds $clock(ts(r))$ by δ, implying:

$$c(t_r^{resp}, p_r) - c(t, p_v) > \delta \geq 4o_{\max}.$$

Given $r \prec_H w$, the invocation of w at time t_w^{inv} follows the response of r, so $t_r^{resp} < t_w^{inv}$. By Lemma 4, this implies:

$$c(t_w^{inv}, p_w) > c(t, p_v).$$

Since $clock(ts(w)) = c(t_w^{inv}, p_w)$ and $clock(ts(r)) = c(t, p_v)$, it follows that:

$$clock(ts(w)) > clock(ts(r)).$$

Thus, $ts(r) < ts(w)$.

($\Rightarrow$) Suppose $\delta < 4o_{\max}$, and let $\delta = 4o_{\max} - \epsilon_1$ for some $\epsilon_1 > 0$. A counterexample is constructed in which $r \prec_H w$, yet $ts(r) > ts(w)$, violating Property 5.2.3.

Let $clock(ts(r)) = c(t, p_v) = t + o_3$, where t is the real time at which r's timestamp is assigned. The algorithm requires that:

$$c(t_r^{resp}, p_r) - c(t, p_v) > \delta.$$

By Lemma 2, this implies:

$$t_r^{resp} - t > \delta - 2o_{\max}.$$

Assume w is invoked at time:

$$t_w^{inv} = t + (\delta - 2o_{\max} + \epsilon_2),$$

for some $\epsilon_2 > 0$, ensuring $r \prec_H w$. Substituting $\delta = 4o_{\max} - \epsilon_1$, we get:

$$t_w^{inv} = t + 2o_{\max} + \epsilon_2 - \epsilon_1.$$

Let o_2 and o_3 denote the clock offsets of p_w and p_v, respectively. Then:

$$\begin{aligned} clock(ts(w)) - clock(ts(r)) &= c(t_w^{inv}, p_w) - c(t, p_r) \\ &= (t + 2o_{\max} + \epsilon_2 - \epsilon_1 + o_2) - (t + o_3) \\ &= 2o_{\max} + \epsilon_2 - \epsilon_1 + (o_2 - o_3). \end{aligned}$$

Choose $\epsilon_2 = \frac{\epsilon_1}{4}$, and let $o_2 - o_3 = -2o_{\max} + \epsilon_2$, then:

$$clock(ts(w)) - clock(ts(r)) = \frac{-\epsilon_1}{2} < 0.$$

Thus, $ts(w) < ts(r)$ even though $r \prec_H w$, violating Property 5.2.2.

It follows that $r \prec_H w \Rightarrow ts(r) < ts(w)$ holds if and only if $\delta \geq 4o_{\max}$. □

Theorem 3. *Algorithm 3 guarantees Property 5.2.2, i.e., for any write w and read r, if $w \prec_H r$, then $ts(w) \leq ts(r)$, for an arbitrary δ.*

Proof. Assume $w \prec_H r$, meaning the write w completes before the read r is invoked. Let Q_w and Q_r denote the quorums used by the write and read operations, respectively.

According to Algorithm 3, a write operation w completes only after broadcasting a Write message with timestamp $ts(w)$ and receiving acknowledgments from all replicas in a quorum Q_w. This guarantees that each replica in Q_w has recorded the pair $(ts(w), v)$ in its local state by line 19, prior to the invocation of the subsequent read r.

The read operation r broadcasts Read1 and collects responses from quorum Q_r, then selects the maximum timestamp $ts(r)$ among the responses it receives. Since $Q_w \cap Q_r \neq \emptyset$, and at least one replica has $ts(w)$ or a greater timestamp, the maximum observed by r must satisfy:

$$ts(r) \geq ts(w).$$

Therefore, Algorithm 3 satisfies write-read monotonicity regardless of the value of δ. □

Theorem 4. *Algorithm 3 guarantees Property 5.2.4, i.e., for any read r_1 and read r_2, if $r_1 \prec_H r_2$, then $ts(r_1) \leq ts(r_2)$, for an arbitrary δ.*

Proof. Assume $r_1 \prec_H r_2$, meaning that read r_1 completes before read r_2 is invoked. Let Q_1 and Q_2 denote the quorums contacted by r_1 and r_2, respectively.

According to Algorithm 3, each read operation broadcasts a Read1 message and receives Read1Resp(ts_q, v_q) replies from all replicas in its quorum. The read then selects the maximum timestamp among the responses, denoted $ts(r_i) = \max_{q \in Q_i} ts_q$, for $i \in \{1, 2\}$.

Since r_1 completes before r_2 begins, each process in quorum Q_1 must have responded with its latest known timestamp $ts_q \leq ts(r_1)$ prior to the completion of r_1. If Q_1 is a *consistent quorum* such that $\forall q \in Q_1, ts_q = ts(r_1)$, then any process $q \in Q_1 \cap Q_2$ maintains a local timestamp $ts_q \geq ts(r_1)$ when responding to the Read1 request of r_2.

Otherwise, if Q_1 is not a consistent quorum, let Q_1' denote the quorum from which r_1 receives responses at line 8. Since all replicas in Q_1' have received $ts(r_1)$, any process $q \in Q_1' \cap Q_2$ maintains a local timestamp $ts_q \geq ts(r_1)$ when responding to the Read1 request of r_2.

Therefore, for any process $q \in Q_1 \cap Q_2$, its response to r_2 must satisfy $ts_q \geq ts_1$. Since $Q_1 \cap Q_2 \neq \emptyset$, the maximum timestamp returned to r_2, i.e., $ts(r_2)$, must satisfy:

$$ts(r_2) \geq ts(r_1).$$

Hence, the read-read monotonicity condition $r_1 \prec_H r_2 \Rightarrow ts(r_1) \leq ts(r_2)$ holds. This result is independent of the choice of δ, since it relies solely on quorum intersection. □

5.3.3 Discussion

By setting $\delta > 4o_{\max}$, Algorithm 3 guarantees all consistency properties defined in Section 5.2. This design enables a reduction in write latency when $o_{\max}$ is significantly smaller than the latency of a communication round-trip between replicas. For instance, in typical AWS deployments, the maximum clock offset $o_{\max}$ can be bounded to approximately 10ms using services like Amazon Time Sync Service [7]. Setting $\delta = 50$ms satisfies the condition $\delta > 4o_{\max}$. In contrast, inter-region WAN communication on AWS—e.g., between instances in `us-east-1` (N. Virginia) and `ap-southeast-2` (Sydney)— commonly incurs round-trip latencies exceeding 200ms.

Under such conditions, a write coordinated by CABD can terminate after a single round-trip of communication, as the elapsed duration already exceeds 50ms, thereby eliminating the need for an additional coordination round-trip. This results in significant performance gains in WAN environments, where network latency often dominates coordination cost.

The use of local waiting similarly affects read operations. To maintain consistency, each read must defer its response until the local clock exceeds the largest observed timestamp

by at least $\delta > 4o_{\max}$, as enforced by line 9 of Algorithm 3. However, in read-dominant workloads, this waiting period is often negligible, since the most recent write may have been issued well before the read begins.

5.4 Watermark Integrated ABD

In read-heavy workloads, requiring two coordination round-trips for read operations can introduce substantial overhead. To mitigate this, prior works [43, 61] propose passive mechanisms for handling reads. Rather than actively coordinating each read, replicas can delay responses until their local state is sufficiently up-to-date, and then serve the read directly. This class of techniques is referred to as watermark-based algorithms [61]. By avoiding message-based coordination, watermark reads significantly reduce communication overhead, making them particularly well-suited for read-heavy scenarios.

This section introduces Watermark-Integrated ABD (WABD), a hybrid protocol that integrates the CABD with watermark-based techniques. By dynamically selecting between active and passive coordination, WABD adapts to diverse workload scenarios. The hybrid design preserves all consistency guarantees while offering flexibility to balance read and write performance adaptively.

Local state maintained by process p:

1. *valMap*: a map from timestamps to their corresponding values
2. wm_p: local watermark of process p
3. gwm_p: global watermark
4. T: list of timestamps of pending write operations
5. *wmMap*: a map from process identifiers to their local watermarks
6. ts_{max}: the largest timestamp issued by process p
7. δ_g: interval between successive gossip broadcasts of the local watermark

---

Figure 5.1: Local state maintained by each process for Algorithm 4 and Algorithm 5

5.4.1 Watermark-based Read/Write Algorithm

We begin by introducing the watermark-based read/write algorithm adapted from [61]. Figure 5.1 lists the local variables maintained by each process p, including a local watermark wm_p and a global watermark gwm_p, both of which are defined as follows:

Definition 14 (Local Watermark)**.** Let W_p denote the set of write operations issued by process p. A write operation $w_i \in W_p$ is said to be *fully acknowledged* if it has been received and acknowledged by all processes. This condition is denoted by $AckByAll(w_i)$. The local watermark wm_p is defined as the highest timestamp such that

$$\forall w \in W_p, \quad ts(w) < wm_p \;\Rightarrow\; AckByAll(w).$$

Definition 15 (Global Watermark)**.** The global watermark gwm_p maintained at process p is defined as

$$gwm_p = \min_{q \in R} wmMap[q],$$

where $wmMap[q]$ stores the most recent local watermark reported by process q.

Algorithm 4 presents the watermark-based read/write algorithm. Both the read procedure (line 5) and the write procedure (line 10) complete after the global watermark gwm_p exceeds the operation's timestamp ts. Once this condition is satisfied, the `WatermarkRead` procedure returns the value associated with the largest timestamp less than ts, retrieved from *valMap*, which stores historical values. Code section from Algorithm 4 presents the procedures for managing watermarks.

The `GetTs` procedure (line 24, Algorithm 4) is invoked by both read and write procedures to generate a timestamp $ts = \langle \texttt{GetTime}(), p \rangle$, where p denotes the process identifier. If invoked by `WatermarkWrite`, the generated timestamp ts is appended to the list T (line 28) before being returned. The `AfterAckByAll` procedure removes timestamp ts from T once acknowledgements for the corresponding write are received from all processes (line 23).

Each process p periodically invokes the `GossipWatermark` procedure (line 36) to broadcast its current local watermark to peers. Upon receiving watermark values from other processes (line 20), process p invokes the `UpdateGlobalWatermark` procedure (line 30), which updates the global watermark gwm_p to the minimum value among all entries in *wmMap*, as specified in line 33.

Algorithm 4: Watermark-Based Read/Write Algorithm

```
1  Var T ← ∅
2  Var valMap ← {⟨⊥, q⟩ ↦ ⊥ | q = min(R)}
3  Var wmMap ← {q ↦ ⟨⊥, q⟩ | q ∈ R}
4  Var wm_p, gwm_p, ts_max ← ⟨⊥, p⟩
5  Procedure WatermarkRead():
6      ts ← GetTs(false)
7      wait until gwm_p > ts
       ▷ valMap.get(ts) returns the value associated with the largest timestamp
         strictly less than ts
8      v ← valMap.get(ts)
9      return v
10 Procedure WatermarkWrite(v):
11     ts ← GetTs(true)
12     broadcast Write(ts, v)
13     wait until gwm_p > ts
   ▷ Upon receiving message m from process q
14 Procedure HandleMessageWM(m):
15     switch m do
16         case m is Write(ts_q, v_q) do
17             valMap[ts_q] ← v_q
18             send WriteResp(ts_q) to q
19         case m is wm_q do
20             UpdateGlobalWatermark(q, wm_q)
21         case m is WriteResp(ts) do
22             if WriteResp(ts) has been received from all processes then
23                 AfterAckByAll(ts)
   ▷ Watermark management procedures (executed sequentially by process p):
24 Procedure GetTs(isWrite):
25     wait until GetTime() > max(clock(ts_max), clock(wm_p))
26     ts_max ← ⟨GetTime(), p⟩
27     if isWrite then
28         T.append(ts_max)
29     return ts_max
30 Procedure UpdateGlobalWatermark(q, wm_q):
31     if wm_q > wmMap[q] then
32         wmMap[q] ← wm_q
33         gwm_p ← min_{q'∈R} wmMap[q']
34 Procedure AfterAckByAll(ts):
35     T.delete(ts)
```

```
    ▷ invoked by process p every δ_g time units
36  Procedure GossipWatermark():
37      ts ← ⟨GetTime(), p⟩
38      if ¬T.isEmpty() then
39          ts ← T.head
40          wait until: ts ∉ T
41      wmMap[p] ← ts
42      wm_p ← ts
43      broadcast wm_p
```

Note that the `GetTs`, `AfterAckByAll`, `UpdateGlobalWatermark`, and `GossipWatermark` procedures are executed sequentially by process p, ensuring that no two are invoked concurrently by the same process. This sequential execution is enforced using a mutual exclusion lock, which guards access to shared state variables, including the timestamp list T, the watermark map $wmMap$, the global watermark gwm_p, and the local watermark wm_p.

5.4.2 Combining Watermark-based Algorithm with CABD

Although Algorithm 4 permits coordination-free reads, progress relies on advancing the global watermark gwm_p, which precludes liveness under replica failures. Blocking on messages from all replicas induces elevated tail latency when stragglers or overloaded nodes are present. CABD tolerates up to f replica failures using quorum-based communication but incurs delays from conservative estimates of o_{max}, designed for worst-case clock drift scenarios. While CABD enhances fault tolerance, latency may increase under adverse network or failure conditions. To address these complementary limitations, WABD integrates quorum-based and coordination-free techniques to achieve balanced performance and resilience.

Algorithm 5 defines the read and write operations in WABD. A write broadcasts its value and completes when either the completion condition of `DeltaWrite` (line 15, Algorithm 3) or that of `WatermarkWrite` (line 10, Algorithm 4) is satisfied. For reads, WABD executes `DeltaRead` (line 3, Algorithm 3) in parallel with `WatermarkRead` (line 5, Algorithm 4) and returns upon the earliest completion. This dual-path strategy minimizes latency under varying system loads. Unlike writes, `DeltaRead` is triggered probabilistically, governed by $prob_d$, computed as the sigmoid of $\frac{n_r - C}{C}$, where n_r denotes the number of pending read operations whose coordination has begun but not yet completed, and C is a configurable concurrency threshold. If $n_r < C$, `DeltaRead` is always executed. When $n_r > C$, the probability decreases, approaching zero as n_r grows significantly beyond C.

Algorithm 5: WABD

```
1  Procedure WABDRead():
2      start ← GetTime()
3      v_1, v_2 ← ⊥
4      prob_d ← 1/(1 + e^{(n_r − C)/C})
5      asynchronously run:
6          v_1 ← WatermarkRead() // line 5, Algorithm 4
7      asynchronously run with probability prob_d
8          v_2 ← DeltaRead(4o_max) // line 3, Algorithm 3
9      C_1 ← (v_1 ≠ ⊥ ∨ v_2 ≠ ⊥)
10     C_2 ← (GetTime() − start > T)
11     wait until: C_1 ∨ C_2
12     if C_1 then
13         return the value among v_1 and v_2 that is set first
14     else
           ▷ invoke DeltaRead to preserve liveness when failures inhibit watermark
             advancement
15         return DeltaRead(4o_max)
16 Procedure WABDWrite(v):
17     ts ← GetTs(true)
18     broadcast Write(ts, v)
       ▷ completion condition for DeltaWrite (line 15, Algorithm 3)
19     C_d : (WriteResp(ts) received from some quorum) ∧ (GetTime() − clock(ts) > 4o_max)
       ▷ completion condition for WatermarkWrite (line 10, Algorithm 4)
20     C_w : gwm_p > ts
21     wait until: C_d ∨ C_w
   ▷ Upon receiving message m from process q
22 Procedure HandleMessage(m):
23     switch m do
24         case m is Write(ts_q, v_q) do
25             valMap[ts_q] ← v_q
26             Update(ts_q, v_q)
27             send WriteResp(ts_q) to q
28         case m ∈ {wm_q, WriteResp(ts_q)} do
29             HandleMessageWM(m) // line 14, Algorithm 4
30         case m ∈ {Read1, Read2(ts_q, v_q), Read2Resp(ts_q)} do
31             HandleMessageCABD(m) // line 23, Algorithm 3
```

This design enables WABD to prefer coordination-free reads under high load, reverting to quorum-based reads when load is low. Additionally, WABD triggers `DeltaRead` if gwm_p remains stagnant for a preset timeout $\mathcal{T}$, thereby ensuring liveness when failures prevent watermark advancement (line 15). The value of $\mathcal{T}$ should reflect anticipated clock drift and network delay in the deployment environment. A smaller $\mathcal{T}$ improves responsiveness under failure, but may trigger unnecessary active coordination for read operations. In contrast, a larger $\mathcal{T}$ reduces such coordination overhead, but may prolong recovery under persistent failures. The current implementation sets $\mathcal{T}$ to twice the estimated round-trip time between two distant replicas. Future work will explore adaptive tuning of $\mathcal{T}$ based on observed network latency and real-time system behavior.

5.4.3 Correctness of WABD

This section analyzes the consistency guarantees of WABD under the bounded clock offset assumption. In the watermark-based algorithm, each read operation r is associated with a timestamp $\tau(r)$ (see line 6). To distinguish this from $ts(r)$ as defined in Section 5.2.1, $\tau(r)$ denotes the timestamp used in watermark-based coordination. For each operation o, $inv(o)$ denotes the event at which a timestamp is assigned to o, and $resp(o)$ denotes the event at which coordination for o completes. The analysis uses wm_i^t and gwm_i^t to denote the local and global watermarks at process $i \in R$ at real time t, respectively.

Since Algorithm 5 relies on the same watermark management procedures as those in Algorithm 4 (starting after line 24), the analysis begins by establishing the properties ensured by watermark management in Algorithm 4:

Lemma 5. *Algorithm 4 ensures the following: timestamps are appended to T in strictly increasing lexicographic order. This implies that if ts_i is appended to T after ts_j, then $ts_i > ts_j$.*

Proof. Timestamps are appended to T only by the `GetTs` procedure in Line 24, which is executed sequentially by a process. Each invocation waits until `GetTime()` exceeds $clock(ts_{max})$ (line 25), where ts_{max} is the timestamp most recently appended to T. Thus, timestamps are appended to T in strictly increasing lexicographic order. □

Lemma 6. *Algorithm 4 ensures the following: for each process p, if the local watermark wm_p equals ts, then any write operation coordinated by p with timestamp less than ts has been acknowledged by all processes.*

Proof. Assume, for contradiction, that a write w, coordinated by process p, with $ts(w) < ts$, remains unacknowledged by some process when the local watermark wm_p is updated to ts in line 42. Let gt denote the execution of `GetTs` in line 24 that assigns the timestamp $ts(w)$ to w. Let gw denote the execution of `GossipWatermark` in line 36 that updates wm_p to ts.

The state of the timestamp list T at the time of gw can be categorized as follows:

1. T is empty when gw is executed. In this case, because gt and gw are sequential, gt must start after gw completes. Otherwise, $ts(w)$ would have been appended to T and not yet deleted by line 35, as the write has not been acknowledged by all processes.

   However, both ts and $ts(w)$ are constructed using `GetTime()`. The return value of `GetTime()` is monotonically increasing, assuming the local clock does not go backward. Since gt must start after gw completes, it must hold that $ts(w) \geq ts$.

2. T is non-empty when gw is executed. Then $ts = T.head$, as defined in line 39. According to Lemma 5, $ts(w) \notin T$ because $ts(w) < ts$. As a result, gt must start after gw completes. However, according to Lemma 5, it must hold that $ts(w) > ts$.

□

Lemma 7. *Algorithm 4 ensures the following: for each process p, the value of local watermark wm_p is monotonically non-decreasing with respect to real time.*

Proof. Assume, for contradiction, that the local watermark wm_p decreases at some point in real time. Let $ts_1, ts_2, ts_3, \ldots$ denote the sequence of values assigned to wm_p (line 42), ordered by strictly increasing update time. Then there must exist an index n such that $ts_n > ts_{n+1}$. Since only the `GossipWatermark` procedure (line 36) updates wm_p, let gw_n and gw_{n+1} denote the execution of `GossipWatermark` that updates wm_p to ts_n and ts_{n+1}, respectively. The following cases are considered:

1. T is empty during both gw_n and gw_{n+1}. Due to sequential execution, gw_n completes before gw_{n+1} starts, and both timestamps ts_n and ts_{n+1} are assigned using `GetTime()` via line 37. Since the return value of `GetTime()` is monotonically increasing, assuming the local clock does not go backward, it follows that $ts_n \leq ts_{n+1}$.

2. T is non-empty during gw_n, while T is empty during gw_{n+1}. Let gt_n denote the execution of `GetTs` (line 24) that appends ts_n to T. The execution gt_n must complete before gw_n starts; otherwise, $ts_n \notin T$ and could not be retrieved by gw_n at line 39.

Since T is empty during gw_{n+1}, ts_{n+1} is set by gw_{n+1} using `GetTime()` via line 37. As gt_n completes before gw_{n+1} starts, due to sequential execution, it follows that $ts_n \leq ts_{n+1}$, since `GetTime()` is monotonically increasing.

3. T is empty during gw_n, and ts_n is assigned using `GetTime()` via line 37, whereas T is non-empty during gw_{n+1}. Let gt_{n+1} denote the `GetTs` invocation in line 24 that appends ts_{n+1} to T, which is subsequently retrieved via line 39 during gw_{n+1}.

   Since T is empty during gw_n, and $ts_{n+1} \in T$ during gw_{n+1}, sequential execution implies that gt_{n+1} must be invoked after gw_n completes. Otherwise, ts_{n+1} would not appear in T during gw_{n+1}, yielding a contradiction.

   Given that `GetTime()` returns monotonically increasing values, and ts_n is assigned prior to ts_{n+1}, it follows that $ts_n \leq ts_{n+1}$.

4. T is non-empty during both gw_n and gw_{n+1}. Let gt_n and gt_{n+1} denote the executions of `GetTs` that append ts_n and ts_{n+1} to T, which are retrieved via line 39 during gw_n and gw_{n+1}, respectively.

   Since $ts_n > ts_{n+1}$, gt_{n+1} must complete before gt_n starts, according to Lemma 5. Additionally, gt_n completes before gw_n starts; otherwise, gw_n cannot retrieve ts_n from T via line 39. As a result, gt_{n+1} completes before gw_n starts.

   Since gw_{n+1} starts after gw_n completes, ts_{n+1} remains in T until gw_{n+1} completes. However, in this case, gw_n cannot retrieve ts_n via line 39, because $ts_n > ts_{n+1}$ and thus cannot be $T.head$, according to Lemma 5, contradicting the fact that gw_n updates wm_p to ts_n.

□

Lemma 8. *Algorithm 4 ensures the following: if $gwm_p > ts$ at real time t, for some $p \in R$, then $wm_q > ts$ at real time t, for all $q \in R$.*

Proof. Let $wmMap_p^t$ denote the watermark map maintained by process p at time t, as initialized in line 3. The argument proceeds by contradiction. Assume $gwm_p^t > ts$ for some $p \in R$, and suppose there exists $q \in R$ such that $wm_q^t \leq ts$.

According to line 33 in Algorithm 4,

$$gwm_p^t \leftarrow \min_{i \in R} wmMap_p^t[i].$$

Thus, for each $i \in R$, including q, it holds that $wmMap_p^t[q] = wm_q^{t'}$ for some $t' < t$. Hence,

$$wm_q^{t'} \geq gwm_p^t > ts.$$

By Lemma 7, local watermark values are monotonically non-decreasing with respect to real time. So

$$wm_q^t \geq wm_q^{t'} > ts,$$

which contradicts the assumption $wm_q^t \leq ts$. □

The following presents proofs for the monotonicity properties defined in Section 5.2.1 and ensured by Algorithm 5:

Lemma 9. *Algorithm 5 guarantees Property 5.2.1:* $w_1 \prec_H w_2 \Rightarrow ts(w_1) < ts(w_2)$.

Proof. Let p_1 and p_2 denote the coordinator processes of w_1 and w_2, respectively. Let t_1^{inv} and t_1^{resp} denote the real times of $inv(w_1)$ and $resp(w_1)$, respectively. Let t_2^{inv} denote the real time of $inv(w_2)$.

1. **Case 1:** The write w_1 completes when condition C_d is satisfied (line 19). Condition C_d enforces the same termination criterion at $resp(w_1)$ as `DeltaWrite` with δ set to $4o_{\max}$. As a result, the following holds:

   $$c(t_1^{resp}, p_1) - c(t_1^{inv}, p_1) > 4o_{\max}$$

   Since $w_1 \prec_H w_2$, it follows that $t_1^{inv} < t_1^{resp} < t_2^{inv}$. By Lemma 4, it follows that:

   $$c(t_2^{inv}, p_2) > c(t_1^{inv}, p_1)$$

   Given that $clock(ts(w_2)) = c(t_2^{inv}, p_2)$ and $clock(ts(w_1)) = c(t_1^{inv}, p_1)$, it follows that $ts(w_1) < ts(w_2)$.

2. **Case 2:** Let wm_i^t and gwm_i^t denote the values of the local and global watermarks at process $i \in R$ at real time t, respectively.

   The write w_1 completes when condition C_w is satisfied (line 20). Since w_1 completes under C_w, it follows that $gwm_{p_1}^{t_1^{resp}} > ts(w_1)$. By Lemma 8, the local watermark $wm_{p_2}^{t_1^{resp}} > ts(w_1)$.

Given that $w_1 \prec_H w_2$, it follows that $t_1^{resp} < t_2^{inv}$. According to line 25 in Algorithm 4 and Lemma 7, the timestamp $ts(w_2)$, assigned at time t_2^{inv} by `GetTs` (line 24), satisfies $wm_{p_2}^{t_1^{resp}} \leq wm_{p_2}^{t_2^{inv}} < ts(w_2)$. Therefore,

$$ts(w_1) < wm_{p_2}^{t_1^{resp}} \leq wm_{p_2}^{t_2^{inv}} < ts(w_2),$$

and Property 5.2.1 holds.

□

Lemma 10. *Algorithm 5 guarantees Property 5.2.2:* $w \prec_H r \Rightarrow ts(w) \leq ts(r)$.

Proof. Let p_w and p_r denote the coordinator processes of w and r, respectively. Let t_w^{inv} and t_w^{resp} denote the real times of $inv(w)$ and $resp(w)$, respectively. Let t_r^{inv} and t_r^{resp} denote the real times of $inv(r)$ and $resp(r)$, respectively.

1. **Case 1:** w completes when C_d is satisfied (line 19).
   (a) **Subcase 1:** r is a `DeltaRead` operation (line 3, Algorithm 3). In this case, the condition C_d ensures the same completion criteria as `DeltaWrite`. Consequently, Theorem 3 applies, and Property 5.2.2 is satisfied.
   (b) **Subcase 2:** r is a `WatermarkRead` operation (line 5, Algorithm 4). Since w completes under condition C_d, which enforces the same termination criterion at $resp(w)$ as `DeltaWrite` with $\delta = 4o_{\max}$, the following holds:

   $$c(t_w^{resp}, p_w) - c(t_w^{inv}, p_w) > 4o_{\max}.$$

   Given that $w \prec_H r$, it follows that $t_w^{resp} < t_r^{inv}$. By Lemma 4,

   $$c(t_r^{inv}, p_r) > c(t_w^{inv}, p_w).$$

   Since $clock(\tau(r)) = c(t_r^{inv}, p_r)$ and $clock(ts(w)) = c(t_w^{inv}, p_w)$, it follows that

   $$\tau(r) > ts(w).$$

   Let $t < t_r^{resp}$ denote the real time at which line 7 of Algorithm 4 is satisfied in r, such that $gwm_{p_r}^t > \tau(r)$.
   By Lemma 8,

   $$wm_{p_w}^t > \tau(r).$$

   Since $ts(w) < \tau(r) < wm_{p_w}^t$, then by Lemma 6, w has been acknowledged by all processes, including p_r, at time t. Therefore, r returns the value written by w, unless a write with a higher timestamp exists. Hence, Property 5.2.2 holds.

2. **Case 2:** w completes when C_w is satisfied (line 20).

   (a) **Subcase 1:** r is a `DeltaRead` operation (line 3, Algorithm 3). In this case, condition C_w ensures that $gwm_{p_w}^{t_w^{resp}} > ts(w)$. According to Lemma 7, w has been acknowledged by all processes at time t_w^{resp}.

   Since r reads from a quorum of processes and $t_w^{resp} < t_r^{inv}$, it must return the value written by w, provided no write with a higher timestamp exists in the quorum. Therefore, Property 5.2.2 holds.

   (b) **Subcase 2:** r is a `WatermarkRead` operation (line 5, Algorithm 4). Since w completes under condition C_w, it follows that $gwm_{p_w}^{t_w^{resp}} > ts(w)$, and w has been acknowledged by all processes at time t_w^{resp}. By Lemma 8, the local watermark $wm_{p_r}^{t_w^{resp}} > ts(w)$.

   Given $w \prec_H r$, it follows that $t_r^{inv} > t_w^{resp}$. According to line 25 in Algorithm 4 and Lemma 7, the timestamp $\tau(r)$ assigned by `GetTs` at t_r^{inv} satisfies $ts(w) < wm_{p_r}^{t_w^{resp}} \leq wm_{p_r}^{t_r^{inv}} < \tau(r)$.

   Since the write w has been acknowledged by p_r at time t_r^{inv}, and $ts(w) < \tau(r)$, line 8 of Algorithm 4 ensures that the read operation r returns the value written by w, provided no write with a higher timestamp is present in *valMap* at p_r. Therefore, Property 5.2.2 holds.

□

Lemma 11. *Algorithm 5 guarantees Property 5.2.3:* $r \prec_H w \implies ts(r) < ts(w)$.

Let p_w and p_r denote the coordinator processes of w and r, respectively. Let p_v denote the process that coordinates the write w' whose value is returned by r. Assume w' is invoked at real time t, such that $clock(ts(r)) = c(t, p_v)$, as determined in line 6. Let t_r^{inv} and t_r^{resp} denote the real times of $inv(r)$ and $resp(r)$, respectively. Let t_w^{inv} and t_w^{resp} denote the real times of $inv(w)$ and $resp(w)$, respectively.

Proof.

1. **Case 1:** r is a `DeltaRead` (line 3, Algorithm 3), invoked with δ set to $4o_{\max}$. Line 9 of Algorithm 3 enforces waiting at p_r implying:

$$c(t_r^{resp}, p_r) - c(t, p_v) > \delta \geq 4o_{\max}.$$

Given $r \prec_H w$, the invocation of w at time t_w^{inv} follows the response of r, so $t_r^{resp} < t_w^{inv}$. By Lemma 4, this implies:

$$c(t_w^{inv}, p_w) > c(t, p_v).$$

Since $clock(ts(w)) = c(t_w^{inv}, p_w)$ and $clock(ts(r)) = c(t, p_v)$, it follows that:

$$clock(ts(w)) > clock(ts(r)).$$

Thus, $ts(r) < ts(w)$.

2. **Case 2:** r is a `WatermarkRead` (line 5, Algorithm 4). Let $gwm_{p_r}^{t_r^{resp}}$ denote the global watermark maintained by process p_r at time t_r^{resp}. Line 7 of Algorithm 4 ensures that $gwm_{p_r}^{t_r^{resp}} > \tau(r)$.

   By Lemma 8,

   $$wm_{p_w}^{t_r^{resp}} > \tau(r).$$

   Given $r \prec_H w$, it follows that $t_r^{resp} < t_w^{inv}$. According to Line 25 in Algorithm 4 and Lemma 7, the timestamp $ts(w)$ assigned at t_w^{inv} by `GetTs` (Line 24) satisfies

   $$wm_{p_w}^{t_r^{resp}} \leq wm_{p_w}^{t_w^{inv}} < ts(w).$$

   Moreover, since *valMap* thcontains at least one value with a timestamp less than $\tau(r)$ (Line 2, Algorithm 4), line 8 of Algorithm 4 ensures that $ts(w') < \tau(r)$.

   Therefore, the following inequality holds:

   $$ts(r) = ts(w') < \tau(r) < wm_{p_w}^{t_r^{resp}} \leq wm_{p_w}^{t_w^{inv}} < ts(w),$$

   and Property 5.2.3 holds.

□

Lemma 12. *Algorithm 5 guarantees Property 5.2.4:* $r_1 \prec_H r_2 \Rightarrow ts(r_1) \leq ts(r_2)$.

Proof. Let p_1 and p_2 denote the coordinator processes of the read operations r_1 and r_2, respectively. Let p_w denote the coordinator of the write w whose value is returned by r_1. Assume w is invoked at real time t, such that $clock(ts(r_1)) = c(t, p_w)$, as determined in line 6. Let t_1^{inv}, t_1^{resp} and t_2^{inv}, t_2^{resp} denote the invocation and response times of r_1 and r_2, respectively.

1. **Case 1:** Both r_1 and r_2 are `DeltaRead` operations (line 3, Algorithm 3). This case is covered by Theorem 4, and Property 5.2.4 holds.

2. **Case 2:** r_1 is a `DeltaRead` invoked with $\delta = 4o_{max}$ (line 3, Algorithm 3) and r_2 is a `WatermarkRead` (line 5, Algorithm 4). Line 9 of Algorithm 3 enforces waiting at p_1 implying:
$$c(t_1^{resp}, p_1) - c(t, p_w) > \delta \geq 4o_{\max}.$$
Given $r_1 \prec_H r_2$, so $t_1^{resp} < t_2^{inv}$. By Lemma 4, this implies:
$$c(t_2^{inv}, p_2) > c(t, p_w).$$
Since $clock(\tau(r_2)) = c(t_2^{inv}, p_2)$ and $clock(ts(w)) = c(t, p_w)$, it follows that:
$$clock(\tau(r_2)) > clock(ts(w)).$$
Thus, $ts(r_1) = ts(w) < \tau(r_2)$.
Line 7 of Algorithm 4 ensures that for some time t' with $t_2^{inv} < t' < t_2^{resp}$, prior to executing line 8, the condition
$$\tau(r_2) < gwm_{p_2}^{t'}$$
is satisfied.
By Lemma 8, this implies
$$wm_{p_w}^{t'} > \tau(r_2) > ts(w).$$
According to Lemma 6, write w must have been acknowledged by all processes, including p_2, at time t' before line 8 is executed. Therefore, r_2 returns the value written by w, or a write with a higher timestamp.
Hence,
$$ts(r_1) = ts(w) \leq ts(r_2),$$
and Property 5.2.4 holds.

3. **Case 3:** r_1 is a `WatermarkRead`.
Line 7 of Algorithm 4 ensures that
$$\tau(r_1) < gwm_{p_1}^{t'}.$$
for some time t' with $t_2^{inv} < t' < t_2^{resp}$.

According to line 8, it follows that

$$ts(w) < \tau(r_1).$$

By Lemma 8,

$$wm_{p_w}^{t'} > \tau(r_1) > ts(w).$$

By Lemma 6, write w has been acknowledged by all processes, including p_2, by time $t' < t_1^{resp}$. Since $r_1 \prec_H r_2$, and $t_1^{resp} < t_2^{inv}$, read r_2 returns the value written by w, unless a write with a higher timestamp is present.

Therefore,

$$ts(r_1) = ts(w) \leq ts(r_2),$$

and Property 5.2.4 holds.

□

Theorem 5. *Algorithm 5 guarantees linearizability.*

Proof. The proof follows directly from Lemma 9, Lemma 10, Lemma 11, Lemma 12 and Lemma 1. □

Theorem 6. *Algorithm 5 guarantees liveness:* `WABDRead` *and* `WABDWrite` *eventually terminate in any environment that satisfies the assumptions described in Section 6.3.*

Proof. For `WABDRead` (line 1), termination occurs when either condition C_1 or C_2 is satisfied, as specified in line 11. The timeout condition C_2 in line 10 is eventually satisfied after $\mathcal{T}$ units of local time from the invocation.

In such case, line 15 invokes `DeltaRead`. The `DeltaRead` procedure completes within at most two rounds of communication with a majority $(f+1)$ of processes, and terminates once the local clock advances by at least $4o_{\max}$ beyond the timestamp obtained in the first round. Given that at most f processes may fail, and the local clock is monotonic, these conditions are eventually satisfied, ensuring the termination of `DeltaRead`.

`WABDWrite` (line 16) terminates when either C_w or C_d is satisfied, as specified in line 21. The timeout condition C_d (line 19) is eventually satisfied, given that at most f processes may fail and the local clock is monotonic. □

5.5 Evaluation

We evaluate shared register protocols discussed in previous sections: ABD (Algorithm 1), WABD (Algorithm 5), CABD (Algorithm 3). We also include EPaxos [119], as well as its optimized version EPaxosTOQ [151], and Watermark [61] (Algorithm 4) in our candidates for reference. The evaluation addresses two primary objectives, each linked to specific research questions:

Objective 1: Assess the performance advantages of shared register protocols (ABD, WABD) compared with consensus-based protocols (EPaxos, EPaxosTOQ) across various key-value (KV) workload scenarios. This objective is guided by the following research question: **RQ1:** To what extent do shared register protocols outperform consensus-based protocols under different workload conditions? (Section 5.5.2, Section 5.5.3)

Objective 2: Evaluate the effectiveness of optimizations introduced in WABD, guided by two specific research questions: **RQ2:** How does WABD perform relative to alternative protocols regarding throughput, latency, and clock offset sensitivity? (Section 5.5.5) **RQ3:** How effectively does WABD's mode-switching mechanism adapt to fluctuating client workloads to maintain consistent system performance? (Section 5.5.4)

5.5.1 Experiment Setup

For our experiments, up to five Amazon Elastic Compute Cloud (EC2) regions were used: Ireland (IR), Oregon (OR), Virginia (VA), Singapore (SG), and Tokyo (TK). We considered two types of deployment environments: 1) Small replication factor: In this case, the replica processes were deployed in regions VA, OR, and IR, which are geographically close to each other. 2) Large replication factor: For this scenario, replica processes were deployed in all regions, including VA, OR, IR, SG, and TK.

Each region involved an m4.xlarge VM instance equipped with four 2.4 GHz Intel Xeon E5-2676v3 processors and 16 GB of RAM. The VMs operated on Amazon Linux 2023 as the operating system, and the golang version used was 1.17.2. On each VM, both a client process and a replica were executed. The client process communicated exclusively with its co-located replica process and was configured to issue requests at a specific rate in an open loop. To simulate an environment with a bounded clock offset, we synchronized each VM's clock with the same time server using the time synchronization tool `chrony` [1]. An offset value within the range $[1 - o_{max}, o_{max} - 1]$ was randomly assigned to each VM's clock value (this offset does not apply to EPaxosTOQ, as its optimization depends on synchronized clocks). In our experiments, we set o_{max} to either 10 ms or 50 ms, which respectively

stands for P50 percentile and P99 percentile average clock offset on AWS according to recent measurements [7]. We employed YCSB workload [42] to evaluate all the candidates. The keyspace size was set to 10K keys, with both keys and values being 8 bytes in length. The workloads employed in the evaluation were characterized by varying read-write ratios, with key access patterns modeled using a Zipfian distribution.

5.5.2 Throughput

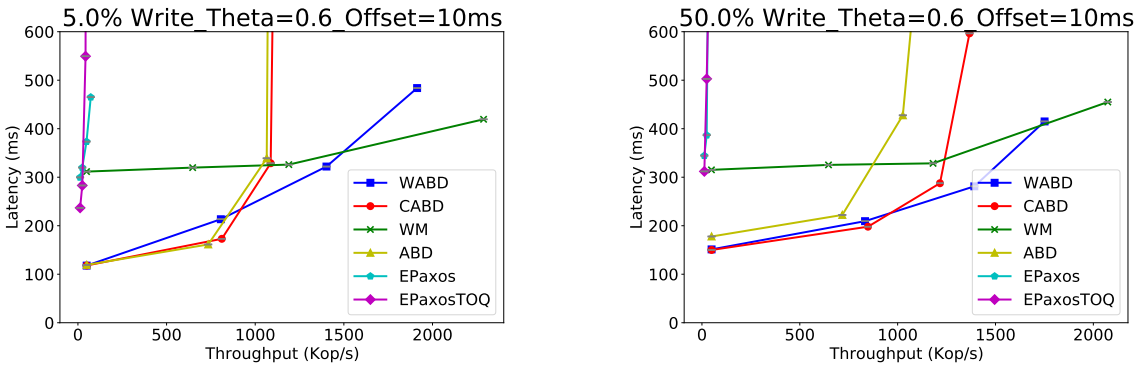


Figure 5.2: Throughput versus mean latency, $n = 5$, $o_{max} = 10$ms.

For this experiment, commands were batched with a maximum of 100 operations per batch. The maximum clock offset (o_{max}) was configured to 10 ms, and the Zipfian parameter (θ) was set to 0.6. Each data point represents an average of three runs, with each run lasting for 15 seconds after a warm-up period of 5 seconds for each VM.

Figure 5.2 illustrates the mean latency at different throughput levels for each of the candidate algorithms in a 5-replica cluster. EPaxos and EPaxosTOQ demonstrate rapidly saturating throughput as the client load increases. EPaxos achieves a peak throughput of 72 thousand operations per second (Kop/s) under a read-heavy workload and 42 Kop/s under a balanced workload. In comparison, EPaxosTOQ achieves a peak throughput of 66 Kop/s for the read-heavy workload and 59 Kop/s for the balanced workload. For EPaxos, the coarse-grained processing and immediate sub-effects of write operations lead to an increasing number of operations committed on the slow path. This amplifies the negative impact of the stable ordering constraint inherent in consensus protocols, as each operation requires a longer waiting time for all preceding operations in the command log to become committed and executed. EPaxosTOQ's execution graph pruning and time-ordered queueing techniques excel at minimizing latency, particularly at lower throughput levels

(below 10Kop/s). However, these techniques have little impact on improving maximum throughput.

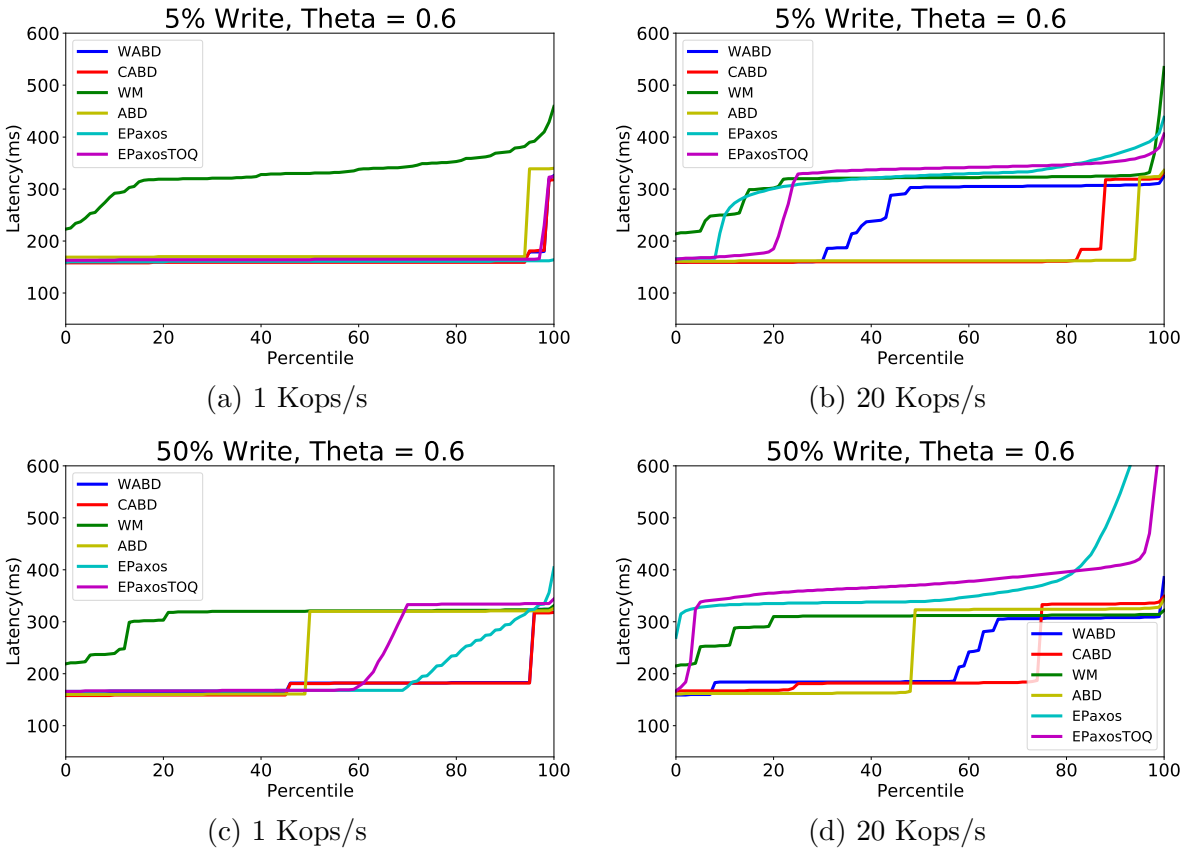
In comparison, ABD achieves a peak throughput of 1037 Kop/s for the read-heavy workload and 1121 Kop/s for the balanced workload, which is roughly 14 times and 25 times greater than that of EPaxos, respectively. CABD outperforms ABD, reaching a peak throughput of 1370 Kop/s for the balanced workload due to its low communication overhead for write operations.

The Watermark algorithm achieves the highest peak throughput of almost 2 Mop/s for both workload types due to its coordination-free approach to handling read operations. WABD strikes a middle ground between CABD and Watermark, achieving 1913 Kops/s under the read-heavy workload and 1750 Kops/s under the balanced workload. However, WABD offers latency comparable to CABD, which is nearly half the latency of Watermark in low-throughput scenarios. Additionally, WABD achieves peak throughput on par with the watermark-based approach, approximately 40% higher than that of CABD.

5.5.3 Latency

Figure 5.3 demonstrates the latency distribution of each of our candidate algorithms. The Zipfian parameter θ is set to 0.6, and the maximum clock offset $o_{\max}$ is configured to 10 ms. Under low client load, each client issues requests at a rate of 1 Kop/s, whereas under high client load, the rate increases to 20 Kop/s. Commands are batched with a maximum of 10 operations per batch. Latency statistics are collected from the client located in the SG region, where the benefits of reduced communication round-trips are most pronounced.

When the client load is low, all candidates, except for the Watermark algorithm, achieve a similar latency distribution. This is because both read and write operations in the Watermark algorithm rely on the advancement of global watermarks, which depend on the latest watermark from the most distant replica. As the client load increases, the latency of EPaxos and EPaxosTOQ rapidly increases while shared register protocols maintain a similar latency distribution to that in low client load scenarios. The reasons for this are discussed in Section 4.2.5 and Section 4.2.6. Figure 5.3c and Figure 5.3d also demonstrate that CABD significantly reduces write latency by eliminating one round-trip of communication under a balanced read-write ratio.



(a) 1 Kops/s

(b) 20 Kops/s

(c) 1 Kops/s

(d) 20 Kops/s

Figure 5.3: Latency distribution varied by client load, $n = 5$, $o_{max} = 10$ms.

5.5.4 The Effectiveness of Mode Switching In WABD

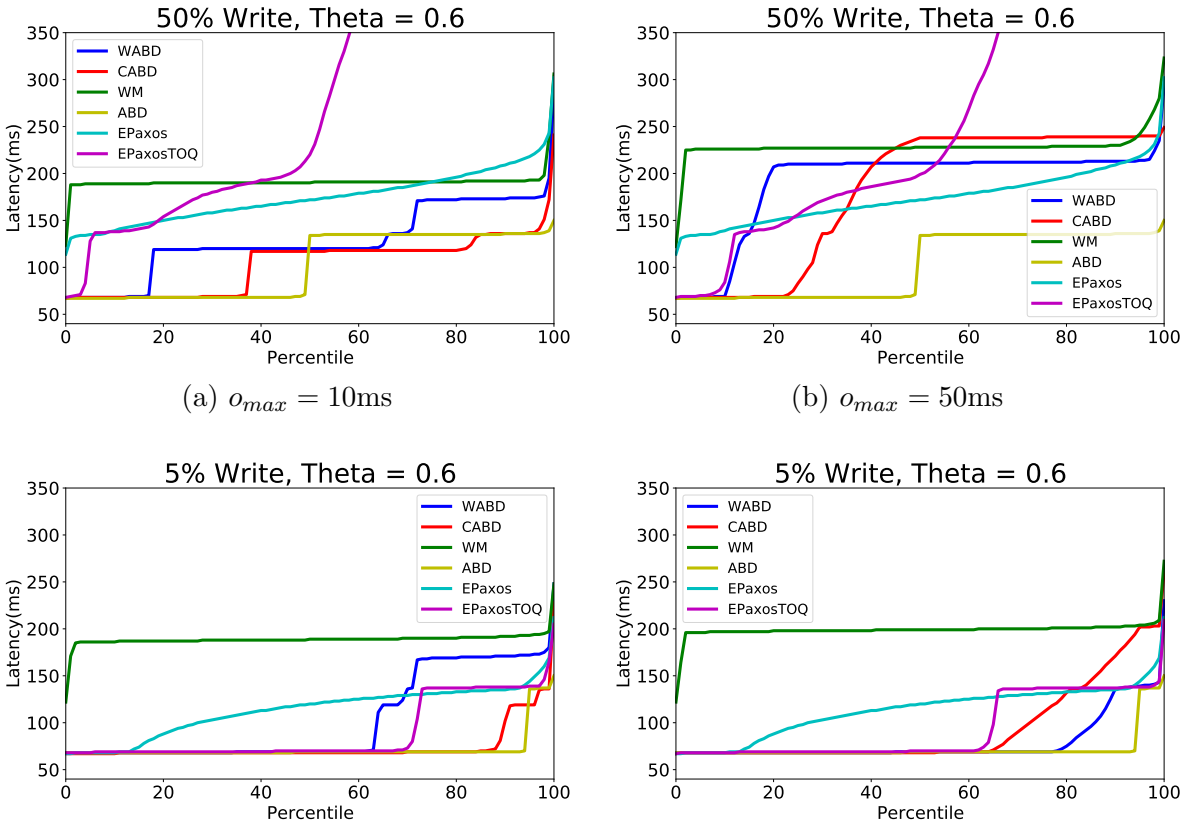
WABD is a hybrid algorithm that combines the CABD algorithm and the Watermark algorithm. In WABD, the invocation of `DeltaRead` is determined by a dynamic probability $prob_d$, which is inversely proportional to the client load. In the implementation, C is configured to be 500 times the maximum batch size to illustrate the intended latency-throughput trade-off.

Based on the results shown in Figure 5.2, both for read-heavy workloads (5% write) and balanced workloads (50% write), WABD achieves higher peak throughput compared to ABD and CABD. However, WABD's throughput reaches saturation earlier than the Watermark algorithm due to the occasional invocation of `DeltaRead`. Regarding latency, Figure 5.3 illustrates that WABD maintains a similar latency distribution to that of ABD

and CABD when the client load is low. As the client load increases, WABD exhibits a latency distribution that lies between CABD and the Watermark algorithms, striking a middle ground before reaching saturation.

5.5.5 The Impact of Maximum Clock Offset

This experiment analyzes the impact of increasing the maximum clock offset on the latency distribution of the CABD, Watermark, and WABD algorithms.

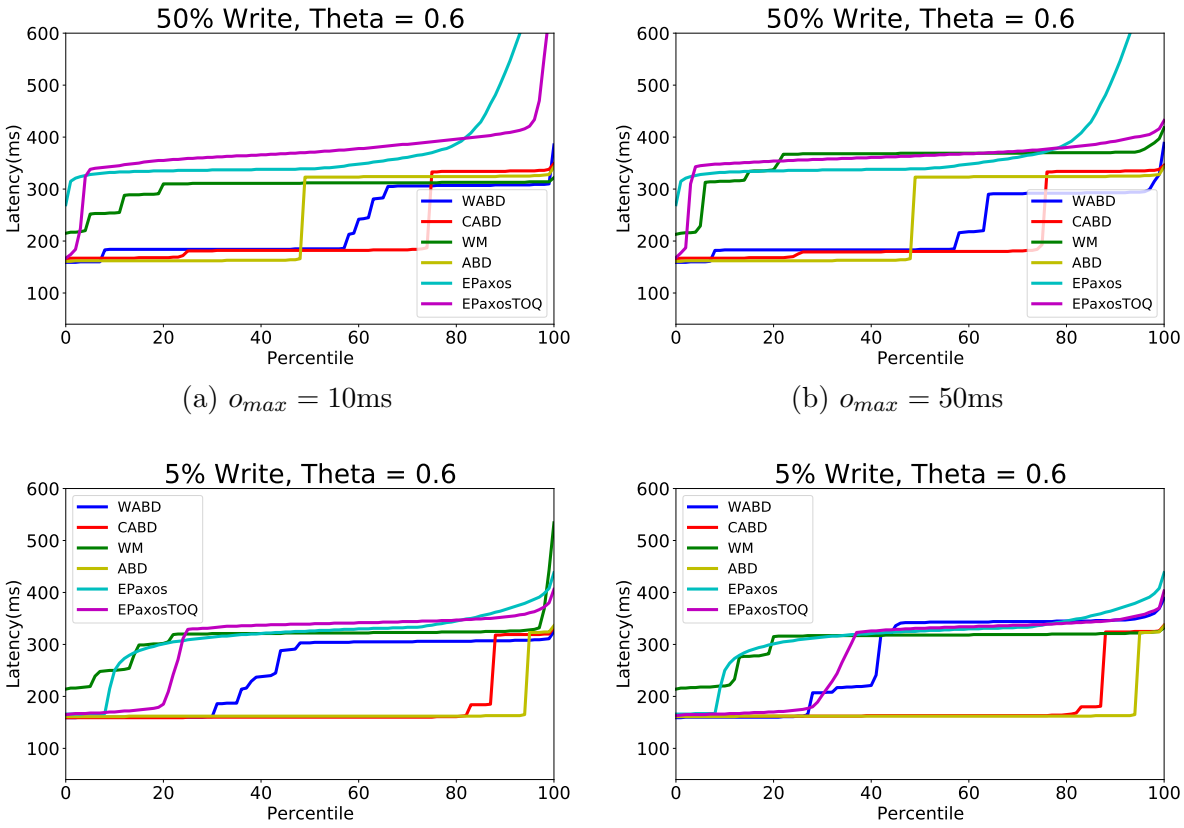


(a) $o_{max} = 10$ms (b) $o_{max} = 50$ms

(c) $o_{max} = 10$ms (d) $o_{max} = 50$ms

Figure 5.4: Impact of Clock Offset on Latency for Replication Factor $N = 3$.

For comparison, the latency distributions of ABD and EPaxos are also included, as these algorithms are not influenced by clock offset. The Zipfian parameter θ is set to 0.6. Clients issue requests at a rate of 20K operations per second, and latency statistics are



(a) $o_{max} = 10$ms (b) $o_{max} = 50$ms

(c) $o_{max} = 10$ms (d) $o_{max} = 50$ms

Figure 5.5: Impact of Clock Offset on Latency for Replication Factor $N = 5$.

collected from clients located in the **SG** region. Figure 5.4 and Figure 5.5 present the results for replication factors of 3 and 5, respectively. In both cases, the Zipfian parameter θ is fixed at 0.6 throughout the experiment.

When examining a balanced workload, increasing o_{max} from 10 ms to 50 ms has the most substantial impact on the CABD, as well as WABD algorithm. This is because the `DeltaWrite` operation in CABD always waits for the $4o_{max}$ period. In contrast, the impact on read operations is less significant for the CABD algorithm. The `DeltaRead` operation only needs to wait until its local clock is $4o_{max}$ ahead of the associated value's clock value, which can be further back in time when writes are infrequent. Consequently, the impact of increasing o_{max} on the CABD algorithm is less pronounced in read-heavy workloads. In the Watermark algorithm, waiting for responses from all replica processes

in the `WatermarkWrite` operation is already time-consuming. As a result, the Watermark algorithm is less affected by increasing o_{max} when handling workloads with a balanced read-write ratio. In a larger cluster with distant peers, the increasing clock offset's impact becomes less significant for all the algorithms. This is because the latency of each operation is already high due to larger network delays or waiting for more responses to ensure the quorum intersection property. Thus, the cost of waiting out the $4o_{max}$ period becomes relatively less expensive.

5.6 Chapter Conclusion

In this chapter, potential optimizations were explored to reduce communication overhead in shared register protocols. Assuming that each replica's clock offset is bounded, a strategy was proposed to eliminate one coordination round-trip for write operations by waiting through an uncertainty period determined by the maximum clock offset. WABD was introduced as a protocol that incorporates this optimization into the ABD algorithm and applies a local-read technique inspired by prior work. Through an adaptive mode-switching mechanism, WABD achieves low latency under light client load and high throughput under heavy load.

To evaluate the performance of WABD, ABD, and EPaxos in handling read-write workloads across diverse environments, experiments were conducted on AWS. The results show that the shared register protocol ABD achieves significantly higher throughput, ranging from 14 to 25 times greater than the consensus protocol EPaxos. Additionally, WABD achieves approximately 1.6 times the throughput of ABD while maintaining comparable latency, especially under low-load conditions.

These results highlight the performance limitations of consensus protocols in key-value workload scenarios and raise important considerations about their appropriateness for such evaluations. Although key-value workloads are commonly used in benchmarking, they may not capture the full capabilities of consensus protocols, which are designed for stronger consistency guarantees and more complex coordination primitives.

Chapter 6

Dolphin: Efficient Processing of Oracle Data Feeds for Chained BFT Protocols

Blockchain technology is a decentralized distributed ledger that securely records transactions in an immutable, chronological order. It eliminates the need for a central authority by using a network of nodes to validate and append data. Blockchains are essentially distributed peer-to-peer (P2P) systems built on State Machine Replication (SMR). SMR replicates the state of a blockchain across a network of nodes, ensuring service availability and consistency. It relies on consensus protocols to enable nodes to agree on a shared state. To achieve decentralization, blockchain systems operate collectively, with nodes functioning without mutual trust. Consequently, Byzantine Fault Tolerant (BFT) consensus protocols address failures caused by arbitrary or malicious node behavior. Over the years, numerous BFT consensus protocols [34, 93, 114, 46, 3, 154, 38] have been proposed to tackle this challenge. Although robust against malicious actions, classical protocols suffer from high message complexity, limited scalability, and leader fairness issues, making them inefficient for large-scale applications. To improve performance, recent BFT protocols for blockchains adopt a chain-like log structure inspired by Nakamoto consensus [123]. These protocols, known as chained BFT (cBFT) [161, 111, 146, 89, 163], offer better scalability and are gaining popularity in the blockchain community.

As blockchain technology advances, many applications require integration with external data. Blockchain oracles address this challenge by connecting blockchains to the external world, providing authenticated data feeds (ADF) [162, 144] to enable transactions using

real-time off-chain information. Oracle data feeds support diverse use cases, including Decentralized Finance (DeFi), supply chains, insurance, gaming, and IoT Applications. To support these applications, there will be increasing volumes of blockchain transactions that rely on real time blockchain data feeds, which we refer to as data feed dependent transactions (DDTs). For instance, Decentralized Finance (DeFi) platforms utilize real-time data to execute transactions, trade assets, and provide liquidity in an automated manner. Smart contracts on these platforms automatically execute trades based on live market conditions, leading to instantaneous asset transfers without requiring intermediaries.

However, using cBFT consensus presents several challenges in handling DDTs. First, in modern blockchain systems, data feed updates are treated as ordinary transactions, requiring full replication across the network. This replication is achieved either through the consensus process or via a gossip protocol. However, not all data feeds are consumed by DDTs, leading to unnecessary replication that wastes on-chain network resources. Second, cBFT protocols rely on a leader role to propose new blocks. A malicious leader can deliberately ignore transactions containing the latest data feed records from oracles to manipulate the outcomes of DDTs. Lastly, even when the leader is not faulty, cBFT protocols typically trade more communication rounds for reduced message complexity to increase throughput. This trade-off results in higher data staleness when DDTs are committed.

To address the aforementioned issues, we seek an alternative data feed processing scheme for cBFT protocols that satisfies the following properties:

P1. Low staleness: The processing scheme should facilitate timely updates and access to data records, reducing the latency between the generation of a data feed by the oracle and its utilization by DDTs.
P2. On-demand replication: The processing scheme should avoid replicating data unnecessarily when no pending DDTs are requesting it, thereby reducing resource consumption and network overhead. Ideally, data records are not replicated unless explicitly requested by DDTs. Full replication across the blockchain should occur only when it is immediately required by DDTs to ensure efficient utilization of resources.
P3. Censorship-resilience: The processing scheme must be robust against censorship attacks by a malicious leader attempting to suppress or delay the inclusion of recent data records provided by oracles.

We made the following observations: 1) Update transactions issued by oracles are primarily blind writes, meaning their execution does not depend on the current state of the state machine. Additionally, data records from an authenticated oracle feed are self-verifiable by checking the oracle's signature. Therefore, using BFT consensus to process data feed updates is unnecessary and inefficient. 2) In cBFT protocols, a certified block

becomes committed (and thus applicable to the state machine) only after it is chained with at least one subsequent certified block. To initiate a round for certifying a block, the leader in cBFT protocols requires input messages from a quorum of $2f + 1$ replicas. These messages may include recently issued updates from oracles required by DDTs in a previously certified but uncommitted block, facilitating on-demand replication.

Based on these observations, we propose Dolphin, a hybrid system that decouples data feed processing from cBFT consensus. Data feeds issued by oracles are **partially** replicated to a random subset of replicas without passing through BFT consensus. This design allows recent updates to be quickly retrieved by clients through quorum reads. DDTs, on the other hand, are processed via the cBFT protocol. When DDTs are proposed, the leader initiates a Verifiably Recent Broadcast (VRB) in parallel with cBFT to reliably disseminate the data records required by DDTs. Since VRB is triggered only when DDTs are proposed, it supports on-demand replication (P2). Moreover, VRB provides censorship resilience (P3), ensuring each data record delivered to a replica is at most as stale as those reported by $f+1$ correct replicas at the time VRB is invoked by the leader. VRB leverages PBFT [34], which achieves optimal communication rounds [4], thereby minimizing staleness (P1). Although VRB incurs quadratic message complexity to commit data records, Dolphin achieves higher transaction throughput by leveraging the on-demand replication of data feeds.

6.1 Background

This section provides additional background for the chapter, covering chained BFT consensus and blockchain oracles.

6.1.1 Chained BFT Consensus

HotStuff [161, 111] is the pioneering cBFT protocol that introduced a modular and pipelined architecture for consensus. Unlike traditional BFT protocols, such as PBFT [34], which incur quadratic message complexity, HotStuff reduces protocol complexity by chaining blocks through linear consensus phases. This design achieves $O(n)$ message and authenticator complexity, while preserving responsiveness and enabling efficient view changes. Owing to these advantages, HotStuff has been adopted in practical blockchain systems [16, 50] and continues to attract significant attention in academic research [89, 111, 76].

In HotStuff-based cBFT protocols, a chain of blocks—each representing a batch of commands—is constructed through a sequence of voting rounds. In each round, the leader

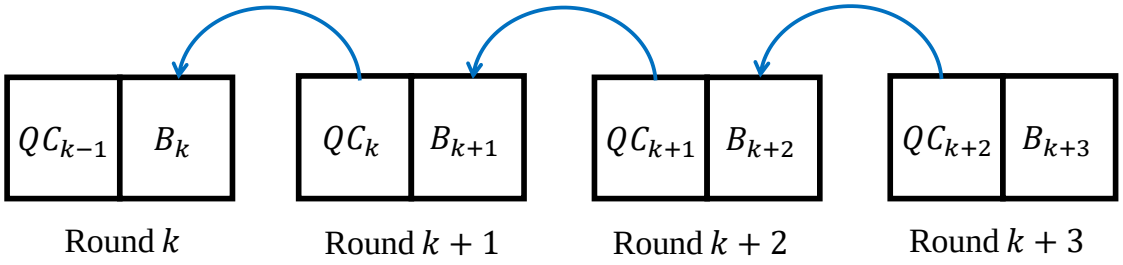


Figure 6.1: HotStuff: block chaining via quorum certificates

collects votes from replicas and forms a quorum certificate (QC) that certifies a previously proposed block. The QC serves as cryptographic evidence that at least $2f+1$ replicas, in a network of n nodes, have endorsed the block. Once a QC is formed, the leader proposes a new block and broadcasts it along with the QC. This proposal cryptographically links the new block to the previously certified block. For example, in Figure 6.1, the leader generates QC_k, certifying block B_k, and embeds it in the newly proposed block B_{k+1} during round $k+1$. Since QC_k includes the hash of B_k, this ensures that B_{k+1} is securely chained to B_k. A block is committed by a replica after it becomes part of a chain that includes one or more subsequent certified blocks.

Figure 6.1 illustrates this commitment rule in HotStuff: block B_k is committed upon receiving QC_{k+2} and block B_{k+3}, which contains QC_{k+2} certifying B_{k+2}. Specifically, QC_k guarantees the uniqueness of B_k by ensuring that no conflicting block exists at round k, where a conflict refers to two blocks extending the same parent but containing different content. Subsequently, QC_{k+1} confirms that $2f+1$ replicas have received both B_k and QC_k. This certificate also locks replicas onto the branch extending from B_k, as replicas accept only blocks that extend B_k, unless a majority of correct replicas ($f+1$) fail to receive QC_{k+1}. Finally, QC_{k+2} ensures that $2f+1$ replicas are locked on the branch extending from B_k, thereby rendering B_k and its ancestors irreversible. Moreover, QC_{k+2} guarantees that QC_{k+1} is stored at at least $f+1$ correct replicas, making QC_{k+1} retrievable using responses from any $2f+1$ distinct replicas.

Dolphin builds upon HotStuff-2 [111]. In HotStuff-2, a block is committed by a replica upon receiving QC_{k+1}, reducing one round-trip delay compared to vanilla HotStuff. As a trade-off, in scenarios where inconsistent votes may be received within a round, HotStuff-2 introduces an additional delay of 2Δ for the leader. This delay ensures the leader can collect votes from all honest replicas before proposing a block, thereby enabling the retrieval of QC_{k+1}.

6.1.2 Blockchain Oracles

Blockchain oracles supply various types of data, such as asset prices, weather conditions, event outcomes, and more. These oracles enable blockchains to perform complex, data-driven operations while preserving decentralization. An authenticated data feed (ADF) record is represented as a tuple $\langle k, val, ts \rangle$, where k denotes a unique identifier for the data feed (e.g., news, game scores, or sensor data), and val represents its corresponding value. Each record includes a timestamp ts, which may be a local clock value or a sequence number, with larger ts values signifying more recent updates. To ensure authenticity, each data feed is cryptographically signed using the oracle's private key. Current applications of blockchain oracles encompass decentralized stablecoins [150, 109], decentralized lending platforms [2], and cross-chain asset swaps [29, 35]. Additionally, numerous other potential applications utilizing data feeds continue to emerge [126].

6.2 Motivation

Algorithm 6 demonstrates an example of an asset dealer smart contract deployed on a blockchain BC_A. The smart contract is associated with two types of DDTs that are common in financial asset trading [141, 101]:

Algorithm 6: Asset Dealer Contract

```
1  Var p_asset // The price in coin A
2  Var p_anchor // The price in an anchor asset
3  Var θ // The thrshold
4  Procedure UpdatePrice():
5  |   E ← getLatest(k)
6  |   p_asset ← p_anchor × E
7
8  Procedure Transfer(ID_transferee):
9  |   E ← getLatest(k)
   |   // Verify if the price exceeds the threshold
10 |   if E > θ then
11 |   |   TransferOwnership(ID_transferee)
12 |   else
13 |   |   Abort() // Abort transaction
```

Real Time Price Adjustment: The dealer aims to sell the asset for coin A, which is the cryptocurrency backed by BC_A. The dealer uses a dynamic pricing strategy that

sets the price of the asset relative to an anchor asset, such as gold, the US dollar, or a stablecoin. To achieve this goal, the dealer periodically issues transactions to invoke the `UpdatePrice`() procedure (line 4) to update the price of the asset based on E, which is the exchange rate between coin A and the anchor asset. Since BC_A does not natively maintain E, the value of the exchange rate is made available on-chain by a price feed oracle, which periodically retrieves the latest rate from trusted off-chain sources, such as centralized exchanges, and stores that value on BC_A. To read the latest value of E stored on BC_A, a transaction calls `getLatest(k)` (line 5), where `k` is the unique ID associated with E.

Conditional Order: Potential asset buyers can issue transcations to invoke line 8, which transfers ownership of the asset from the dealer to the buyer identified by $\texttt{ID}_{\texttt{transferee}}$. The dealer does not wish to sell the asset if E is lower than a threshold θ. Therefore, a condition check $E > \theta$ is required before the actual ownership transfer procedure, represented by $\texttt{TransferOwnership}(\texttt{ID}_{\texttt{transferee}})$, is executed. If the check fails, the transaction will be aborted.

6.2.1 Issues with Conventional Approach

To make the latest data records from data feeds accessible to DDTs on-chain, the conventional approach involves oracles deploying a proxy smart contract. This contract is linked to a key-value store D, which maps data feed IDs to their most recent data records (values). The key-value store D is part of the state machine `M`, which is synchronized across the blockchain network using the underlying BFT consensus protocol. When a new value v corresponding to a data feed identified by k (e.g., a price change or the occurrence of an off-chain event) becomes available, the oracle operator invokes an update transaction $tx : \{D[k] = v\}$ via the proxy smart contract. This transaction tx is then processed by the underlying BFT consensus protocol. However, this approach may result in the issues discussed in subsequent sections.

6.2.2 Data Staleness

For DDTs in asset trading applications, the freshness of data records is critical. In our motivational example illustrated by algorithm 6, if E is lower than the real-time conversion rate when line 4 is executed on-chain, the asset price in coin A could be set to an amount less than the equivalent value of p_{anchor} in the anchor asset. Moreover, if the real-time conversion rate is lower than θ when line 4 is executed on-chain, the ownership can be transferred against the dealer's intentions. Both scenarios could result in potential losses

for the dealer. We define the staleness of a data record associated with a data feed as follows:

Definition 16 (Data Staleness)**.** Let v be a data record issued by the oracle at time t_v. Let e be an event that occurs at time $t_e \geq t_v$. The staleness of v with respect to the event e, denoted as $\epsilon(v, e)$, is defined as:

$$\epsilon(v, e) = t_v - t_e$$

We are particularly interested in the *applied time staleness* of data records read by DDTs:

Definition 17 (Applied Time Staleness)**.** For some DDT tx issued to the blockchain that reads a data record v during its execution, the applied time staleness of v, denoted by $\epsilon_A(v)$, is equal to $\epsilon(v, e_A)$, where e_A denotes the event at which tx is executed and applied to the state machine by the first $f + 1$ replicas.

Intuitively, applied time staleness measures the staleness of a data record read by a DDT, with the execution results guaranteed to survive f failures. In the later sections of this paper, the term "staleness" specifically refers to applied time staleness. With the conventional approach described in section 6.2.1, both data feed updates and DDTs are processed through the cBFT protocol. This approach can result in significant staleness of data records accessed by DDTs. More concretely, consider an update transaction tx_v that encapsulates a data record v issued by the oracle. When v is read by a DDT, the staleness of v must satisfy the following inequality:

$$\epsilon_A(v) \geq \Delta_{propose} + \Delta_{consensus} \tag{6.1}$$

Here, $\Delta_{propose}$ represents the delay between the time tx_v is issued by the oracle and when it is proposed by the leader, while $\Delta_{consensus}$ denotes the delay introduced by the consensus protocol to order tx_v after it has been proposed. In the following sections, we provide a more detailed analysis of how these delays contribute to high data staleness.

High Communication Overhead

For cBFT protocols, $\Delta_{consensus}$ is high due to the extensive communication rounds required by the consensus process. For example, before a replica can apply a block of transactions to the state machine, HotStuff variants [161, 111] generally require 3 to 4 phases in the

absence of leader failures. Each phase consists of two rounds of communication (equivalent to one round-trip) between the leader and the replicas. Consequently, a DDT submitted by a client is decided and applied to the state machine only after undergoing 6 to 8 rounds of communication. As a result, for any DDT executed with a data record v, the staleness of v is at least $6\Delta_C$, where Δ_C is the time required for one round of communication between the leader and replicas. This delay can be over 100ms, in a geo-distributed environment, which is the typical deployment scenario for blockchain systems.

Censorship Attack

In leader-based protocols, during a leader's term, the leader has complete control over which transactions to propose. Consequently, a faulty leader may choose to ignore the most recent data records received from oracles, a situation commonly referred to as a *censorship attack*. Through a censorship attack, a faulty leader can deliberately increase $\Delta_{propose}$, thereby worsening the staleness of v. However, unlike a *liveness attack*, where a leader deliberately prevents progress by withholding messages, a censorship attack on oracle data feeds is much harder for replicas to detect, allowing a faulty leader to potentially remain in reign.

To counter censorship attacks, cBFT protocols can employ a rotating leader scheme to promote decision democracy [161]. With this scheme, on average, it takes $n/(n-f)$ terms for a correct leader to propose a block that includes tx_v. As a result, on average, $\Delta_{propose} \geq \frac{n}{n-f}\Delta_L$, where Δ_L is the duration of a leader term. Other works [48, 145, 66] propose using a mempool layer to ensure a certain portion of transactions are proposed by correct replicas. However, in these approaches, transactions are initially replicated in the mempool layer using reliable broadcast protocols to generate an identifier, which is subsequently ordered by the BFT consensus. This replication process involves at least two additional rounds of communication, resulting in $\Delta_{propose} \geq 2\Delta_C$ even when the leader is not faulty.

6.2.3 Unnecessary Replication

BFT consensus protocols typically require replicating transactions across all replicas. As the number of oracles providing diverse types of information increases, fully replicating every data record for all data feeds can significantly consume on-chain network resources. This may reduce throughput for processing other on-chain transactions beyond data feed updates. Furthermore, as noted in [104], not all data records from a data feed are frequently

accessed by DDTs before being overwritten by newer records. This leads to wasted network resources on data records that remain unused by DDTs.

To address this issue, [104] proposes an adaptive replication approach that replicates data records on-chain only when historical usage patterns indicate heavy access to a specific data feed. This method assumes that data feeds are updated infrequently, ensuring that any delay in replicating a data record has negligible impact on DDTs relying on those feeds. However, this assumption does not hold for DDTs sensitive to data staleness. In such cases, delayed replication (e.g., over a minute as reported in [104]) significantly increases $\Delta_{propose}$ and data staleness, negatively impacting the performance of dependent DDTs.

6.3 Model

The system comprises a set R of $n = 3f + 1$ replicas, among which up to f may exhibit Byzantine faults. Faulty replicas may arbitrarily deviate from the protocol, including delaying or dropping messages and colluding with others. However, they cannot forge the identity or messages of correct replicas. Correct replicas, in contrast, strictly follow the designated protocol.

Each replica $r \in R$ holds a public/private key pair, denoted $(r.pk, r.sk)$, where public keys are globally known and private keys remain confidential. The system operates under a static adversary model, in which the set of faulty replicas is fixed prior to protocol execution. This adversary cannot adaptively compromise replicas during execution.

Clients and oracles operate externally and interact with the system by issuing signed requests to replicas. All messages exchanged within the system are authenticated and signed. Correct replicas accept only messages with valid signatures, allowing them to verify the authenticity and integrity of oracle data. Oracles are trusted by clients to provide accurate and timely data. However, oracles themselves are not assumed to be Byzantine fault-tolerant, and may behave maliciously—e.g., by issuing inconsistent updates, misreporting timestamps, or delaying data publication.

Network communication is point-to-point and reliable: a correct replica receives a message from another correct replica if and only if the sender intends to send it. Communication links among correct replicas are partially synchronous, with a known finite delay Δ after the unknown *Global Stabilization Time* (GST).

6.4 Dolphin Overview

Processing oracle data feeds, such as price feeds, requires only read and write access to a key-value (KV) store. Chapter 4 highlights the limitations of consensus protocols in handling KV workloads. Although WABD, introduced in Chapter 5, supports high-performance read/write operations, it still requires full data replication and incurs high read latency to guarantee *linearizability*. Dolphin improves performance by employing a *probabilistic quorum* technique that eliminates the need for full replication of oracle data feeds. As a trade-off, Dolphin does not enforce *linearizability*, and a read operation may fail to return the most recently issued data record from the oracle.

Dolphin decouples oracle data feed handling from the chained BFT (cBFT) consensus by introducing a lightweight, probabilistic quorum-based read/write protocol. This design reduces both communication and processing overhead associated with write operations, thereby improving system performance. Meanwhile, transactions containing DDTs are processed using the cBFT protocol. However, the data records accessed by DDTs are broadcast only after the DDTs are proposed, enabling *on-demand replication*.

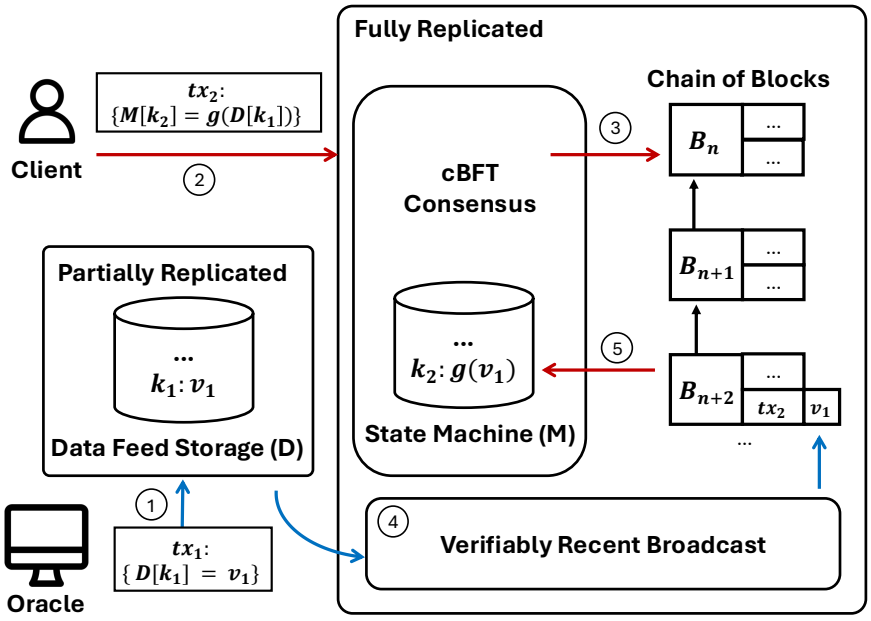


Figure 6.2: The decoupled processing of data feeds and DDTs in Dolphin

As shown in Figure 6.2, the data feed storage D is separated from the state machine

M on each replica. Data records issued by oracles are **partially** replicated to a random subset of replicas without going through BFT consensus (Figure 6.2:②). Recent updates can be retrieved by clients through quorum reads. Dolphin does not enforce linearizable ordering for read and write operations on D. Consequently, write quorums can be much smaller than the ordinary quorum size of $2f+1$, promoting timely updates, while the read quorum size can be adjusted flexibly according to clients' needs. DDTs are still processed through the cBFT protocol (Figure 6.2:③). After the DDT tx_2 is proposed, the leader initiates a Verifiably Recent Broadcast (VRB) in parallel with the cBFT (Figure 6.2:④) to reliably broadcast the data record requested by tx_2. Since VRB is initiated only when a DDT is proposed, it achieves on-demand replication (P2).

6.5 Data Feed Processing

Dolphin decouples oracle data feed handling from cBFT consensus by employing a probabilistic quorum-based read/write protocol. As shown in Algorithm 7, the oracle calls `NewValue`(k)—a blocking procedure that returns the latest data record for feed k once available. The oracle then sends an update request to q_w replicas sampled from R.

Algorithm 7: Oracle Push

▷ oracle main processing loop
1 **for** $val \leftarrow$ `NewValue`(k) **do**
2 $ts \leftarrow$ `NewTimestamp`()
3 $\sigma \leftarrow$ `Sign`($\langle k, val, ts \rangle$)
4 $Q_w \leftarrow$ `Sample`(R, q_w)
5 **send** $\langle$UPDATE$, k, val, ts, \sigma\rangle$ **to all** $r \in Q_w$

Algorithm 8: Data Feed Processing

1 D: data feed storage
2 **Upon receiving** $m = \langle$UPDATE$, k, val, ts, \sigma\rangle$ **do**
3 **if** ¬`Verify`($\langle k, val, ts \rangle, \sigma$) **then**
4 **return**
▷ $D[k]$ equals a default value $\perp$ if k is not present in D.
5 **if** $D[m.k] = \perp \vee m.ts > D[m.k].ts$ **then**
6 $D[m.k] \leftarrow m$
7 **return**
8 **Upon receiving** $\langle$QUERY$, k\rangle$ *from* s **do**
9 **send** $D[k]$ to s

Each replica verifies the record's signature and stores it only if it carries a newer timestamp (Algorithm 8). To retrieve a recent data record for a feed k, a client invokes Algorithm 9, broadcasting a query to all replicas in a chosen quorum Q_r, and waiting for responses from q_r replicas. The client returns the record with the most recent timestamp from $\mathcal{M}$.

Algorithm 9: Client Read

```
1 Procedure OracleRead(k):
2     send ⟨QUERY, k⟩ to all r ∈ Q_r
3     upon receiving a set M of replies from q_r distinct replicas do
4         m ← arg max_{m∈M} m.ts
5     return m.val
```

6.5.1 Simulation of Data Staleness

This section demonstrates the relationship between data staleness returned by Algorithm 9 and configurations of q_r and q_w through simulation. The simulation proceeds as follows:

1. **Data feed generation:** Data record issuance times follow a Poisson process with rate λ_d. Each generated data timestamp is multicast to q_w randomly selected nodes from a total of n nodes. Multicast is simulated by appending the data timestamp to each recipient node's list of received data timestamps.

2. **Client query issuance:** Client queries are generated independently according to a Poisson process with rate λ_q. For each query time t, the client selects q_r nodes uniformly at random. Each selected node returns the interval between t and the most recent data timestamp it received. If the node has not received any data, or is among the f simulated faulty nodes, the interval is set to ∞. The client records the minimum finite interval among the q_r queried nodes.

3. **CDF computation:** Intervals are filtered to exclude infinite values, converted to milliseconds, and offset by 100 ms to account for network latency. The 100 ms offset reflects a 50 ms delay from the oracle to replicas and a 50 ms delay from the client to replicas. The resulting intervals are sorted, and their cumulative distribution function (CDF) is computed. The 95th percentile interval is then recorded for each q_w.

Figure 6.3 presents the CDFs of data staleness obtained by simulating Algorithm 9 1000 times in a network of 64 replicas, with 21 replicas fixed as faulty throughout the

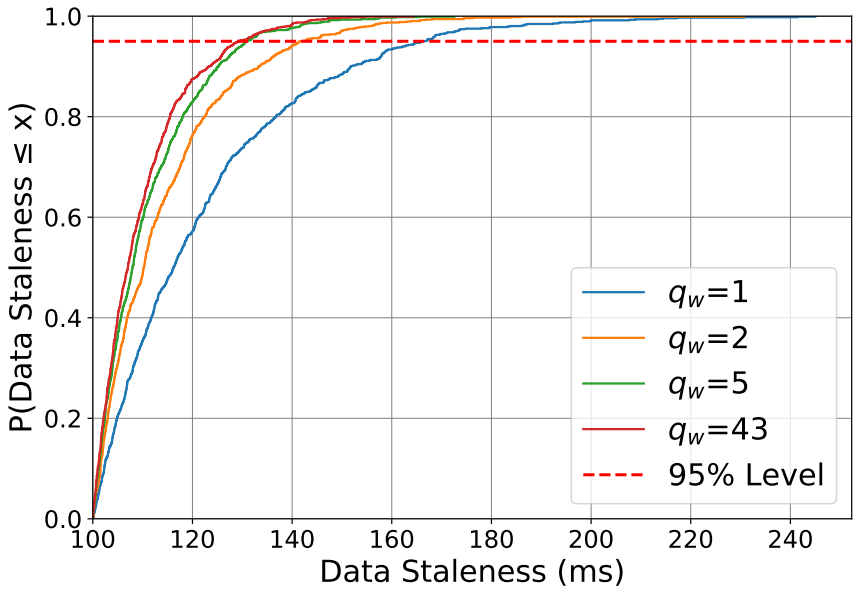


Figure 6.3: CDFs of data staleness from 1000 runs of Algorithm 9 with $n = 64$, $q_r = 43$, varying q_w; $\lambda_d = 100$/sec, $\lambda_q = 1$/sec.

simulation. The read quorum size q_r is fixed to 43, and the CDF is plotted for varying write quorum sizes q_w. The oracle data generation rate is set to $\lambda_d = 100$ updates/sec, while the query arrival rate is $\lambda_q = 1$ query/sec.

As shown in Figure 6.3, reducing q_w only slightly increases data staleness at the 95th percentile. For example, reducing q_w from 43 to 5 increases the 95th percentile staleness from 129 ms to 131 ms. Note that when $q_w = 5$, certain data records may be multicast exclusively to malicious replicas, which may subsequently censor them and prevent their retrieval by clients. Despite this, the result indicates that smaller write quorums cause only marginal degradation in data freshness, even in the presence of Byzantine replicas.

6.5.2 Discussion

Dolphin does not enforce linearizability for oracle data feeds, eliminating the quorum requirement $q_w + q_r > 4f + 1$. By setting $q_w \ll 2f + 1$, frequent updates can be supported without incurring significant overhead or interfering with the underlying cBFT protocol. As a trade-off, Algorithm 9 may fail to return the most recently issued records. However,

as shown in Section 6.5.1, smaller write quorums—even those smaller than the number of Byzantine replicas—result in only marginal degradation in data freshness. Such trade-off is acceptable in many DeFi applications, where oracle data is immutable, signed, and verifiable. For instance, in lending platforms [2] and portfolio trackers [49], users retrieve prices, interest rates, and collateral ratios via oracle data feeds. While the data may be slightly stale, it remains sufficient for position monitoring and informed decision-making.

Dolphin assumes that clients select oracles they trust to provide accurate and timely data. However, oracles are not Byzantine fault-tolerant and may behave maliciously by issuing inconsistent updates, misreporting timestamps, or delaying publication. In such cases, clients may read stale or inaccurate data, and the outcomes of DDTs may diverge from their expectations, potentially leading them to revoke trust in the oracle. Nevertheless, as illustrated in later sections, this behavior does not compromise the safety or consistency of the replicated state machine maintained by Dolphin. The system guarantees that all correct replicas apply an identical sequence of transactions in the same order, even if a malicious oracle reports divergent data records for the same timestamp.

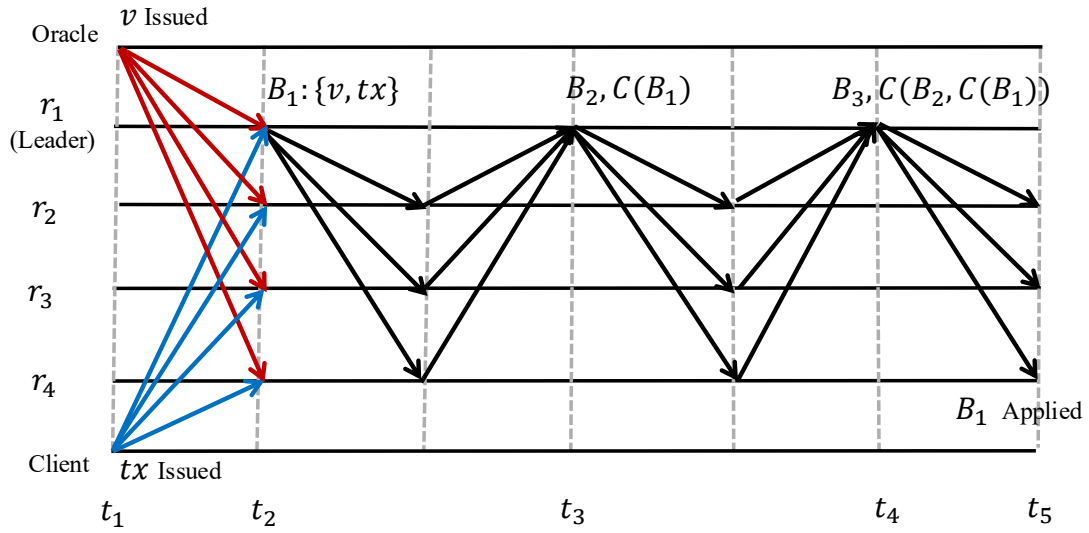


Figure 6.4: 2-Phase cBFT: DDT tx and data record v are committed together in block B_1

6.6 DDT Processing

In Dolphin, transactions, including DDTs, are still processed using the cBFT protocol. However, the data records required by DDTs are broadcast only after the DDTs are proposed, enabling *on-demand replication*. To broadcast these data records, Dolphin employs *verifiably recent broadcast* (VRB), a censorship-resistant BFT protocol with optimal communication rounds. Figure 6.4 and Figure 6.5 illustrate the differing processing of DDTs in a two-phase cBFT protocol compared to Dolphin.

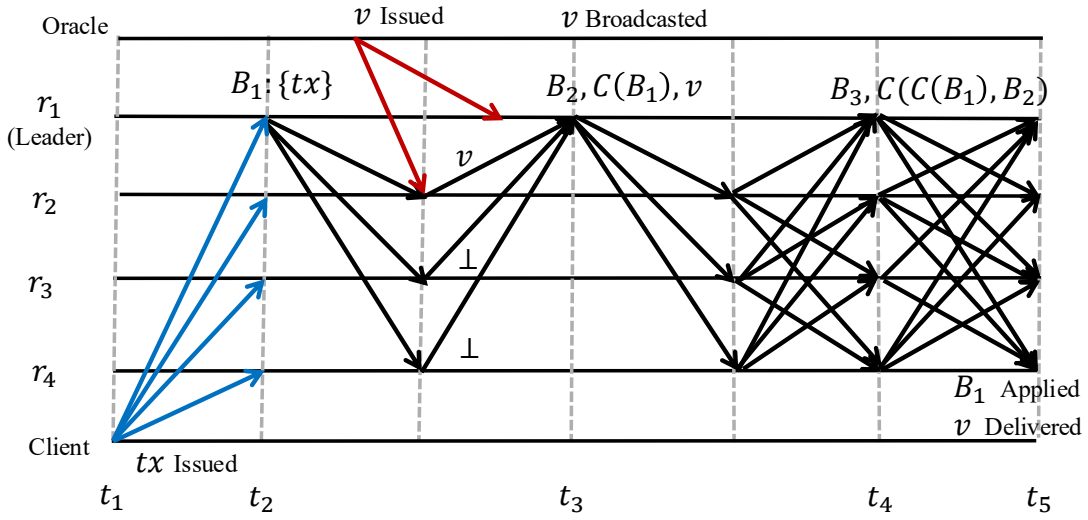


Figure 6.5: Dolphin: data record v is broadcast only when required by DDT tx

As shown in Figure 6.4, without decoupling data feeds and transaction processing, the client's DDT tx and oracle's data record v are broadcast for proposal. The leader includes tx and v in block B_1, broadcasts it, and gathers $n - f$ votes to form a prepare QC $C(B_1)$ at t_3. $C(B_1)$ and a new block B_2, extending B_1, are then broadcast. A QC for B_2 and $C(B_1)$, denoted $C(C(B_1), B_2)$, serves as the commit QC for B_1. Replicas apply B_1 once all predecessor blocks are applied. If v is excluded from B_1 due to a faulty leader or delay, tx uses a stale record from a prior block. Figure 6.5 illustrates DDT processing in Dolphin. The DDT tx is included in block B_1 and committed after two voting rounds. Oracle-issued data records are partially replicated before being needed by proposed DDTs. For instance, v is replicated only to r_1 and r_2. When replicas receive B_1, they verify if tx requires oracle data. r_2 reports v, while r_3 and r_4, lacking oracle data, report $\perp$. At t_3, upon receiving responses from $2f + 1$ replicas, the leader selects the latest data record v

based on timestamps and broadcasts it via VRB. Replicas apply tx with v after receiving $C(C(B_1)), B_2$, and v via VRB. By initiating VRB post-proposal, Dolphin achieves on-demand replication. Due to partial replication, it is possible that no correct replica receives value v, enabling malicious replicas to censor v from data-dependent transactions (DDTs). Nevertheless, as shown in Section 6.5.1, such adversarial conditions cannot significantly increase data staleness.

Local state on replica r:

1. M: a mapping from states to their corresponding values in the state machine.
2. D: data feed storage, mapping data feed keys to their corresponding records. D is initialized as an empty map, and $D[k]$ yields the default value $\perp$ whenever k is not present in D.
3. $Blocks$: block storage, mapping block hashes to their respective blocks. Initialized as an empty map.
4. qc_{high}: the quorum certificate with the highest view number known to replica r.
5. $view_C$: the view number in which replica r is currently participating for cBFT, determining the leader responsible for the present consensus round. Initialized as 0.
6. $view_V$: the view number used by replica r for VRB, indicating the leader epoch governing broadcast and delivery. Initialized as 0.
7. I: a mapping from each block hash to its corresponding VRB instance, which maintains the metadata required for VRB coordination. Initialized as an empty map.

Figure 6.6: Local state maintained by each replica

6.6.1 Data Structures and Utilities

Algorithm 10 presents the key data structures used in Dolphin. The cBFT protocol organizes transactions into a chain of blocks, where each block B contains transactions $txns$, a view number $view$ indicating the leader's term, and a quorum certificate qc referencing

its predecessor B_p. A block B extends B' if there is a succession path from B' to B. In Dolphin, each block also includes a *ddts* field representing a set of DDTs.

Algorithm 10: Data Structure and Utility

```
   ▷ view: view number, qc_p: quorum certificate of the predecessor block
1  Procedure Block(view, qc_p):
2      B.view ← view
3      B.txns ← GetTxns()
4      B.ddts ← GetDDTs()
5      B.qc ← qc_p
6      return B
   ▷ B: a block to be voted for
7  Procedure Vote(B):
8      vote.view ← B.view
9      vote.h ← H(B)
10     K ← DFKeys(B.ddts) ▷ DFKeys(B.ddts) returns the data feed keys required by
         ddts
11     vote.V ← {k ↦ D[k] | k ∈ K}
12     tsmap ← {k ↦ vote.V[k].ts if vote.V[k] ≠ ⊥ else ⊥ | k ∈ K}
13     vote.σ_B ← Sign(vote.h)
14     vote.σ_D ← Sign(⟨B.view, H(tsmap)⟩)
15     return vote
   ▷ M: a set of matching votes with identical view, h, and σ_B fields
16 Procedure QC(M):
17     select any m ∈ M:
18         qc.view ← m.view
19         qc.h ← m.h
20     qc.Q ← {m.σ_B | m ∈ M}
21     return qc
   ▷ qc: a quorum certificate, vote: a vote object
22 Procedure NewViewMsg(qc, vote):
23     m.qc ← qc
24     m.vote ← vote
25     return m
```

Line 7 describes the procedure for generating a vote *vote* for block B. The vote includes $vote.view$, which matches $B.view$, the hash $vote.h$, which equals $H(B)$, and the signature $vote.\sigma_B$. If $B.ddts$ is not empty, $vote.V$ contains the relevant data records, retrieved from the replica's local storage D. The local variable *tsmap* maps data feed keys to their corresponding timestamps in $vote.V$. The signature $vote.\sigma_D$ covers $B.view$ and $H(tsmap)$,

thereby preventing reuse of *tsmap* across views to propose stale values. The signature $vote.\sigma_D$ is essential for ensuring censorship resilience in VRB, as discussed in detail in Section 6.6.3.

Algorithm 11: Block Processing in Dolphin

$qc_{high} \leftarrow \{view : 0, h : \perp, Q : \emptyset\}$
▷ `Replica enters cBFT view` i`:`
▷ `Only Leader(`i`):`
Upon receiving $\mathcal{M}$*:* a set of $n - f$ new-view messages from cBFT view $i - 1$ **do**
  if votes in $\mathcal{M}$ are matching **then**
    $qc_{high} \leftarrow \texttt{QC}(\{m.\text{vote} \mid m \in \mathcal{M}\})$
  else
    ▷ `ensures receipt of new-view messages from all correct replicas`
    wait for 3Δ after entering view i
    $\mathcal{M}_a \leftarrow$ set of all received new-view messages
    $qc_{high} \leftarrow \left(\arg\max_{m \in \mathcal{M}_a} m.qc.view\right).qc$
  $B_i \leftarrow \texttt{Block}(i, qc_{high})$
  broadcast B_i
▷ `All replicas:`
Upon receiving B_i *from* `Leader(`i`)` **do**
  if $B_i.qc.view \geq qc_{high}.view$ **then**
    $qc_{high} \leftarrow B_i.qc$
    `CommitRule(`$B_i.qc$`)`
    $vote \leftarrow \texttt{Vote}(B_i)$
    send $\texttt{NewViewMsg}(qc_{high}, vote)$ to $\texttt{Leader}(i + 1)$
    if $B_i.ddts \neq \perp$ **then**
      send $vote$ to $\texttt{VRBLeader}(view_V)$
    $view_C \leftarrow i + 1$ ▷ `Stop voting for cBFT view` i
Upon cBFT view v **timeout do**
  send $\texttt{NewViewMsg}(qc_{high}, \perp)$ to $\texttt{Leader}(i + 1)$
  $view_C \leftarrow i + 1$
Procedure `CommitRule(`qc`):`
  $B \leftarrow Blocks[qc.h]$
  $CB \leftarrow$ all uncommitted predecessor blocks of B
  foreach $B' \in CB$ **do**
    deliver event COMMIT(B')

6.6.2 Block Processing

Dolphin adopts HotStuff-2 as the underlying cBFT protocol. Replicas in HotStuff-based protocols operate through a sequence of monotonically increasing views. In each view i, a replica designated as `Leader`(i) coordinates block proposals. The leader is chosen according to a pre-configured round-robin schedule. Figure 6.6 illustrates the local state maintained by each replica, and Algorithm 11 presents the block processing procedure in Dolphin, which operates as follows:

1. **Block Proposal (line 2):** The leader of view i, `Leader`(i), enters the view and proceeds after receiving $n - f$ new-view messages (see message structure in line 22, Algorithm 10). Two scenarios are possible:

   (a) **Good Case (line 3):** The leader obtains $n - f$ matching votes for a block proposed in view $i - 1$. Votes are deemed matching if they contain identical view numbers and block hashes. Based on these votes, the leader constructs a quorum certificate (QC), creates a new block B_i containing transactions, data feeds V, and their associated proofs Π, and broadcasts the block to all replicas, including itself.

   (b) **Fallback Case (line 5):** If the leader fails to collect $n - f$ matching votes, possibly due to faults or adversarial behavior in the previous view, the protocol engages a fallback mechanism. As described in line 5, the leader retrieves the highest known quorum certificate from all new-view messages received from the previous view, constructs a new block B_i, and broadcasts it to all replicas.

2. **Block Reception and Validation (line 11):** Upon receiving the proposed block B_i, each replica, including the leader, verifies that $B_i.qc.view \geq qc_{high}.view$. If this condition is not met, the block is ignored (line 12).

3. **Commit Rule Invocation (line 14):** If the quorum certificate in B_i passes validation, the replica invokes the `CommitRule` procedure to commit any blocks certified by $B_i.qc$ (line 14). The complete logic of this procedure is detailed in Algorithm 14 and discussed in Section 6.6.7.

4. **Vote Casting (line 15):** After committing the necessary blocks, the replica constructs a vote for B_i and sends it to `Leader`$(i + 1)$, the designated leader of the next view (line 16). If $B_i.ddts$ is non-empty, the vote is also sent to `VRBLeader`$(view_V)$ (line 18), which is responsible for VRB processing (as covered in Section 6.6.3).

5. **View Timeout (line 20):** Each view v is associated with a timeout Δ_{view} that begins upon entering the view. If a replica does not receive a valid proposal from the leader within Δ_{view}, it transitions to the next view $i+1$. In doing so, the replica sends a new-view message containing its local qc_{high}, but does not cast a vote.

HotStuff-based cBFT protocols incorporate a dedicated module, referred to as the **Pacemaker** [161, 111], which is responsible for synchronizing and advancing views among participating replicas to ensure liveness under partial synchrony. Dolphin integrates the same **Pacemaker** implementation as employed in HotStuff-2 [111], benefiting from its formally established correctness and reliability guarantees. This module satisfies the following key property:

Definition 18 (View Synchronization Property)**.** If a correct replica reaches view i after GST, then all correct replicas are guaranteed to reach the same view i within 2Δ.

Consequently, the leader elected for view i is guaranteed to receive new-view messages from all correct replicas, thereby enabling the successful initiation of the view-change process.

6.6.3 Verifiably Recent Broadcast

A faulty leader may withhold recent oracle data records and instead broadcast stale values to manipulate the outcomes of DDTs, even after receiving the most recent data records. To mitigate this threat, Dolphin introduces *Verifiably Recent Broadcast* (VRB), a censorship-resilient broadcast primitive that ensures both consistency and data freshness.

According to Algorithm 8, for each data feed, a correct replica $r \in R$ updates its local storage D with data records received from the oracle in monotonically increasing order of timestamps. A data record input refers to either the data record associated with an update to D on a correct replica, or the default record $\perp$ if no update has occurred. The VRB primitive is defined as follows:

Definition 19 (Verifiably Recent Broadcast)**.** Let d_r^i denote a data record input of a replica r after it enters cBFT view i. A leader refers to the replica that initiates the broadcast during cBFT view i. A verifiably recent broadcast satisfies the following properties:

1. **Validity:** If a correct leader broadcasts d_r^i, then the leader eventually delivers d_r^i.
2. **Agreement:** If any correct replica delivers d_r^i, then every correct replica eventually delivers d_r^i.

3. **Recency:** If a correct replica delivers d_r^i, then there exists a subset $R' \subseteq R$ of correct replicas, with $|R'| \geq f+1$, satisfying one of the following conditions: (1) There exists a set of timestamps

$$\mathcal{I} = \{d_{r'}^i.ts \mid r' \in R',\ d_{r'}^i \neq \perp\}$$

such that $d_r^i.ts \geq \max(\mathcal{I})$; or (2) $d_r^i = \perp$ and for all $r' \in R'$, there exists a data record input $d_{r'}^i$ such that $d_{r'}^i = \perp$.

Algorithm 12: VRB Message Processing

▷ Only VRBLeader($view_V$):

Upon receiving $\mathcal{M}$: a set of $n-f$ matching vote messages **do**
  $V, \Pi \leftarrow$ MergeDataFeeds($\mathcal{M}$)
  select any $vote \in \mathcal{M}$:
    $i \leftarrow vote.view$ // view of the block
    $h_B \leftarrow vote.h$ // hash of the block
  broadcast $\langle$VRB-PRE-PREPARE$, i, h_B, V, \Pi\rangle$

▷ All replicas:

Upon receiving $\langle$VRB-PRE-PREPARE$, i, h_B, V, \Pi\rangle$ from VRBLeader($view_V$) **do**
  if VerifyPoR(i, h_B, V, Π) **then**
    $h_V \leftarrow H(V)$
    $I[h_B] \leftarrow$ VrbInstance($h_B, view_V, V$)
    broadcast $\langle$VRB-PREPARE$, view_V, h_B, h_V\rangle$

Upon receiving $\mathcal{M}$: a set of $n-f$ $\langle$VRB-PREPARE$, view_V, h_B, h_V\rangle$ from distinct senders, where $h_V = H(I[h_B].data)$ **do**
  $I[h_B].pmsgs \leftarrow \mathcal{M}$
  broadcast $\langle$VRB-COMMIT$, view_V, h_B, h_V\rangle$ // stop sending VRB-COMMIT for instance $I[h]$ in $view_V$

Upon receiving $n-f$ $\langle$VRB-COMMIT$, view_V, h_B, h_V\rangle$ from distinct senders, where $h_V = H(I[h_B].data)$ **do**
  $I[h_B].status \leftarrow$ COMMITTED // $I[h_B].data$ is delivered

▷ h: block hash, $view$: VRB view number, V: set of data records

Procedure VrbInstance($h, view, V$):
  $ins.view \leftarrow view$ // VRB view number
  $ins.h \leftarrow h$ // block hash associated with the instance
  $ins.data \leftarrow V$ // set of data records
  $ins.status \leftarrow \perp$ // instance status
  $ins.pmsgs \leftarrow \emptyset$ // set of VRB prepare messages
  return ins

```
   ▷ M:  a set of n − f votes
24 Procedure MergeDataFeeds(M):
   |  ▷ Π:  proof of recency; Keys(V) returns the domain of V
25 |  Π ← {({k ↦ m.V[k].ts | k ∈ Keys(m.V)}, m.σ_D) | m ∈ M}
26 |  V ← {}
27 |  for m ∈ M do
28 |  |  for (k, d) ∈ m.V do
29 |  |  |  if V[k] = ⊥ ∨ (d ≠ ⊥ ∧ d.ts > V[k].ts)) then
30 |  |  |  |  V[k] ← d          // k:  data feed key, d:  corresponding data record
31 |  return V, Π
32 Procedure VerifyPoR(view, h, V, Π):
33 |  if Π does not contain n − f signatures from distinct replicas then
34 |  |  return false
   |  ▷ maxTs[k] returns the default value ⊥ if key k is not present in maxTs
35 |  maxTs ← {}
36 |  for (tsmap, σ) ∈ Π do
37 |  |  if Verify (⟨view, tsmap⟩, σ) then
38 |  |  |  for (k, ts) ∈ tsmap do
39 |  |  |  |  if ts ≠ ⊥ ∧ (maxTs[k] = ⊥ ∨ ts > maxTs[k]) then
40 |  |  |  |  |  maxTs[k] ← ts
41 |  |  else
42 |  |  |  return false
   |  ▷ DFKeys(ddts) returns the data feed keys required by ddts
43 |  for all k ∈ DFKeys(Blocks[h].ddts) do
44 |  |  if V[k] ≠ ⊥ ∧ maxTs[k] = V[k].ts then
45 |  |  |  return true
46 |  |  if V[k] = ⊥ ∧ maxTs[k] = ⊥ then
47 |  |  |  return true
48 |  return false
```

VRB operates in two main phases: the **collect phase** and the **consensus phase**. In the collect phase, the leader gathers data records from replicas and constructs a *proof of recency* (PoR). A PoR is a collection of signed timestamps from $2f + 1$ distinct replicas, where each timestamp corresponds to the data record input of the respective replica. Since timestamps are signed with the cBFT view in which the data record is collected by the leader, the PoR serves as evidence that the broadcasted data record satisfies the recency property. In the consensus phase, the leader proposes the data record along with its PoR, coordinating replicas to agree on its delivery. Progress in this phase requires a valid PoR, ensuring that $2f + 1$ signed timestamps are included. This design prevents faulty

leaders from broadcasting stale records arbitrarily, thereby limiting their influence on DDT outcomes. To align VRB commitments with block commitments in cBFT and to reduce data staleness compared to running cBFT directly on data records, VRB uses a PBFT-based consensus phase. This phase completes in three rounds, achieving optimal good-case communication performance [4], and commits the data records required by DDTs.

6.6.4 Collect Phase

In Dolphin, if a block proposed by the cBFT leader contains any DDT, all replicas automatically initiate the VRB collect phase for each required data feed key. The process unfolds as follows:

1. As specified in Algorithm 11, upon receiving such a block, replicas send their votes to both the next cBFT leader and the designated VRB leader, as shown in line 17.
2. Upon receiving $n - f$ matching vote messages, the VRB leader invokes the `MergeDataFeeds` procedure (line 2) to aggregate the data feed information (pseudocode specified in line 24 of Algorithm 12). The `MergeDataFeeds` procedure constructs two components:
   (a) A map V containing the data record input for each required data feed key.
   (b) A PoR, denoted as Π, consisting of $n-f$ tuples. Each tuple includes a timestamp map ($tsmap$) and a corresponding signature (σ_D) extracted from a vote.
3. The VRB leader then broadcasts a VOTE-PRE-PREPARE message, which serves a role analogous to PBFT's *pre-prepare* message. This message includes the data record set V, the proof-of-recency Π, the cBFT view v, and the block hash h_B. These fields enable replicas to verify the validity of Π.

6.6.5 Consensus Phase

Upon receiving a VOTE-PRE-PREPARE message, each replica executes the following steps as specified in Algorithm 12:

1. **PoR Verification (line 8):**

(a) Verify that the data feed set V satisfies recency requirements by invoking the `VerifyRecency` procedure (line 32).

(b) Since the PoR Π includes *tsmap* values from at least $f + 1$ correct replicas, successful verification confirms that all data records in V meet the required recency condition (Definition 19).

(c) If verification succeeds, create a `VrbInstance` object associated with the block hash h_B (line 10). The `VrbInstance` object *ins* (line 17) enables each replica to track the status of the VRB consensus phase (either committed or not) via $ins.status$, and to store prepare messages needed for view change (as explained in Section 6.6.6) in $ins.pmsgs$, for the VRB instance identified by h_B.

2. **PBFT Prepare Phase (line 11):**

(a) Broadcast a VRB-PREPARE message of the form $\langle$VRB-PREPARE, $view_V, h_B, h_V\rangle$, where h_V is the hash of V (line 11). This message functions analogously to the prepare message in PBFT.

(b) Upon receiving $n - f$ matching VRB-PREPARE messages, store them in the corresponding VRB instance to support recovery from a malicious leader, as described in Section 6.6.6.

3. **PBFT Commit Phase (line 14):**

(a) Broadcast a VRB-COMMIT message with the same fields as the prepare message.

(b) Upon receiving $n - f$ matching VRB-COMMIT messages, update the status of the VRB instance to COMMITTED, and deliver the data feed set V.

6.6.6 VRB View Change

Algorithm 13 illustrates the VRB view change mechanism, which is analogous to PBFT's view change process.

Algorithm 13: VRB View Change

Upon VRB view $view_V$ **timeout do**
  ▷ Stop processing any VRB messages with a VRB view number lower than $view_V + 1$
  for all instances $ins \in I$ such that $ins.status \neq$ COMMITTED **do**
    $h \leftarrow ins.h$
    $V \leftarrow ins.data$
    $\mathcal{P} \leftarrow ins.pmsg$
    send $\langle$VRB-VIEW-CHANGE$, view_V + 1, h, \mathcal{P}, V\rangle$ to `VRBLeader`$(view_V + 1)$

Unlike cBFT views, a VRB view times out only when correct replicas fail to make progress for delivering data records. When a view change is triggered for the VRB view $view_V$, a replica stops processing VRB messages with a view number lower than $view_V + 1$ and sends a VRB-VIEW-CHANGE message. This message includes $\mathcal{P}$, a set of $n - f$ VRB-PREPARE messages for the data record set V, corresponding to the VRB instance associated with block hash h. The message is sent to the new leader determined by `VRBLeader`$(view_V + 1)$. If $\mathcal{P}$ is unavailable, the replica sets both $\mathcal{P}$ and V to empty sets in the message.

The view change process for a VRB instance identified by h proceeds as follows:

1. Upon receiving $n - f$ VRB-VIEW-CHANGE messages for the VRB instance associated with h, the designated leader broadcasts a VRB-NEW-VIEW message of the form:
$$\langle \text{VRB-NEW-VIEW}, view_V + 1, h, V, \mathcal{M}\rangle$$
where:
   (a) $\mathcal{M}$ is the set of $n - f$ VRB-VIEW-CHANGE messages received during the previous view.
   (b) V is the corresponding data record set, selected from a VRB-VIEW-CHANGE message in $\mathcal{M}$ that contains the highest view number and a valid $\mathcal{P}$—a set of $n - f$ VRB-PREPARE messages supporting V.
   (c) If no such V can be extracted from $\mathcal{M}$, meaning messages in $\mathcal{M}$ either contain the empty data record set or do not include a valid $\mathcal{P}$ for V, then V is assigned the empty set $\emptyset$.
2. Upon receiving a valid VRB-NEW-VIEW message from `VRBLeader`$(view_V + 1)$, the replica verifies whether the proposed data set V is consistent with the contents of $\mathcal{M}$, using the same selection method applied by the designated leader. If the verification

succeeds, the replica proceeds to vote for V in view $view_V + 1$, as described in Algorithm 12.

3. After the view change, it is possible that the data record set associated with the DDTs is empty (line 6, Algorithm 14). In such cases, the corresponding DDTs are not executed and are instead aborted.

This section describes replicas as sending separate view-change messages for each uncommitted VRB instance. In practice, replicas aggregate all uncommitted VRB instances into a single batched view-change message. A similar batching strategy is applied to the VRB-NEW-VIEW message constructed and disseminated by the leader.

6.6.7 Transaction Execution

Algorithm 14 outlines the procedure for committing and executing transactions in Dolphin. When line 23 (Algorithm 11) is invoked with a quorum certificate qc, all uncommitted predecessor blocks of the certified block B are committed, which triggers a commit event for B. This commit rule is identical to HotStuff-2 [111].

Algorithm 14: Dolphin Transaction Execution

```
1  Upon event COMMIT(B) do
2      wait until: all predecessor blocks of B are executed
3      for txn ∈ B.txns do
4          Apply(txn, ∅)            // apply regular transaction txn to the state machine
5      wait until: I[H(B)].status = COMMITTED
6      if I[H(B)].data ≠ ∅ then
7          K ← DFKeys(B.ddts)    // DFKeys(ddts) retrieves the data feed keys required
            by ddts
8          V ← {}
9          for k ∈ K do
10             d ← I[H(B)].data[k]
11             if M[k] = ⊥ ∨ (d ≠ ⊥ ∧ M[k].ts < d.ts) then
12                 M[k] ← d
13             V[k] ← M[k]
14         for txn ∈ B.ddts do
15             Apply(txn, V)   // apply DDT txn to the state machine using data record
                set V
16     else
17         Abort(B.ddts)                                   // abort executing DDTs
```

Upon triggering the commit event for block B, the execution of transactions in B proceeds through the following steps:

1. Wait until all predecessor blocks of B have been executed (line 2).
2. Execute each transaction in $B.txns$, as these transactions do not rely on external data feeds and are evaluated using an empty set as the data feed input (line 4).
3. Wait until the VRB instance $I[H(B)]$ reaches the COMMITTED status (line 5).
4. Resolve the data feed values required by the DDTs (line 6 to line 13):
   (a) Extract the required data feed keys K using `DFKeys`$(B.ddts)$.
   (b) Initialize an empty map V to store the resolved data feed values.
   (c) For each $k \in K$, compare the timestamp from the committed VRB instance with $M[k].ts$, if $M[k]$ exists in the world state. If the committed instance contains a newer value, update $M[k]$ accordingly.
   (d) Record the resulting value of $M[k]$ in $V[k]$.
5. Execute each DDT in $B.ddts$ using the resolved data feed map V (line 15). If V is empty, abort execution of all DDTs in $B.ddts$ (line 17).

In Dolphin, transactions in $B.txns$ can be executed immediately. In contrast, executing $B.ddts$ requires that the VRB instance associated with $H(B)$ is also committed. Although this VRB instance is initiated after block B is proposed, the PBFT-inherited optimal communication structure ensures that the waiting time for committing the associated VRB instance remains bounded and efficient.

For DDTs that rely on the same data feed, Dolphin ensures a *monotonic read* consistency level [149]. If two transactions, tx_1 and tx_2, both access the same data feed identified by k, and tx_1 is ordered before tx_2 by the cBFT, then the timestamp of the data record read by tx_1 will always be earlier than or equal to the timestamp of the data record read by tx_2.

6.7 Correctness

Theorem 7 (Dolphin Safety)**.** *In Dolphin, no pair of correct replicas r_1 and r_2 commit or execute different sequences of transactions.*

Proof. Dolphin inherits its core block proposal (line 2, Algorithm 11), block voting (line 11, Algorithm 11), and commitment rule (line 14, Algorithm 11) from HotStuff-2 without modification. According to the safety guarantees of HotStuff-2, no two correct replicas can commit conflicting blocks at the same block height. The block height refers to the position of a block within the total order of committed blocks, with the initial block assigned height zero.

For blocks containing DDTs, each VRB instance in Dolphin is executed as a one-shot PBFT protocol (line 6 to line 16, Algorithm 12) to resolve and commit a specific set of data records V for the block identified by its hash (line 17, Algorithm 12). PBFT guarantees that no two different values can be committed for the same instance. As a result, no two correct replicas can commit different data feed sets for the same block.

Combining these two arguments, Dolphin ensures that (1) no two correct replicas commit the different sequences of blocks, and (2) each DDT in a committed block is executed using a uniquely determined data record set. Thus, all correct replicas execute the same sequence of transactions in the same order, proving safety. □

Theorem 8 (Dolphin Liveness)**.** *Given an unbounded supply of transactions, some transaction will eventually be executed by all correct replicas.*

Proof. Dolphin inherits the block processing logic from HotStuff-2, which guarantees that all correct replicas eventually commit some block B proposed by a correct leader, as long as the network satisfies partial synchrony. If block B contains DDTs (line 17, Algorithm 11), Dolphin initiates a corresponding VRB instance associated with the block hash $H(B)$ (line 10, Algorithm 12). Two cases arise: if the VRB leader responsible for the instance is correct, then all correct replicas will commit and deliver a data record set V for DDT execution (line 16, Algorithm 12). If the VRB leader is faulty, the VRB protocol leverages PBFT's proven view change mechanism (Algorithm 13) to eventually select a correct leader. The new correct leader will take responsibility for any uncommitted VRB instance, ensuring that it is eventually committed. □

Theorem 9 (Dolphin Recency)**.** *Any data record delivered by VRB (Algorithm 12) satisfies the recency property.*

Proof. Let d be a value in a set V delivered by VRB, and let k be its associated data feed key, such that $V[k] = d$.

To deliver d, at least one correct replica must have verified a proof of recency (PoR) Π (line 8) using the `VerifyPoR` procedure (line 32). According to line 33, Π must contain

signed timestamp maps from at least $2f+1$ distinct replicas, including at least $f+1$ correct replicas. Denote this subset of correct replicas as R'.

Let $tsmap_r$ denote the timestamp map from replica $r \in R'$. Each $tsmap_r$ is signed (line 14) and verified (line 37) with respect to a specific cBFT view i. According to line 11 and line 12, each entry $tsmap_r[k]$ corresponds to a data record input of r during cBFT view i.

1. **Case 1:** $d \neq \perp$.

   Let
   $$\mathcal{I} = \{tsmap_r[k] \mid r \in R',\ tsmap_r[k] \neq \perp\}.$$
   From these timestamp maps, the `VerifyPoR` procedure constructs $maxTs$ (line 40), which records the highest timestamp observed for each data feed key, including k, such that
   $$maxTs[k] \geq \max(\mathcal{I}).$$
   Line 44 ensures that
   $$d.ts = V[k].ts = maxTs[k] \geq \max(\mathcal{I}),$$
   thereby satisfying the recency property.

2. **Case 2:** $d = \perp$.

   The check at line 46 ensures that $maxTs[k] = \perp$ and key k is absent from $maxTs$ when verified by a correct replica, implying that line 40 has never been executed for k. As a result, $\forall r \in R',\ tsmap_r[k] = \perp$. According to line 12, this implies $D[k] = \perp$ at the time line 11 is executed by each $r \in R'$ during cBFT view i, thereby satisfying the recency property.

□

Theorem 10 (Monotonic Read Consistency for DDT Execution)**.** Let $tx_1 \in B_1.ddts$ and $tx_2 \in B_2.ddts$ be two DDTs accessing the same data feed key k, where B_1 and B_2 are distinct blocks committed by cBFT. If B_1 precedes B_2 in cBFT order, then the timestamp of the data record read by tx_1 is less than or equal to the timestamp of the data record read by tx_2, i.e.:
$$B_1 \prec B_2 \ \wedge\ V_1[k] \neq \perp \implies V_1[k].ts \leq V_2[k].ts$$
where V_1 and V_2 are the data record sets used during the execution of B_1 and B_2, respectively.

Proof. According to the execution procedure described in Section 6.6.7 and Algorithm 14, the DDT execution mechanism in Dolphin mandates that each block B must wait for the execution of all its predecessor blocks before proceeding (line 2). This requirement ensures that block B_1 is fully executed, including all associated DDTs, prior to the execution of block B_2.

Execution of a DDT requires that the VRB instance associated with the block hash $H(B)$ reaches the COMMITTED status (line 5). At that point, the world state M is updated with the latest timestamped data record from the VRB instance if it is more recent than the current $M[k]$ (line 11). This ensures that $M[k]$ always reflects the most recent value committed by any prior block. The data record set used to execute each DDT copies the current value of $M[k]$ at the time of execution (line 13). Since $B_1 \prec B_2$ in the cBFT order, and all updates to $M[k]$ made by B_1 are visible before B_2 begins execution (line 2), it follows that $V_1[k].ts \leq V_2[k].ts$.

Thus, the timestamp of the data record read by tx_1 will never exceed that read by tx_2, ensuring monotonic read consistency for shared data feed key k. □

6.8 Evaluation

We implement HotStuff-2 and Dolphin using the open-source bamboo framework [65] and evaluate their performance. Our evaluation environment consists of up to 100 instances evenly distributed across four geo-distributed regions on AWS: US-EAST, US-WEST, EU-WEST, and AP-SOUTHEAST. Each instance runs on an `m5.4xlarge` machine with 4 CPU cores and 16GB of memory. The data record storage is modeled as a key-value (KV) store with a keyspace of up to 2K keys. Each instance includes an oracle process that issues data feeds at intervals of 10ms, matching the refresh rate of professional trading tools [142], or 100ms, consistent with the update speed of market data streaming services [24]. Each data record contains a randomly generated payload of 256 bytes and a 64-byte ECDSA signature. To eliminate the impact of network latency caused by leaders in different regions, we use a static leader configuration. The replicas are evenly distributed across all geo-regions. For each experiment, we conduct three runs, with each run lasting 1 minute.

Our evaluation seeks to address the following questions:

1. How do the techniques employed by Dolphin impact the commit latency and throughput of cBFT when processing DDTs? (section 6.8.1, section 6.8.4)
2. How effective is Dolphin at reducing data staleness for DDTs? (section 6.8.2)

3. How resilient is Dolphin to censorship attacks by malicious leaders? (section 6.8.2)

6.8.1 Commit Latency

In this experiment, we evaluate the impact of VRB on the commit latency of blocks containing DDTs. To eliminate variations, we set up the experiment as follows: A single oracle process is deployed in the US-EAST region, issuing data feeds every 10ms. The data feeds are broadcast to all nodes in the replica. For both Dolphin and HotStuff-2, a batch size of 400 transactions per block is used, achieving an average throughput of approximately 4K transactions per second. Each batch contains a DDT that accesses an identical data feed key. In Dolphin, both the VRB leader and the cBFT leader are colocated on the same instance in the US-EAST region. We measure both data staleness and commit latency (average, P90) by having replicas report the commit latency of DDTs and the corresponding staleness of the data feed at the time of application. We conduct experiments in clusters with 64 replicas and 100 replicas.

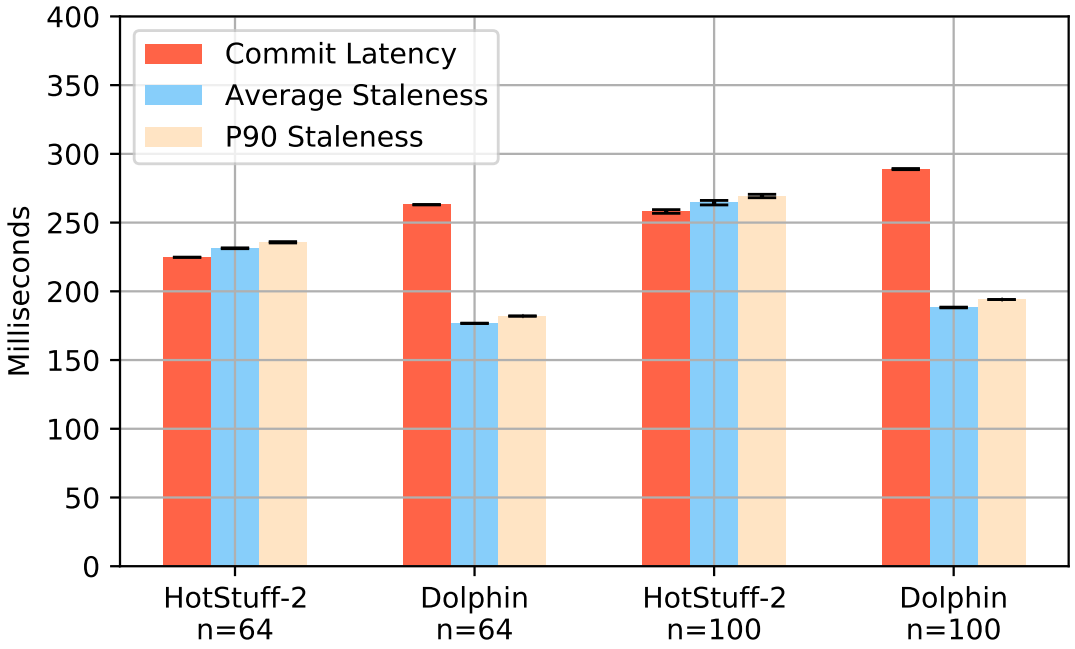


Figure 6.7: Impact of VRB on commit latency of DDTs

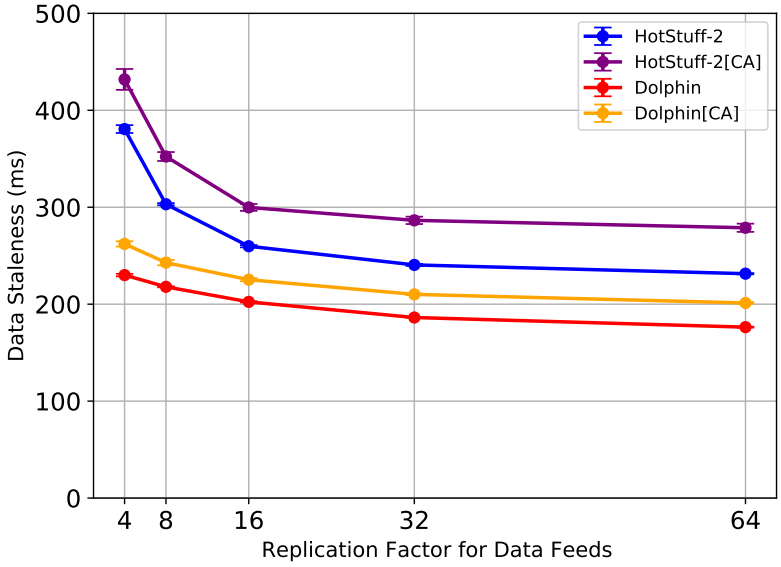
Figure 6.7 illustrates the experiment results. From the graph, we observe that due to the higher message complexity of on-demand VRB, Dolphin incurs a higher commit latency for blocks containing DDTs, around 30ms in both 64-replica and 100-replica networks. However, Dolphin achieves lower data feed staleness, reducing average staleness by 55ms (64 replicas) and 76ms (100 replicas), and P90 staleness by 53ms (64 replicas) and 75ms (100 replicas) compared to HotStuff-2. We will discuss the reduction of data staleness in Dolphin along with the experiment in the later section.

6.8.2 Data Staleness

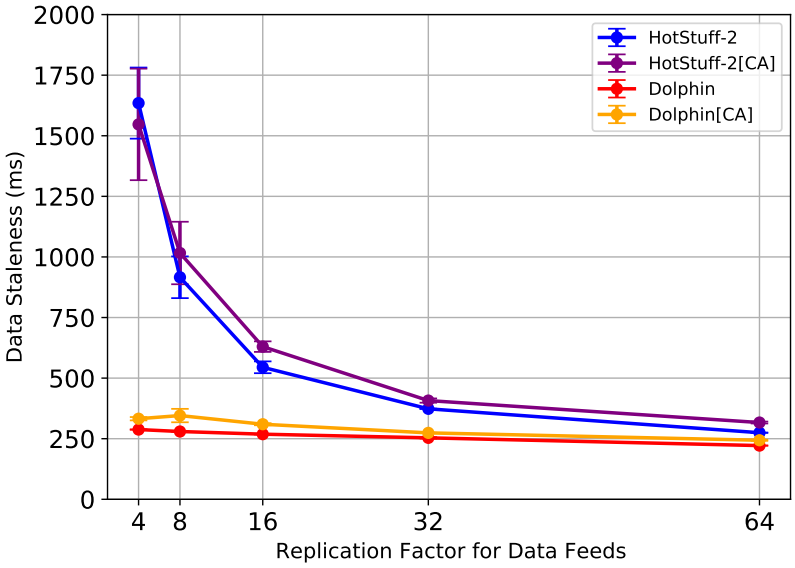
In this experiment, we evaluate data staleness across different replication factors of data records. The experiment is conducted in a cluster with 64 replicas, using data records issued every 10ms and 100ms from a single oracle located in the US-EAST region, chosen for its proximity to major financial data sources. The workload remains identical to the previous experiment. Dolphin demonstrates significantly lower average staleness when data feeds are partially replicated. As shown in Figure 6.8, with a replication factor of 4 for data feeds, Dolphin achieves staleness reductions of 150ms and 1347ms for data records issued every 10ms and 100ms, respectively. With fully replicated data feeds, Dolphin achieves reductions of approximately 50ms for both update intervals. The reduction in staleness stems from two key factors: first, VRB reaches consensus with fewer communication rounds than HotStuff-2, albeit with increased message complexity; second, VRB is more likely to capture the latest data feed. With a correct leader, the data record proposed by VRB is the most recent one reported by at least $f + 1$ replicas. In contrast, HotStuff leaders propose the latest data record stored locally, which may miss the most recent data feed from the oracle due to network delays or message processing order, even with fully replicated data feeds. In general, with partially replicated data feeds, VRB effectively leverages data collected during the collect phase, significantly reducing data staleness. As replication factors approach full replication, the staleness reduction of VRB primarily results from its fewer communication rounds.

6.8.3 Censorship Attack Scenario

We also evaluate the effectiveness of PoR in mitigating high data staleness caused by censorship attacks from a malicious leader. To simulate a rotational leader scheme, the leader conducts censorship attacks with a probability of 1/3. In HotStuff-2, the malicious leader does not propose any data record for its view. In Dolphin, to evade detection by

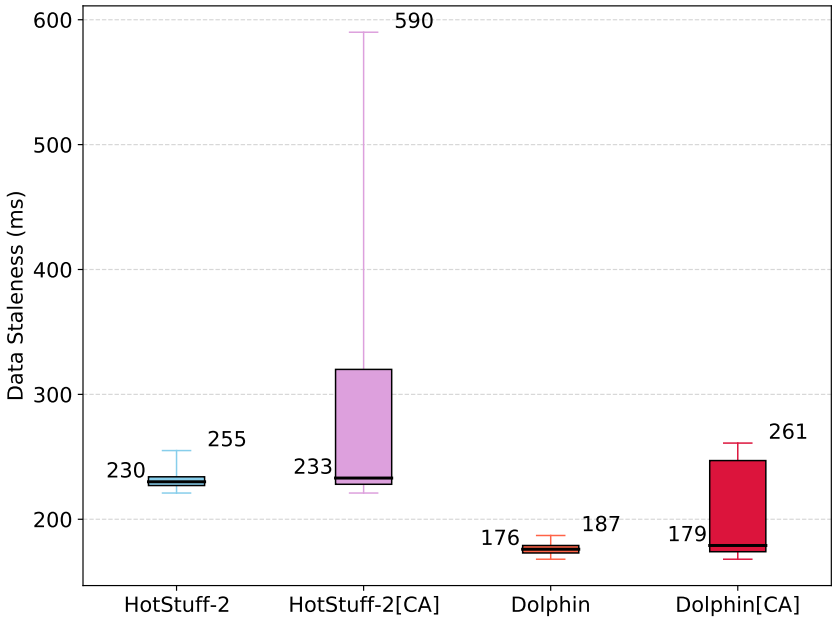


(a) 10ms update interval

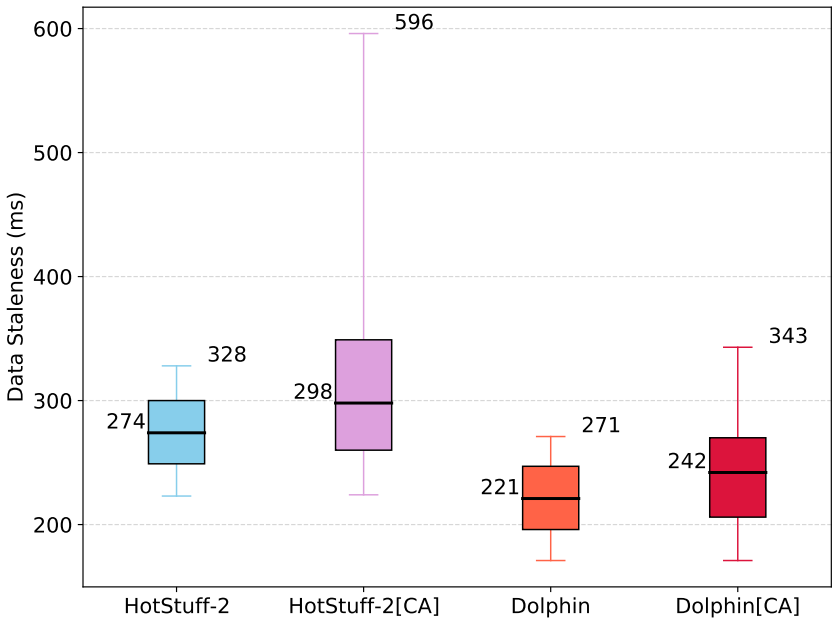


(b) 100ms update interval

Figure 6.8: Average data staleness (ms) versus data feed replication factor in a 64-replica system. `CA` denotes results under censorship attacks.



(a) 10ms update interval



(b) 100ms update interval

Figure 6.9: Box-and-whisker plot illustrating data staleness across different scenarios under a fully replicated data feed.

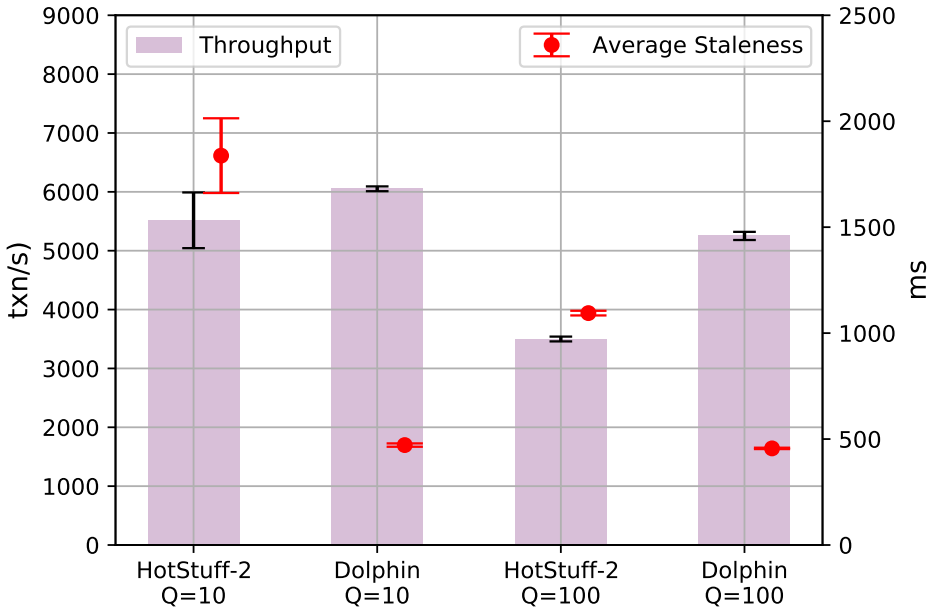
correct replicas, a malicious leader proposes a data record whose timestamp ranks $f+1$ among those reported by correct replicas in the PoR. The line or whisker labeled [CA] presents the results for a scenario in which the leader engages in censorship attacks.

As shown in Figure 6.8, censorship attacks have limited impact on Dolphin. For a 10ms update interval, a malicious leader increases average staleness by at most 30ms. For a 100ms update interval, the increase in average staleness is at most 66ms. In comparison, in HotStuff-2, a malicious leader can increase average staleness by up to 50ms and 100ms for 10ms and 100ms update intervals, respectively.

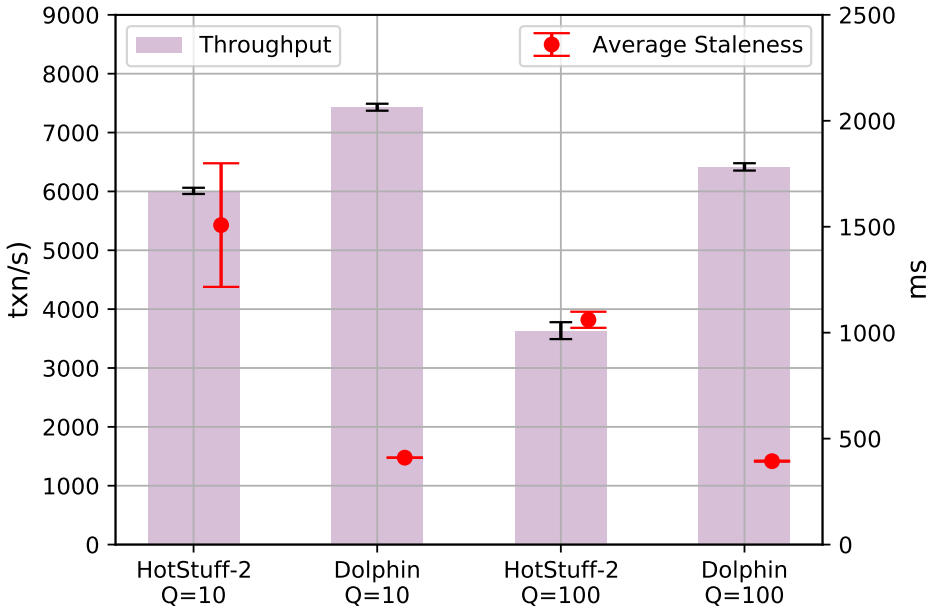
Figure 6.9 presents a box-and-whisker plot of data staleness measured under full replication of data records. The value to the left of each box indicates the median staleness, while the value at the top whisker represents the 99th percentile (P99) staleness. The gap between the P99 and median staleness quantifies the extreme-case impact of a malicious leader. Both Dolphin and HotStuff-2 exhibit an increased P99-to-median gap under censorship attacks. In HotStuff-2, a malicious leader increases this gap by approximately 360 ms for a 10 ms update interval, and 300 ms for a 100 ms update interval. In contrast, Dolphin incurs a smaller increase: approximately 81 ms for the 10 ms interval and 100 ms for the 100 ms interval—corresponding to a roughly 70% reduction in adversarial impact compared to HotStuff-2.

6.8.4 Transaction Processing Throughput

This experiment evaluates the benefits of the on-demand VRB invocation used in Dolphin. The experiment is conducted with 100 replicas. Each replica runs an oracle process that issues updates for 20 unique data feed keys every 100ms. Data feeds are either partially replicated to 10 replicas or fully replicated to all replicas. The data feed keys issued by the oracles do not overlap. The workload is configured as follows: 10% of each batch consists of DDTs, and all DDTs access data feed keys using either a Uniform distribution or a highly skewed Zipfian distribution with $\alpha = 1.5$. For each candidate, the saturation throughput for processing transactions is measured by increasing the transaction batch size until the throughput stabilizes. Note that data feed replication performed by the oracle is not counted as a transaction. We also record the average data staleness for a specific data feed key issued by an oracle located in the US-East region. Figure 6.10 presents the experimental results. Each bar in Figure 6.10 represents the saturation throughput of a candidate protocol with respect to a replication factor Q for oracle data feeds. Figure 6.10a shows the results for a uniform key access distribution, while Figure 6.10b illustrates the results for a skewed key access distribution.



(a) Uniform



(b) Zipfian, $\alpha = 1.5$

Figure 6.10: Saturation throughput (txn/s) and average staleness (ms) for 2K data feeds issued every 100ms by oracles

Full replication of data feeds results in a drop in saturation throughput for both candidates due to the increased processing time spent on receiving and verifying data records. However, HotStuff-2 experiences a more significant throughput drop compared to Dolphin. HotStuff-2 experiences a throughput loss of approximately 40%, while Dolphin loses around 13%. In HotStuff-2, the leader must propose data feed update transactions for the most recently issued data feeds it has received that were not proposed by previous leaders. In contrast, Dolphin employs on-demand invocation of VRB, where a data record is only proposed if it is required by already proposed DDTs. To maintain transaction processing throughput, HotStuff-2 can adopt partial replication for data feeds, but this comes at a significant cost of data staleness (exceeding 500ms). In contrast, the cost of data staleness for Dolphin is negligible, remaining around 20ms. Figure 6.10 further shows that Dolphin benefits from highly skewed data feed key access distributions, as significantly fewer data records are replicated in an on-demand manner. With a skewed key access distribution, Dolphin using partially replicated data feeds achieves approximately 2× higher throughput compared to HotStuff-2 with fully replicated data feeds, while still achieving 40% lower data staleness.

6.9 Chapter Conclusion

To enable on-chain transactions to access fresh data records issued by oracles, we proposed Dolphin, a blockchain system that decouples the processing of oracle data feeds from the cBFT consensus process. In Dolphin, data feeds are partially replicated to a small random subset of replicas, avoiding the overhead of full data replication. Consensus on data records required by DDTs is achieved through Verifiable Recency Broadcast (VRB), a censorship-resilient and communication-round-optimal consensus protocol. VRB is invoked on demand, minimizing unnecessary overhead. Our evaluation demonstrates that Dolphin significantly reduces data staleness and enhances transaction processing throughput with partial replication of data feeds. In a cluster of 100 replicas distributed across four geo-regions, Dolphin achieves up to 2× higher throughput and reduces data staleness by 40% compared to HotStuff-2, which employs traditional full replication of data feeds. For future work, we aim to extend Dolphin's approach to cBFT protocols leveraging more efficient data dissemination protocols.

Chapter 7

Conclusion and Future Works

This thesis demonstrates how the strategic application of weaker coordination protocols can substantially enhance blockchain performance. The study begins by uncovering critical limitations in conventional consensus protocols when applied to key-value (KV) workloads, specifically highlighting the inefficiencies introduced by coarse-grained processing and the immediate propagation of state updates. These design choices amplify latency and limit throughput, particularly in high-load environments where finer-grained coordination is preferable.

To further optimize shared register protocols for write-heavy workloads, this thesis presents WABD, a novel protocol specifically designed for wide-area network (WAN) environments. By leveraging loosely synchronized clocks and asymmetric quorum techniques, WABD minimizes coordination overhead and dynamically adapts to fluctuating client demands. Its design incorporates a clock-based optimization to eliminate one round of communication in write operations and a local-read mechanism to further reduce read latencies. Through adaptive mode-switching, WABD effectively balances low-latency operation under light workloads with high-throughput capabilities under heavy client loads.

The thesis further extends its contributions by addressing performance limitations in oracle data feed processing. To this end, Dolphin is introduced as a blockchain system that decouples oracle data processing from the core BFT consensus layer. By integrating a probabilistic quorum protocol with a censorship-resistant broadcast channel, Dolphin ensures the timely availability of fresh oracle data while mitigating the performance bottlenecks inherent in traditional consensus-based approaches. As a result, Dolphin significantly improves throughput and reduces the staleness of oracle-provided data for on-chain transactions.

Together, these contributions offer a comprehensive perspective on how weaker protocols, when properly applied, can overcome the performance and scalability challenges of blockchain systems. This work lays a foundation for future research into modular blockchain designs where coordination mechanisms are tailored to specific workloads, achieving a balance between efficiency, consistency, and resilience.

7.1 Future Works

During the development of Dolphin, several limitations were identified, primarily resulting from the decoupled processing of transactions and oracle data feeds. A key challenge arises when a failure in the Verifiable Reliable Broadcast (VRB) component prevents the commitment of an oracle value v, while the corresponding transaction tx has already been finalized by the cBFT consensus, causing transaction execution to be blocked. To recover from such inconsistencies, Dolphin relies on PBFT's view-change protocol, which, while correct, may introduce substantial delays that stall not only the affected transaction but also subsequent transactions in the cBFT order. In addition, Dolphin's use of PBFT to finalize oracle data raises a broader question about whether consensus is necessary for this task. Although PBFT is round-optimal in Byzantine settings, its three-phase protocol—including two rounds with quadratic message complexity—incurs high communication costs when applied to rapidly updating oracle feeds. These concerns highlight the need to investigate lighter-weight alternatives that reduce communication costs while preserving sufficient reliability. Another promising direction is to modularize Dolphin's data feed processing module into a plug-in component, enabling seamless integration with the consensus layers of existing BFT protocols and facilitating broader adoption across diverse blockchain platforms. Addressing these limitations and enhancing Dolphin's modular design constitute important directions for future research.

References

[1] Chrony Webpage. https://chrony.tuxfamily.org/, 2021. Accessed: 2021-01-30.

[2] Aave Protocol. Aave: Decentralized Liquidity Protocol. https://app.aave.mex.com/, 2025. Accessed: 2025-03-30.

[3] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 59–74, Brighton, United Kingdom, 2005.

[4] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal Good-Case Latency for Rotating Leader Synchronous BFT. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217, pages 27:1–27:19, Strasbourg, France, 2021.

[5] John Adler, Ryan Berryhill, Andreas G. Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A Decentralized Blockchain Oracle. In *IEEE International Conference on Internet of Things (iThings), IEEE Green Computing and Communications (GreenCom), IEEE Cyber, Physical and Social Computing (CPSCom), and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018*, pages 1145–1152, Halifax, NS, Canada, 2018.

[6] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic Atomic Storage without Consensus. *Journal of the ACM*, 58(2), 2011.

[7] Aarush Ahuja, Vanita Jain, and Dharmender Saini. Measuring Clock Reliability in Cloud Virtual Machines. *Real-Time Intelligence for Heterogeneous Networks: Applications, Challenges, and Scenarios in IoT HetNets*, pages 87–98, 2021.

[8] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 15, pages 1696–1710, 2019.

[9] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2020.

[10] Ifteher Alom, Romana Mahjabin Eshita, Anam Ibna Harun, Md. Sadek Ferdous, Mirza Kamrul Bashar Shuhan, Mohammad Jabed Morshed Chowdhury, and M. Shahidur Rahman. Dynamic Management of Identity Federations Using Blockchain. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia*, pages 1–9.

[11] Abdulaziz Alshaeri and Mohamed F. Younis. A Blockchain-Based Energy Trading Scheme for Dynamic Charging of Electric Vehicles. In *IEEE Global Communications Conference, GLOBECOM 2021, Madrid, Spain*, pages 1–6.

[12] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Modern Large-Scale Data Management Systems after 40 Years of Consensus. In *36th IEEE International Conference on Data Engineering (ICDE 2020)*, pages 1794–1797, Dallas, TX, USA, 2020.

[13] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation*, pages 371–400, 2024.

[14] Elli Androulaki et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th ACM EuroSys Conference*, pages 30:1–30:15, Porto, Portugal, 2018.

[15] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2437–2449.

[16] Aptos Foundation. Aptos Foundation Official Website. https://aptosfoundation.org/, 2025. Accessed: 2025-05-22.

[17] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A Fair Consensus Protocol for Transaction Ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP 2018)*, pages 55–65, Cambridge, UK, 2018.

[18] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.

[19] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2004.

[20] Hagit Attiya and Jennifer L. Welch. Sequential Consistency Versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[21] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.*, 32(4), 2015.

[22] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling Blockchain Innovations with Pegged Sidechains. https://blockstream.com/sidechains.pdf, 2014. Accessed: 2025-05-01.

[23] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous Secure Computations with Optimal Resilience (Extended Abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192, Los Angeles, California, USA, 1994.

[24] Binance Developers. Binance Spot API Documentation: WebSocket Streams. https://developers.binance.com/docs/binance-spot-api-docs/web-socket-streams, 2024. Accessed: 2024-06-14.

[25] Binance Team. Binance Official Website. https://www.binance.com/, 2017. Accessed: 2021-06-30.

[26] Gabriel Bracha. An Asynchronous $[(n-1)/3]$-Resilient Consensus Protocol. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, August 27–29, 1984*, pages 154–162, 1984.

[27] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *J. ACM*, 32(4):824–840, 1985.

[28] Lorenz Breidenbach. Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks. https://research.chain.link/whitepaper-v2.pdf, 2021. Accessed: 2022-08-01.

[29] BTC Relay. BTC Relay: A Bitcoin Ethereum Bridge. https://btcrelay.org/, 2024. Accessed: 2024-12-14.

[30] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 591–617, 2020.

[31] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014. Accessed: 2021-06-30.

[32] Cinthia Paola Pascual Cáceres, José Vicente Berná-Martínez, Francisco Maciá Pérez, and Iren Lorenzo-Fonseca. Blockchain Validity Register for Healthcare Environments. In *Blockchain and Applications - 3rd International Congress, BLOCKCHAIN 2021, Salamanca, Spain*, volume 320, pages 64–73.

[33] Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26–29, 2005, Proceedings*, volume 3724, pages 503–504, 2005.

[34] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, Louisiana, USA, 1999.

[35] Chainlink Labs. Cross-Chain Swap: Education Hub. https://chain.link/education-hub/cross-chain-swap, 2024. Accessed: 2024-12-14.

[36] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, July 2019.

[37] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-Based Speculative Transaction Execution for Ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 570–587, 2021.

[38] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168, Boston, Massachusetts, 2009.

[39] Manuel Coenen, Christoph Wagner, Alexander Echler, and Sebastian Frischbier. Benchmarking Financial Data Feed Systems. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, pages 252–253, Darmstadt, Germany, 2019.

[40] Coinbase Team. Coinbase Official Website. https://www.coinbase.com/, 2017. Accessed: 2021-06-30.

[41] ConsenSys. ConsenSys Quorum Official Website and Documentation. https://consensys.io/quorum, 2025. Accessed: 2025-06-04.

[42] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, Indianapolis, Indiana, USA, 2010.

[43] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 251–264, Hollywood, CA, USA, 2012.

[44] CoreOS. etcd GitHub Repository. https://github.com/etcd-io/etcd, 2013. Accessed: 2021-06-30.

[45] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[46] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190, Seattle, Washington, 2006.

[47] Arav Dalwani. A Secure Blockchain-Based Solution for Management of Pandemic Data in Healthcare Systems. In *Blockchain and Applications - 3rd International Congress, BLOCKCHAIN 2021, Salamanca, Spain*, volume 320, pages 189–198.

[48] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *EuroSys '22: 17th European Conference on Computer Systems*, pages 34–50, Rennes, France, 2022.

[49] DeBank. DeBank: Web3 Portfolio Tracker and DeFi Wallet. https://debank.com/, 2025. Accessed: 2025-03-30.

[50] Diem Association. Diem: A Scalable and Secure Blockchain Payment System. https://www.diem.com/en-us/, 2025. Accessed: 2025-05-22.

[51] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-Efficient Byzantine Fault Tolerance. *IEEE Transactions on Computers*, 65(9):2807–2819, 2016.

[52] John (JD) Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[53] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19*, pages 2028–2041, 2018.

[54] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How Fast Can a Distributed Atomic Read Be? In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 236–245, St. John's, Newfoundland, Canada, 2004.

[55] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Fast Access to Distributed Atomic Memory. *SIAM Journal on Computing*, 39(8):3752–3783, 2010.

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

[57] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A Shared Database on Blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[58] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-Machine Replication for Planet-Scale Systems. In *Fifteenth EuroSys Conference 2020*, pages 24:1–24:15, 2020.

[59] European Central Bank (ECB). Digital Euro: Progress on the Digital Euro Rulebook and Scheme. https://www.ecb.europa.eu/euro/digital_euro/timeline/profuse/shared/pdf/ecb.degov230424_progress.en.pdf, April 2023. Progress report related to the digital euro project. Accessed: 2025-05-25.

[60] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, 2016.

[61] Hua Fan and Wojciech Golab. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proceedings of the VLDB Endowment*, 12(11):1471–1484, 2019.

[62] Federal Reserve Bank of Boston and Massachusetts Institute of Technology (MIT) Digital Currency Initiative. A High-Performance Payment System with a Central Bank Digital Currency: Project Hamilton Phase 1 Appendix. https://www.bostonfed.org/-/media/Documents/Project-Hamilton/Project-Hamilton-Phase-1-Whitepaper.pdf, February 2022. Project Hamilton Phase 1 technical report. Accessed: 2025-05-25.

[63] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[64] FISCO Open Source Working Group. Financial Blockchain Open Source Platform FISCO BCOS Whitepaper. https://raw.githubusercontent.com/FISCO-BCOS/whitepaper/master/FISCO%20BCOS%20Whitepaper(EN).pdf, 2017. Accessed: 2025-06-04.

[65] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the Performance of Chained-BFT. In *41st IEEE International Conference on Distributed Computing Systems (ICDCS 2021)*, pages 595–606, Washington DC, USA, 2021.

[66] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Scaling Blockchain Consensus via a Robust Shared Mempool. In *39th IEEE International Conference on Data Engineering (ICDE 2023)*, pages 530–543, Anaheim, CA, USA, 2023.

[67] Fangyu Gai, Jianyu Niu, Seyed Ali Tabatabaee, Chen Feng, and Mohammad M. Jalalzai. Cumulus: A Secure BFT-based Sidechain for Off-chain Scaling. In *29th IEEE/ACM International Symposium on Quality of Service (IWQOS 2021)*, pages 1–6, Tokyo, Japan, 2021.

[68] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: A zk-SNARK Verifiable Cross-Chain Transfer Protocol Enabling Decoupled and Decentralized Sidechains. In *40th IEEE International Conference on Distributed Computing Systems (ICDCS 2020)*, pages 1257–1262, Singapore, 2020.

[69] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. Hybrid Blockchain Database Systems: Design and Performance. *Proceedings of the VLDB Endowment*, 15(5):1092–1104, 2022.

[70] Johannes Gehrke, Lindsay Allen, Panagiotis Antonopoulos, Arvind Arasu, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Srinath T. V. Setty, Jakub Szymaszek, Alexander van Renen, Jonathan Lee, and Ramarathnam Venkatesan. Veritas: Shared Verifiable Databases and Tables in the Cloud. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, Online Proceedings*.

[71] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proceedings of the VLDB Endowment*, 16(4):629–642, 2022.

[72] Chryssis Georgiou, Nicolas Nicolaou, and Alexander A. Shvartsman. On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, page 425, Toronto, Canada, 2008.

[73] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16, Vienna, Austria, 2016.

[74] Bishakh Chandra Ghosh, Venkatraman Ramakrishna, Chander Govindarajan, Dushyant Behl, Dileban Karunamoorthy, Ermyas Abebe, and Sandip Chakraborty. Decentralized Cross-Network Identity Management for Blockchain Interoperation. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia*, pages 1–9.

[75] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28–31, 2017*, pages 51–68, 2017.

[76] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. BeeGees: Stayin' Alive in Chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing (PODC 2023)*, pages 233–243, Orlando, FL, USA, 2023.

[77] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. https://eprint.iacr.org/2019/1188, 2019. Accessed: 2024-12-14.

[78] Nidhi Gupta, Manik Lal Das, and Sukumar Nandi. LandLedger: Blockchain-Powered Land Property Administration System. In *2019 IEEE International Conference on Advanced Networks and Telecommunications Systems, Goa, India*, pages 1–6.

[79] Stuart Haber and W. Scott Stornetta. How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2):99–111, 1991.

[80] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 175–188, Stevenson, Washington, USA, 2007.

[81] Bahalul Haque, Rifat Hasan, and Oahiduzzaman Mondol Zihad. SmartOil: Blockchain and Smart Contract-Based Oil Supply Chain Management. *IET Blockchain*, 1(2-4):95–104, 2021.

[82] Adiseshu Hari, Murali Kodialam, and T. V. Lakshman. ACCEL: Accelerating the Bitcoin Blockchain for High-Throughput, Low-Latency Applications. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 2368–2376, 2019.

[83] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015.

[84] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[85] Maurice Herlihy and Mark Moir. Enhancing Accountability and Trust in Distributed Ledgers. *CoRR*, abs/1606.07490, 2016.

[86] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[87] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.

[88] Gwan-Hwan Hwang, Po-Han Chen, Chun-Hao Lu, Chun Chiu, Hsuan-Cheng Lin, and An-Jie Jheng. InfiniteChain: A Multi-chain Architecture with Distributed Auditing of Sidechains for Public Blockchains. In *Blockchain - ICBC 2018 - First International Conference, Held as Part of the Services Conference Federation (SCF 2018)*, volume 10974, pages 47–60, Seattle, WA, USA, 2018.

[89] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-HotStuff: A Fast and Robust BFT Protocol for Blockchains. *IEEE Transactions on Dependable and Secure Computing*, 21(4):2478–2493, 2024.

[90] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High Performance State-Machine Replication. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, pages 454–465, Hong Kong, China, 2011.

[91] Ari Juels and Burton S. Kaliski Jr. PoRs: Proofs of Retrievability for Large Files. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28–31, 2007*, pages 584–597, 2007.

[92] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10–12, 2016*, pages 279–296, 2016.

[93] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the*

21st ACM Symposium on Operating Systems Principles (SOSP 2007), pages 45–58, Stevenson, Washington, USA, 2007.

[94] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2010.

[95] Sebastian Kummer, David M. Herold, Mario Dobrovnik, Jasmin Mikl, and Nicole Schäfer. A Systematic Review of Blockchain Literature in Logistics and Supply Chain Management: Identifying Research Questions and Future Directions. *Future Internet*, 12(3):60, 2020.

[96] Jae Kwon. Tendermint: Consensus without Mining. https://tendermint.com/static/docs/tendermint.pdf, 2014. Accessed: 2021-06-30.

[97] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, 1984.

[98] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121)*, pages 51–58, 2001.

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

[100] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. In *Concurrency: The Works of Leslie Lamport*, pages 203–226. 2019.

[101] Grace Lepone and Gary Tian. Usage of Conditional Orders and the Disposition Effect in the Stock Market. *Pacific-Basin Finance Journal*, 61:101302, 2020.

[102] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, 2016.

[103] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Tang, XiaoFeng Wang, and Xiapu Luo. As Strong as Its Weakest Link: How to Break Blockchain DApps at RPC Service. In *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[104] Kai Li, Yuzhe Richard Tang, Jiaqi Chen, Zhehu Yuan, Cheng Xu, and Jianliang Xu. Cost-Effective Data Feeds to Blockchains via Workload-Adaptive Data Replication.

In *Middleware '20: 21st International Middleware Conference*, pages 371–385, Delft, The Netherlands, 2020.

[105] Junxiong Lin, Yajing Xu, Zhihui Lu, Jie Wu, Houhao Ye, Wenbing Huang, and Xuzhao Chen. A Blockchain-Based Evidential and Secure Bulk-Commodity Supervisory System. In *International Conference on Service Science, ICSS 2021, Xi'an, China*, pages 1–6.

[106] Linux Foundation. Hyperledger Sawtooth Project. https://github.com/hyperledger/sawtooth-core, 2018. Accessed: 2021-06-30.

[107] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 305–323, February 2020.

[108] N. A. Lynch and A. A. Shvartsman. Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts. In *Proceedings of the IEEE 27th International Symposium on Fault Tolerant Computing*, pages 272–281, 1997.

[109] MakerDAO. MakerDAO Whitepaper. https://makerdao.com/da/whitepaper/, 2024. Accessed: 2024-12-14.

[110] Daniel Alejandro Maldonado-Ruiz, Jenny Torres, and Nour El Madhoun. 3BI-ECC: A Decentralized Identity Framework Based on Blockchain Technology and Elliptic Curve Cryptography. In *2nd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2020, Paris, France*, pages 45–46.

[111] Dahlia Malkhi and Kartik Nayak. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. *IACR Cryptology ePrint Archive*, page 397, 2023.

[112] Petros Maniatis and Mary Baker. Secure History Preservation Through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Symposium*, pages 297–312, San Francisco, CA, USA, 2002.

[113] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 369–384, San Diego, California, 2008.

[114] J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[115] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing Bitcoin Work for Data Preservation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014*, pages 475–490, 2014.

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

[117] D. L. Mills. RFC1129: Internet Time Synchronization: The Network Time Protocol, 1989.

[118] Henrique Moniz. The Istanbul BFT Consensus Algorithm. https://arxiv.org/abs/2002.03613, 2020. Accessed: 2025-06-04.

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

[120] Giorgio Alessandro Motta, Bedir Tekinerdogan, and Ioannis N. Athanasiadis. Blockchain Applications in the Agri-Food Domain: The First Wave. *Frontiers Blockchain*, 3:6, 2020.

[121] Daniel-Jesus Munoz, Denisa-Andreea Constantinescu, Rafael Asenjo, and Lidia Fuentes. ClinicAppChain: A Low-Cost Blockchain Hyperledger Solution for Healthcare. In *Blockchain and Applications - International Congress, BLOCKCHAIN 2019, Avila, Spain*, volume 1010, pages 36–44.

[122] Matthias Nadler, Felix Bekemeier, and Fabian Schär. DeFi Risk Transfer: Towards A Fully Decentralized Insurance Protocol. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2023)*, pages 1–9, Dubai, United Arab Emirates, 2023. IEEE.

[123] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2008. Accessed: 2021-06-30.

[124] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[125] Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu. Achieving Probabilistic Atomicity with Well-Bounded Staleness and Low Read Latency in Distributed Datastores. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):815–829, 2021.

[126] Amirmohammad Pasdar, Young Choon Lee, and Zhongli Dong. Connect API with Blockchain: A Survey on Blockchain Oracle Implementation. *ACM Computing Surveys*, 55(10):208:1–208:39, 2023.

[127] Gabriele Pellegrino et al. EIP-225: Clique PoA. https://eips.ethereum.org/EIPS/eip-225, 2017. Ethereum Improvement Proposal. Accessed: 2025-06-04.

[128] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. FalconDB: Blockchain-Based Collaborative Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 637–652.

[129] People's Bank of China (PBOC). Progress of Research and Development of E-CNY in China. http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf, July 2021. Accessed: 2025-05-25.

[130] PingCAP. TiDB Webpage. https://etcd.io/, 2015. Accessed: 2021-06-30.

[131] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. Bidl: A High-Throughput, Low-Latency Permissioned Blockchain Framework for Datacenter Networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 18–34, 2021.

[132] Kaihua Qin, Liyi Zhou, Pablo Gamito, Philipp Jovanovic, and Arthur Gervais. An Empirical Study of DeFi Liquidations: Incentives, Risks, and Instabilities. In *IMC '21: ACM Internet Measurement Conference*, pages 336–350, Virtual Event, USA, 2021.

[133] R3. Corda: An Open-Source Distributed Ledger Platform for Business. https://www.r3.com/wp-content/uploads/2016/09/corda_whitepaper_final.pdf, 2016. Accessed: 2025-06-04.

[134] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A Scalable and Massively Parallel Consensus Protocol. In *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies*, pages 426–438, Incheon, Republic of Korea, 2017.

[135] Gökay Saldamli, Vamshi Reddy, Krishna S. Bojja, Manjunatha K. Gururaja, Yashaswi Doddaveerappa, and Lo'ai Tawalbeh. Health Care Insurance Fraud Detection Using Blockchain. In *2020 7th International Conference on Software Defined Systems, Paris, France*, pages 145–152.

[136] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. chainifyDB: How to Get Rid of Your Blockchain and Use Your DBMS Instead. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, Online Proceedings*.

[137] Denisolt Shakhbulatov, Arshit Arora, Ziqian Dong, and Roberto Rojas-Cessa. Blockchain Implementation for Analysis of Carbon Footprint Across Food Supply Chain. In *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA*, pages 546–551.

[138] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the Lines Between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122, 2019.

[139] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT Protocol Forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1722–1743, 2021.

[140] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT Protocol Forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1722–1743, 2021.

[141] Hammad Siddiqi. Anchoring-Adjusted Capital Asset Pricing Model. *Journal of Behavioral Finance*, 19(3):249–270, 2017.

[142] Sierra Chart. Sierra Chart Official Website. https://www.sierrachart.com/, 2024. Accessed: 2024-12-14.

[143] Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. RMWPaxos: Fault-Tolerant In-Place Consensus Sequences. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2392–2405, 2020.

[144] SmartContract Chainlink Ltd. Chainlink: A Decentralized Oracle Network. https://research.chain.link/whitepaper-v1.pdf, 2017. Accessed: 2024-06-24.

[145] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS 2022)*, pages 2705–2718, Los Angeles, CA, USA, 2022.

[146] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-Phase BFT with Linearity. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2022)*, pages 54–66, Baltimore, MD, USA, 2022.

[147] Jian Sun, Gufan Zhao, and Ruiyao Wang. Application of Blockchain Technology in Oil and Gas Industry. In *ICBCT '21: The 3rd International Conference on Blockchain Technology, Shanghai, China*, pages 9–13.

[148] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proceedings of the VLDB Endowment*, 14(6):1019–1032, 2021.

[149] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94)*, pages 140–149, Austin, Texas, USA, 1994.

[150] Tether Limited. Tether: Stable Digital Currency. https://tether.to/en/, 2024. Accessed: 2024-12-14.

[151] Sarah Tollman, Seo Jin Park, and John K. Ousterhout. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021)*, pages 613–632, 2021.

[152] Robbert van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Computing Surveys*, 47(3):42:1–42:36, 2015.

[153] F.B. Vernadat. Interoperable Enterprise Systems: Architectures and Methods. *IFAC Proceedings Volumes*, 39(3):13–20, 2006.

[154] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[155] Qin Wang and Rujia Li. A Weak Consensus Algorithm and Its Application to High-Performance Blockchain. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[156] Michael J. Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling Replicated State Machines with Compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[157] Gavin Wood. Solidity Project Webpage. https://soliditylang.org/, 2014. Accessed: 2021-06-30.

[158] Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. Trade or Trick?: Detecting and Characterizing Scam Tokens on Uniswap Decentralized Exchange. In *SIGMETRICS/PERFORMANCE '22: ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 23–24, Mumbai, India, 2022.

[159] Yingjie Xue and Maurice Herlihy. Hedging Against Sore Loser Attacks in Cross-Chain Transactions. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC 2021)*, pages 155–164, Virtual, Italy, 2021.

[160] Jian Yang, Yawen Lu, Zhihui Lu, Jie Wu, and Hui Zhao. BBCT: A Smart Blockchain-Based Bulk Commodity Trade System. In *Smart Computing and Communication - 6th International Conference, SmartCom 2021, New York City, NY, USA, Proceedings*, volume 13202, pages 186–197.

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

[162] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 270–282, Vienna, Austria, 2016.

[163] Yunhao Zhang, Srinath T. V. Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine Ordered Consensus Without Byzantine Oligarchy. In *14th USENIX Sym-*

posium on Operating Systems Design and Implementation (OSDI 2020), pages 633–649, Virtual Event, 2020.

[164] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. VeriDB: An SGX-Based Verifiable Database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2182–2194, 2021.

[165] Yanchao Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, and Ying Yan. SEBDB: Semantics Empowered Blockchain Database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1820–1831.