Scaling a Hierarchical Coordination Fabric for Geo-Distributed Consensus

by

Shashank Joshi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Shashank Joshi 2026

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

Decentralized Ledger Technology provides system resilience and security but faces significant performance bottlenecks when deployed across geographically distributed replicas. Standard Byzantine Fault Tolerance protocols employ a flat network model where communication costs between all participants are uniform. This abstraction ignores the reality of heterogeneous wide-area network links where inter-continental latencies are up to 60 times higher than local connections. Consequently, existing systems experience throughput ceilings and high coordination costs that worsen as the network scales geographically.

While recent topology-aware protocols attempt to address these disparities through geo-clustering, they remain constrained by rigid architectural limitations. Systems such as GeoBFT and LT-DBFT suffer from high inter-cluster communication complexity (ranging from $O(z^2 f)$ to $O(z^3)$ for z clusters and f faults) and treat hierarchical layers as decoupled black boxes. Furthermore, they lack mitigation mechanisms to recover the stragglers in the system. These designs fail to exploit cross-layer visibility, leaving significant performance gains unrealized in the face of regional load variance and network jitter.

This thesis introduces Polaris, a hierarchical coordination framework designed to optimize geo-distributed consensus through cross-layer information sharing. Polaris enhances the geoscaling of any BFT protocol, assuming a pragmatic limit on the maximum number of faults in each geographic zone, by coordinating the actions of geographically localized BFT groups. Leveraging its hierarchical design, Polaris delegates expensive consensus operations locally at the BFT group level, thereby significantly minimizing cross-group communication while maintaining simpler and more lightweight coordination at the global level. It elects primary envoys (datacenter delegated with global coordination) based on correctness and timeliness, reducing inter-Byzantine Group bandwidth complexity to $O(z^2)$. To address the straggler problem, the system employs PredictiveSkip which leverages metadata embedded in the cross-layer messages to synchronize lagging groups with the global execution window in a single round-trip. Additionally, it also leverages a state-aware control loop to adjust execution pipeline depth based on observed network congestion and regional load drift. Finally, Chained Finality relaxes client-side transaction finality requirements from $C + K$ to $C + 1$ (for cycle C), decoupling transaction finality and pipeline depth (K).

Experimental evaluation via network emulation demonstrates that Polaris achieves a peak throughput of more than 200k transactions per second, representing a 2.6× to 3.6× improvement over state-of-the-art topology-aware protocols at a 48-node scale deployed across 16 datacenters. The PredictiveSkip mechanism maintains high system liveness and incurs only a 16% throughput loss under 3× straggler delays compared to the 41-77% degradation observed in GeoBFT and Steward.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Wojciech Golab and Prof. Bernard Wong, for their invaluable guidance and support throughout this research. I am also deeply thankful to the members of the Shoshin Group and the Distributed Algorithms and Systems Lab (DASL) for their insightful feedback and thought-provoking discussions that helped shape this work.

My deepest thanks go to my parents and my sister for their unwavering support. I am especially grateful to my mother, Geetika Joshi, whose constant motivation and inspiration have been my driving force throughout this entire journey. Finally, I am grateful to the broader research community and my peers at the University of Waterloo for providing such a collaborative and intellectually stimulating environment.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables x

List of Abbreviations xi

1 Introduction 1

2 Background And Related Works 7

2.1 Blockchain 7

2.1.1 Permissionless Blockchain 8

2.1.2 Permissioned Blockchain 9

2.2 Consensus Protocols 10

2.3 RCanopus 12

3 System Model and Cross Layer Optimizations **15**
3.1 Polaris System Design . . . 15
3.1.1 Client Actions . . . 16
3.1.2 Server Actions . . . 17
3.1.3 Monitor Actions . . . 17
3.1.4 Representative and emulator actions . . . 18
3.1.5 Node/SL Join/Re-join Protocol . . . 19
3.2 Optimistic Global Coordination . . . 19
3.2.1 Convergence Score . . . 20
3.3 BFTCompletionProof as a Cross-Layer Metadata Carrier . . . 22
3.4 PredictiveSkip Design . . . 23
3.5 Dynamic Pipeline . . . 28
3.6 Chained Finality . . . 32

4 Safety And Liveness Analysis **34**
4.1 Safety Proofs of Polaris . . . 37
4.1.1 Safety Proof of SkipCycle Optimization . . . 42
4.1.2 Safety Proof of Chained Finality . . . 44
4.1.3 Safety Analysis of Global Coordination Optimization . . . 44
4.1.4 Safety Analysis of Dynamic Pipeline Optimization . . . 45
4.2 Liveness Analysis of Polaris . . . 45
4.2.1 Liveness Analysis of Dynamic Pipeline . . . 47
4.2.2 Liveness Under Network Partition . . . 47

5 Analytical Evaluation of Consensus Properties **48**
5.1 Communication Complexity Analysis . . . 48
5.1.1 Intra-BG Communication Complexity (Local Communication) . . . 49
5.1.2 Inter-BG Communication Complexity (Global Communication) . . . 50

5.2 Fault Tolerance Analysis . . . 51
5.2.1 Case 1: Normal Node Failure . . . 51
5.2.2 Case 2: SL Monitor Failure . . . 52
5.2.3 Case 3: BG Primary Failure . . . 52
5.2.4 Case 4: Envoy Representative Failure . . . 53
5.3 Case 5: Emulator Failure . . . 54

6 Implementation **55**
6.1 Implementation Overview . . . 55
6.2 System Component Design . . . 55
6.3 Implementation Limitations and Design Trade-offs . . . 57
6.3.1 Transaction Deduplication . . . 58
6.3.2 SBFT Pipeline Limitations and Batching Workarounds . . . 58
6.4 Implementation of Baseline Protocols and Polaris Variants . . . 60

7 Evaluation **62**
7.1 Impact of Horizontal Scaling . . . 64
7.2 Impact of Parallel Byzantine Agreement . . . 65
7.3 Impact of Byzantine Group Size . . . 67
7.4 Impact of Straggler Groups . . . 69
7.5 Impact of Optimistic Inter-BG Coordination . . . 70
7.6 Impact of Pipeline Depth . . . 74
7.7 Impact of Finality Semantic . . . 76
7.8 Impact of Predictive Skipping . . . 78

8 Conclusion **81**

References **83**

List of Figures

1.1 Leaf Only Tree (LOT) Structure 3

2.1 Leaf Only Tree (LOT) Structure 13

3.1 Polaris Topology: Four global BGs, each comprising four nodes distributed across distinct cloud providers to achieve $f = 1$ fault tolerance. Each node is implemented as a SL, with a designated monitor server facilitating BG-wide consensus for the SL. 16

3.2 OCM under three operating conditions. *Left:* Optimistic path where the primary envoy fetches $[\{T\}, QC]$ from a remote emulator via a single BCP exchange. *Center:* Emulator omission where the primary envoy falls back to an alternative emulator within the same remote BG. *Right:* Primary envoy failure where f secondary envoys independently contact the remote emulator pool to recover the BFT result. 21

6.1 Main components of Polaris 56

7.1 Throughput and Latency as a function of Cluster Count (z): Performance scaling results for $12z$ replicas. 64

7.2 Throughput and Latency as a function of Cluster Count (z): Performance scaling results for a fixed total of $zn = 48$ replicas. 66

7.3 Throughput and Latency as a function of number of replicas per cluster: Performance scaling results for a fixed total $z = 4$ clusters. 68

7.4 Impact of Straggler Cluster on Throughput and Latency. 69

7.5 Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism. . . . 71

7.6 Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under 10% stochastic Emulator failure probability per cycle. 72

7.7 Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under a targeted straggling attack by a malicious Envoy inducing a sub-timeout propagation delay ($< \tau_{max}$). 73

7.8 Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under 10% stochastic Envoy failure probability per cycle. 74

7.9 Throughput and Latency vs. Pipeline Depth across varying inter-BG Bandwidths (250, 500, 750 Mbps). Bars represent throughput; matching-colored lines represent latency. 75

7.10 Comparison of Throughput-Latency trade-offs between the ChainedFinality and PipelineFinality setup. 77

7.11 Comparison of Throughput-Latency trade-offs between the PredictiveSkip an OneSkip under three different straggler configurations with pipeline utilization for pipeline depth $k = 16$. 78

7.12 Throughput and Latency comparison for PredictiveSkip and OneSkip across different pipeline depths. 79

List of Tables

5.1 Comparative Complexity and Architectural Characteristics † 49
5.2 Comparative Fault Tolerance Thresholds 51

6.1 Polaris variants implemented for ablation study. 61

7.1 Available bandwidth (Gbps) and roundtrip latency (ms) between clusters. 63

List of Abbreviations

BCP BFTCompletion Proof viii, 20, 21

BDP Bandwidth-Delay Product 28, 30, 31, 74

BFT Byzantine Fault Tolerance viii, 1–5, 9–13, 17, 18, 20–28, 32–35, 38, 41, 42, 46, 48–53, 55–60, 65–68, 71, 81, 82

BG Byzantine Group viii, ix, 12–35, 37–42, 44–47, 49–54, 57, 61–63, 65–67, 70, 73–76, 78–80

CFT Crash Fault Tolerance 4, 10–13, 17, 49–52, 58

DLT Decentralized Ledger Technology 1, 7, 8

HoL Head-of-Line Blocking 31, 32, 80, 81

LOT Leaf-Only Tree viii, 3, 4, 13

OCM Optimistic Global Coordination Mechanism 20, 70–73

PoS Proof of Stake 2, 8

PoW Proof of Work 1, 8, 9

QC Quorum Certificate 42, 45, 53, 54

RCM Redundant Coordination Mechanism 70–73

RTT Round-Trip Time 2, 29, 30

SL Superleaf viii, 12, 13, 16–21, 23, 26, 34, 35, 38–41, 44, 46, 49–54, 56, 57, 59, 62, 63, 65, 66, 70, 73, 76

WAN Wide-Area Network 5, 10, 11, 19, 20, 26–28, 81

Chapter 1

Introduction

Decentralized Ledger Technology (DLT) has been increasingly adopted across numerous critical sectors, including finance, healthcare, and supply chain management [10, 52, 3, 27, 17, 18, 8, 47, 48], primarily due to its inherent capacity to deliver resilience and system security. Essentially, DLT implements a distributed, decentralized record system where multiple parties participate in establishing and validating the current state of the record. And to achieve this current state, all the participants must agree on a specific, globally consistent ledger state that is produced by following a set of rules collectively referred to as the consensus protocol. This structure guarantees immutability, transparency, and high availability. However, the application of DLT in these multi-actor systems faces two primary challenges. First, geographically scattered replicas in the network must maintain an identical, consistent state of the ledger. Second, the system must defend against Byzantine failures, including arbitrary malicious behavior, hardware corruption, and coordinated attacks [17, 7], necessitating the need for Byzantine Fault Tolerance (BFT) consensus mechanisms to reliably order transactions across planetary-scale deployments. Although with the BFT consensus mechanism, DLT satisfies the stringent security requirements of these applications, performance remains a major hurdle. Consensus protocols consistently emerge as the primary bottleneck, imposing high latency and severely limiting transaction throughput.

One of the major contributors to this performance gap is the early design choices for the blockchain consensus protocols that did not prioritize performance. Instead, due to the unique decentralized trust model of the DLT, the focus was on correctness and adversarial resilience. This is evident in several foundational systems. Bitcoin's consensus mechanism Proof of Work [40] (PoW), introduced expensive cryptographic puzzles for the selection of block proposer to prevent Sybil attacks and double spending, even though this design

limited throughput to a few transactions per second and imposed confirmation delays on the order of tens of minutes [14]. Nakamoto consensus further chose probabilistic finality sacrificing fast transaction finalization for correctness. Similarly, early Proof of Stake (PoS) [46] protocols such as Ouroboros [29] and Casper [6] emphasized stake-based security, chain finality, and long-range attack prevention, which led to multi-step voting rounds and conservative timeout choices that prolonged decision latency. This trade-off between security and performance prompted several systems to explore alternative designs that restricted participation. This trend eventually motivated the development of permissioned blockchains [25], where membership is fixed and consensus operates within a smaller, more predictable set of participants making it more viable for real-world applications. While these systems eliminated the open-membership challenges faced by public chains such as Bitcoin and Ethereum, and allowed the adoption of classical BFT protocols, they still struggled to achieve high performance. Protocols such as PBFT [7], despite their strong safety guarantees, suffered from quadratic communication patterns and high coordination costs. Even later optimizations, including HotStuff [58], SBFT[23] and other committee based protocols that attempt to reduce the $O(n^2)$ communication overhead of PBFT, still fail to address the challenges introduced by the heterogeneous network model. As a result, these systems continue to experience severe performance bottlenecks when deployed across globally distributed replicas.

Despite this shift toward improving throughput, a fundamental performance barrier persists when deploying BFT protocols at a global scale. The root cause lies in a foundational design assumption embedded in virtually all existing BFT protocols, i.e., the flat network model. These protocols implicitly assume that communication costs between any two participants are roughly equivalent, treating a message sent within a datacenter the same as one traversing continents. While convenient for formal analysis, this abstraction conflicts with real-world network constraints. Empirical RTT measurements show that inter-continental latencies can be 16–60 times higher than local or intra-continental links, while throughput can be 35–260 times lower [37], reflecting the heterogeneous latency and bandwidth characteristics of real-world networks. These disparities clearly highlight that communication between regions is significantly more costly than communication within regions. However, the traditional BFT protocols remain network oblivious, where critical consensus messages traverse these costly intercontinental links repeatedly. This flat network topology-based design imposes a strict throughput ceiling that worsens as the system scales geographically.

Beyond performance, the flat network assumption misrepresents failure patterns. Real infrastructure exhibits correlated failures within geographic boundaries and provider networks, while independent providers in different regions fail largely independently. Network

disruptions predominantly occur at regional boundaries where infrastructure dependencies concentrate. Ignoring these realities means protocols either over-provision for unrealistic failure scenarios or remain vulnerable to realistic correlated failures.

Ideally, a performance centric consensus design for geo scale deployment should distinguish between global and local communication and use this information to maximize bandwidth utilization, since most protocols are bottlenecked by the slowest link in the network. Recent systems such as Steward [2], GeoBFT [24], and LTDBFT [54] acknowledge this challenge and introduce location aware clustering based on the same underlying principle: use topological information to reduce expensive inter regional communication. Although these designs are a meaningful step forward, they remain incomplete. They arrange nodes by geography but do not fundamentally rethink how consensus should operate across multiple network tiers. They lack mechanisms that would allow truly independent parallel progress at different levels of the hierarchy, they provide only limited optimization of information flow across these levels, and they offer no robust method for handling the performance stragglers that inevitably arise in geo distributed environments.

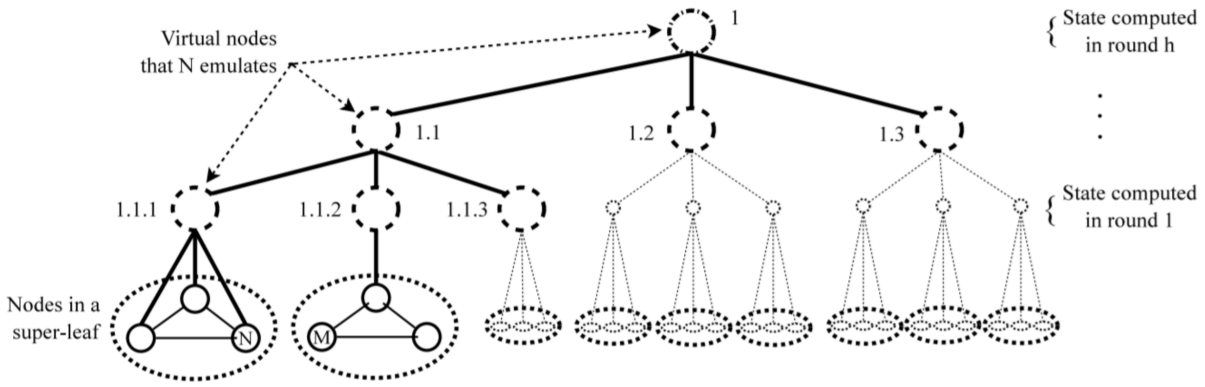


Figure 1.1: Leaf Only Tree (LOT) Structure

In this work, to address the limitations of both state of the art network oblivious designs and the more recent topology aware BFT protocols in permissioned blockchains, we introduce Polaris. Polaris is a multi tier coordination framework for Byzantine consensus that not only acknowledges the hierarchical structure of decentralized networks but also exploits this structure to optimize communication overhead and inter-layer coordination. The design builds upon the overlay architecture established by RCanopus [28], which first

demonstrated the effectiveness of a LOT based hierarchical organization for geo-distributed consensus. As illustrated in the LOT diagram shown in Figure 1.1 (taken from RCanopus [28]), the arrangement of replicas within this overlay and the delegation of responsibilities allow a single replica to emulate the behavior of an entire group. This arrangement reduces the number of participants that must engage in higher level coordination and enables true parallel execution across different phases of the protocol. RCanopus uses the LOT structure to improve bandwidth utilization by separating local and global communication paths. The lower levels of the tree capture communication within a datacenter, while higher levels progressively represent intra region, and eventually inter continental communication. Similar to RCanopus, Polaris organizes its execution temporally into cycles. A cycle contains 3 phases. In Phase 1, Polaris performs a datacenter-wide CFT replication of transactions followed by a region-wide BFT consensus in phase 2 and finally inter-region sharing and coordination of the BFT result.

While RCanopus introduced the promising concept of organizing nodes into geographic hierarchies with Super Leafs and Byzantine Groups (formal definitions covered in Chapter 2), our investigation during protocol extension revealed several limitations in both its design and implementation that significantly constrain performance and prevent the system from fully realizing the potential benefits of hierarchical organization. These include:

- The result aggregation mechanism across hierarchical layers suffered from improper implementation. For instance, multiple batching operations occurring at various consensus rounds introduced unnecessary waiting periods and frequent timeouts. When results from lower layers needed to propagate upward, the batching logic created artificial synchronization barriers that serialized operations that could proceed in parallel, substantially degrading overall system throughput.

- The skip cycle feature, originally intended to mitigate tail latency by allowing fast groups to proceed without waiting for stragglers, had the opposite effect in implementation. Rather than isolating performance degradation to stragglers, the implementation throttled faster clusters to the pace of stragglers.

- The protocol mandated that transactions wait K complete pipeline cycles (where K is the pipeline depth) before achieving finality. This conservative approach, while simplifying correctness reasoning, imposed unnecessary latency penalties. Transactions that could safely commit earlier were forced to wait for additional pipeline stages, directly impacting end-to-end latency and limiting throughput potential.

- The pipeline architecture employed a fixed depth regardless of workload characteristics. While this rigid structure performed adequately under light to moderate loads

with respect to pipeline depth (under new transaction finality rules), it became a significant bottleneck under high load conditions.

- Even after reducing coordination to a smaller number of groups to limit communication overhead, the bandwidth utilization across expensive intercontinental links remained far below optimal levels. The protocol transmitted multiple redundant copies of identical protected consensus data over WAN connections.
- The evaluation framework lacked systematic comparisons with other topology-aware consensus systems such as GeoBFT, Steward, and LTDBFT. Without these comparative benchmarks, it remained unclear how the hierarchical approach stacked up against alternative designs that also attempted to exploit geographic structure.

In this thesis, we make the following contributions:

- We conducted comprehensive correctness verification of the hierarchical consensus implementation, identifying and resolving critical issues for streamlining inter-layer transitions that could compromise system safety and performance.
- We redesign the skip cycle mechanism, where protocol leverages inter-layer state forwarding to make optimal skip decisions. This is discussed in Section 3.4
- We redesign inter-Byzantine Group communication protocols with performance-based envoy selection. This is discussed in Section 3.2
- We redesign and implement novel transaction finality semantics that eliminate mandatory pipeline waiting periods. This is discussed in Section 3.6
- We develop dynamic pipeline depth adaptation mechanisms that maintain correctness under varying load conditions. The system automatically adjusts pipeline depth in response to workload characteristics. This is discussed in Section 3.5
- We provide formal safety proofs demonstrating that Polaris maintains underlying safety guarantees with or without proposed extensions. This is discussed in Section 4.1
- We conduct a comprehensive comparative evaluation against topology-aware BFT protocols. We implement or adapt three state-of-the-art protocols (GeoBFT, Steward, LTDBFT) and deploy them alongside multiple Polaris variants across 48 emulated globally distributed nodes. Our experimental evaluation provides detailed performance comparisons characterizing trade-offs in throughput, latency, communication complexity, and fault tolerance. This is discussed in Chapter 7

- We perform systematic ablation studies isolating individual feature contributions. By implementing and evaluating multiple Polaris variants with different combinations of optimizations enabled, we quantify the performance contribution of each architectural component. This is discussed in Chapter 7

Chapter 2

Background And Related Works

In this chapter we will cover the foundational concepts necessary to understand Polaris and position our work within the landscape of distributed consensus research. This chapter begins by covering the fundamentals of blockchain networks, different blockchain deployment models and their respective requirements, then traces the evolution of consensus protocols from basic crash fault tolerance through Byzantine fault tolerance to modern topology-aware designs. Finally, we analyze existing hierarchical approaches and their limitations, providing the foundation for understanding Polaris' contributions.

2.1 Blockchain

Blockchain is a DLT that enables a set of mutually distrusting participants to maintain a shared, append-only ledger which doesn't need the intervention of a centralized managing authority. This decentralized ledger organizes data into a sequence of cryptographically linked blocks where each block contains a batch of transactions and the hash of its predecessor. This hash linkage provides tamper evidence and ensures ledger integrity [40]. Ledger replicas maintain this record consistency by validating transactions, disseminating updates, and executing a consensus protocol that determines the true ordering and state of blocks. However, the specific mechanisms by which nodes propose, validate, and agree on blocks differ significantly across blockchain designs, giving rise to diverse consensus approaches optimized for varying trust assumptions, security requirements, and performance goals. Understanding these models is essential because they directly influence protocol design choices and achievable performance bounds. Broadly, blockchain systems can be divided into two primary categories: permissioned and permissionless.

2.1.1 Permissionless Blockchain

Permissionless blockchain networks such as Bitcoin and Ethereum operate as completely open networks where any participant can join, leave, and contribute to consensus without authorization. The prominent characteristic of permissionless systems is their open and dynamic membership where the set of validators is unbounded, continuously changing, and potentially adversarial with no identity verification or authentication requirements.

While these blockchain networks are fundamentally more decentralised as compared to other DLT variations, this openness creates a fundamental challenge: Sybil attacks [16], where malicious actors generate numerous pseudonymous identities to overwhelm honest participants and subvert consensus outcome. Traditional quorum-based consensus protocols fail completely in this setting because an adversary could create $3f + 1$ fake identities and unilaterally determine consensus outcomes. Permissionless blockchains address this through economic mechanisms that make identity creation costly. For example, Bitcoin's consensus mechanism PoW, requires participants to solve computationally expensive cryptographic puzzles to propose blocks. The difficulty adjusts dynamically according to hashrate to maintain a target block interval (around 10 minutes for Bitcoin). Security derives from the basic assumption that no adversary controls more than 50% of total computational power and if an attacker had majority hash power, they could execute double-spending attacks by mining alternative chains faster than the honest network. As for transaction finality, PoW provides probabilistic finality where transactions in blocks buried deeper in the chain become exponentially more difficult to revert as an attacker would need to outpace the cumulative work of the honest network.

Similarly, PoS, used by Ethereum 2.0 [30], Cardano [26], and others [57, 56, 21], replaces computational expenditure with capital lockup. Validators stake cryptocurrency as collateral, with selection probability for block proposals proportional to stake amount. Byzantine behavior results in slashing (partial or complete forfeiture of staked capital). PoS variants differ in their selection mechanisms for example Ethereum's Casper employs a committee-based approach with validators organized into slots and epochs, while Algorand [21] uses verifiable random functions (VRFs) for cryptographic sortition that makes validator selection unpredictable and thus resistant to targeted attacks.

This high degree of decentralization and Sybil resistance mechanisms of permissionless blockchains impose severe performance constraints. Bitcoin achieves approximately 7 transactions per second and requires around 6 block confirmations for reasonable finality confidence. Ethereum processes 15-30 transactions per second with 12-15 second block times. These limitations stem from several factors: blocks must propagate across an unbounded network with highly variable latency, the system must accommodate nodes with

limited bandwidth and computational capacity, and the security model requirement where block intervals must exceed network propagation time to minimize fork probability. The FLP impossibility result [4] further constrains permissionless systems operating in asynchronous networks which permissionless blockchains circumvent through either synchrony assumptions or probabilistic guarantees rather than deterministic finality.

2.1.2 Permissioned Blockchain

Permissioned blockchains restrict participation to authorized entities whose identities are known, authenticated, and managed through access control mechanisms. This closed membership model fundamentally transforms security assumptions and enables different consensus approaches that achieve orders of magnitude better performance than permissionless alternatives. The defining characteristic of permissioned systems is authenticated identity with every validator verified against a known set managed by system administrators or a governance consortium. For instance, Hyperledger Fabric employs certificate authorities (CAs) that issue X.509 certificates binding public keys to organizational identities [34]. R3 Corda [5] uses a network map service that tracks participant identities and network addresses. Quorum [11], the enterprise Ethereum variant, restricts network participation through permissioned node discovery and authenticated peer connections.

With known participants, Sybil attacks become irrelevant, as an adversary can no longer arbitrarily create identities rendering the whole attack vector ineffective. This enables traditional BFT consensus protocols that achieve consensus among $n = 3f + 1$ nodes tolerating f Byzantine failures, where f represents a fixed bound rather than a fraction of unbounded participants. Permissioned BFT systems achieve dramatically superior performance to permissionless alternatives. Hyperledger Fabric reports throughput exceeding 20,000 transactions per second under optimal conditions with sub-second latency [34]. This performance advantage stems from several factors: the fixed validator set eliminates block propagation to unbounded participants, cryptographic authentication is substantially cheaper than PoW puzzle solving, and further optimizations like batching can amortize message overhead across many transactions.

Although the transition from permissionless to permissioned systems successfully removed the identity related bottlenecks, it exposed the communication bottleneck. Traditional consensus protocols continue to treat the network as a flat abstraction and assume communication costs between any two participants are equivalent, regardless of whether they reside in the same rack or on different continents.

2.2 Consensus Protocols

In this section, we provide background on distributed consensus, both generally and in the WAN context. The development of consensus protocols has progressed through several evolutionary stages, from Crash Fault Tolerant (CFT) systems to Byzantine fault tolerant (BFT) protocols, and more recently to topology-aware designs that recognize network heterogeneity. We trace this evolution and position Polaris within this landscape.

Early distributed consensus protocols operated under the CFT model, where participants fail silently by stopping execution but never behave maliciously. Seminal CFT protocols include Viewstamp Replication [42], Paxos [32], and its variants [33, 39], and Raft [43]. These protocols achieve consensus among $2f + 1$ nodes in the presence of up to f crash failures, relying on the assumption that failed nodes simply stop participating rather than sending incorrect or conflicting messages. The transition to BFT represented a fundamental shift in the threat model that is more suitable for real-world applications. BFT protocols must achieve consensus even when up to f participants exhibit arbitrary malicious behavior such as sending conflicting messages, colluding with other Byzantine nodes, or attempting to subvert the protocol through timing attacks. This stronger adversarial model necessitates both larger quorum sizes (at least $3f + 1$ total nodes) and additional rounds of authenticated all-to-all communication to prevent Byzantine nodes from forging consensus.

There has been significant past work on BFT protocols starting from PBFT [7] that reduced the complexity of Byzantine agreement from exponential to $O(n^2)$ per consensus round. PBFT demonstrated that BFT consensus could achieve practical performance for replicated state machines, making it viable for real-world deployment. This sparked extensive research into BFT consensus, yielding protocols such as Zyzzyva [31], which optimizes for the common case of no failures; Honeybadger [38], which provides asynchronous BFT; BFT-SMART [49], which offers modular BFT services; and DBFT [13], which targets blockchain applications. More recent work includes HotStuff [58], which achieves linear communication complexity and improved responsiveness under partial synchrony, Algorand [21], which employs cryptographic sortition to randomly select small committees for consensus, reducing the number of active participants, and Kauri [41], which utilizes a bandwidth-aware tree for dissemination but lacks explicit mechanisms to partition nodes into independent groups, instead operating as a single, flat overlay.

Despite these advances, all traditional BFT protocols share a fundamental limitation: they treat the network as a flat, homogeneous structure where all participants communicate with roughly equal latency. Contrary to this network-oblivious design, the slow

nature of WAN links is a key hurdle for scaling any planetary-scale system that spans multiple datacenters, and geo-distributed consensus protocols are no different, as the need for intensive communication between participants limits the scalability of such consensus protocols. A key principle for achieving scale in any geo-distributed system is to partition its state space in order to reduce WAN communication [22]. Geo-distributed systems in other domains such as distributed analytics [44], [53] have successfully used this principle to achieve scale through a partitioned and hierarchical organization. For example, the use of geo-distributed clusters that perform analytics locally and in parallel, followed by a merge of their results, has enabled these systems to achieve one to two orders of magnitude increase in throughput [44], [53]. The Polaris framework leverages these notions of partitioning and parallelism to extend and scale any BFT protocol to geo-distributed settings.

Hierarchical CFT protocols demonstrated early success with topology-aware design. WPaxos [1] employs a multi-leader Paxos protocol that partitions the object space across concurrent leaders in different regions, using flexible quorums to reduce cross-region coordination. Canopus [45] improves CFT consensus scalability within a single datacenter through rack-aware hierarchical organization, restricting consensus to participants in the same rack and using virtual tree overlays for cross-rack message dissemination. These CFT protocols established key principles, including state partitioning, hierarchical organization, and locality-aware communication, which reduce WAN traffic and improve throughput. Atlas [19] extends this to planetary-scale state machine replication by assuming that concurrent failures of entire cloud datacenters are rare, allowing the fault tolerance parameter f to remain small regardless of the number of sites n.

Hierarchical BFT protocols adapt these principles to Byzantine settings, though with limited success. Steward [2] employs location-based clustering with a two-tier architecture but relies on centralized global coordination by the primary cluster that reintroduces WAN latency bottlenecks. GeoBFT [24] similarly partitions replicas into geographic clusters, with each cluster achieving local consensus before engaging in optimistic global sharing. However, GeoBFT's coordination protocol requires each cluster's primary to share decisions with $f + 1$ replicas in every other cluster (where f is the maximum number of Byzantine faults per cluster), resulting in $O(z^2 f)$ inter-cluster messages where z is the number of clusters. This quadratic communication overhead limits GeoBFT's scalability as the number of nodes per geographic cluster grows. LT-DBFT [54] combines location-based clustering with trust-based node selection for IoT deployments, integrating real-time trust evaluation into the consensus process. It addresses GeoBFT's communication complexity by introducing a round-robin-based global sharing mechanism which reduces the WAN message complexity to $O(z^3)$, focusing on denser clusters (i.e., $z \ll f$). However,

LT-DBFT achieves this by adopting a weaker fault tolerance model. By design, it requires an extra layer of trust in cluster primaries for global sharing, assuming a majority of honest primaries on top of local cluster assumptions. While these protocols demonstrate measurable performance improvements over flat-network BFT designs, they fundamentally remain constrained by their limited exploitation of hierarchical structure and their inability to achieve true parallel execution across geographic boundaries.

Across all hierarchical designs reviewed, a common shortcoming emerges: each layer operates with limited visibility into the state of adjacent layers. Existing designs treat local consensus and global coordination as decoupled black boxes. This siloed architecture forces static configuration of critical parameters (such as pipeline depth, quorum composition, and skip logic), parameters whose optimal values are fundamentally cross-layer in nature. No existing hierarchical protocol treats cross-layer state sharing as a first-class design concern, leaving significant performance gains unrealized. Polaris addresses this through a cross-layer information sharing design, where metadata propagated across protocol layers enables dynamic, coordinated optimization on inter-layer transitions of transaction (or batch of transactions) that we detail in the following chapters.

2.3 RCanopus

RCanopus is the direct predecessor of Polaris and the first protocol to demonstrate that hierarchical BFT consensus can be effectively scaled to geo-distributed settings through topology-aware organization. Similar to Canopus [45], RCanopus organizes participants into a two-level hierarchy of SuperLeafs (SLs) and Byzantine Groups (BGs), where SLs are geographically proximate replicas within a region and BGs aggregate multiple SLs across a continent. This two-level aggregation mirrors the natural topology of wide-area distributed systems, where network latency exhibits a pronounced hierarchical structure. As replicas in an SL reside in the same datacenter, they are connected by a high-speed low-latency network. They use a CFT replication protocol, such as Raft [18] or Zookeeper [56], to stay synchronized using an intra-SL atomic broadcast protocol. They also leverage this protocol for leader election. Implementing each BG as clusters of SLs rather than collection of individual servers embodies a pragmatic observation about real-world failures: crash failures (server restarts, transient network issues, software bugs) are orders of magnitude more common than Byzantine failures (compromised servers, malicious operators). It not only makes BGs resilient to crash failures and server churn but also increases overall throughput because servers collect and process transactions in parallel. Note that separation of these two failure classes at different hierarchical levels does not change our fault assumptions

because the Byzantine failure of a single server in an SL can be viewed as a Byzantine failure of the entire SL and the BFT protocol operating at the BG is resilient to this fault.

We now sketch how RCanopus coordinates the BGs to create a global transaction ordering (refer Figure 2.1). In each cycle RCanopus produces a global ordering of all the transactions submitted to it at the start of the cycle. Each cycle has three phases.

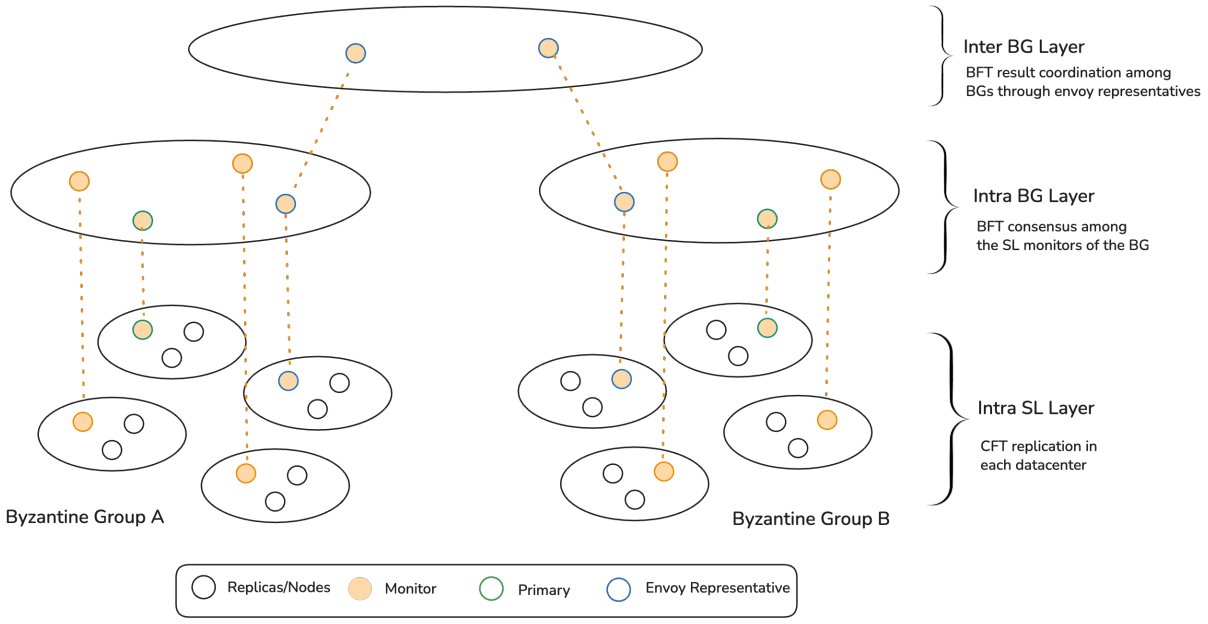


Figure 2.1: Leaf Only Tree (LOT) Structure

In Phase 1, transactions proposed by clients are sent to servers. Each server collects these transactions into a block, assigns each block an ordering ID as the hash of its contents, and exchanges blocks amongst peer servers in its SL using CFT protocol. Servers in each SL use the ordering ID to obtain a unique local ordering. One member of each superleaf is elected to be a leader, called a monitor.

In Phase 2, monitors from SLs in each BG use an arbitrary BFT protocol to form a BFT ordering of the set of transactions they received in the previous phase, and to generate a quorum certificate, consisting of signatures from $2f+1$ monitors, to certify the ordering. Concurrently during this phase, the BFT leader of each BG designates $2f+1$ SLs as envoys for the current cycle.

In Phase 3, envoy SLs initiate requests to fetch ordered transaction lists from all other

BGs, then merge these lists to construct a unique global ordering. Once constructed, the global ordering is logically replicated throughout the BG.

Chapter 3

System Model and Cross Layer Optimizations

In this chapter, we describe the Polaris architecture and the specific cross-layer optimizations designed to address the scalability and performance bottlenecks of RCanopus. We first introduce the Optimistic Global Coordination mechanism, which improves the utilization of inter-BG bandwidth by electing a primary envoy based on a Convergence Score that weighs correctness against timeliness. Next, we reconceptualize the BFTCompletionProof from a simple status signal into a cross-layer metadata carrier, providing the information needed for subsequent optimizations at zero additional communication cost. Our third extension, Predictive Skip-Cycle, addresses the straggler problem. By utilizing the cross-layer state, it allows lagging groups to synchronize with the global ingestion horizon in a single round-trip. We then present the Dynamic Pipeline, a state-aware control loop that adjusts pipeline depth based on both network-layer congestion and regional load drift. Finally, we introduce Chained Finality, a $C + 1$ ordering model that decouples the global transaction finality from the total pipeline depth to minimize finality lag.

3.1 Polaris System Design

In this section, we describe the design of our system (refer Figure 3.1), elaborating the actions taken by each role. The output from Polaris is a unique global ordering on transactions that all participant nodes agree to. The actual transaction execution is not in the scope of our design. Note that although we discuss actions in each cycle independently, Polaris is pipelined, so multiple cycles are in progress simultaneously.

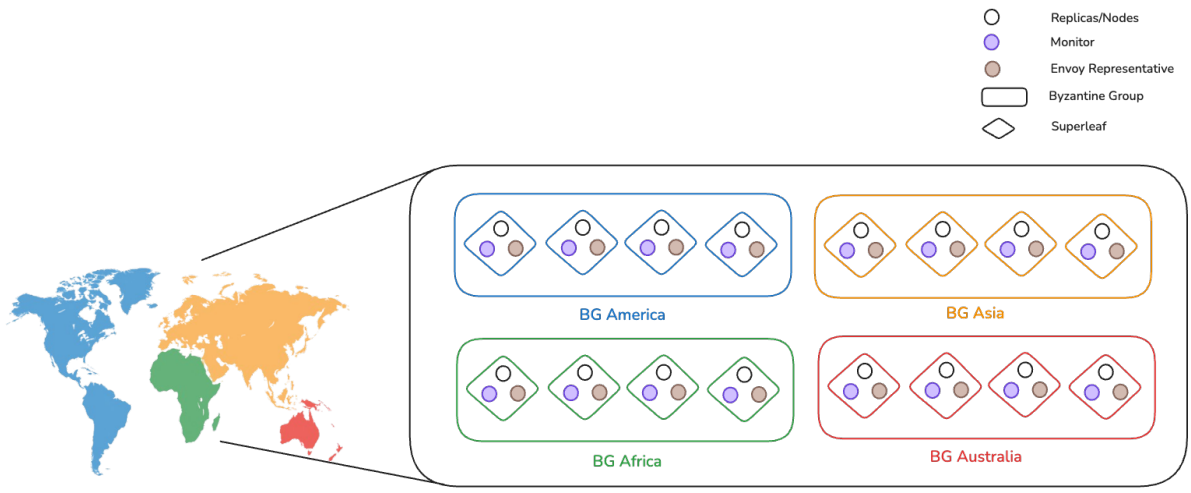


Figure 3.1: Polaris Topology: Four global BGs, each comprising four nodes distributed across distinct cloud providers to achieve $f = 1$ fault tolerance. Each node is implemented as a SL, with a designated monitor server facilitating BG-wide consensus for the SL.

3.1.1 Client Actions

Clients submit transactions to a server within their local region, with each transaction accompanied by a client-generated transaction ID, a client-generated nonce, and a signature from the client's private key. Following the approach in PBFT [7], clients initially direct transactions to servers in their nearest SL, thereby minimizing latency.

To protect against Byzantine servers that may deliberately ignore transactions, clients employ a timeout-based retry mechanism. If a transaction fails to appear in Polaris's output within a configurable timeout period, the client resubmits the request to servers in f additional distinct SLs within the same BG, thus preserving geographic locality while ensuring delivery.

To mitigate Byzantine client behavior, the system enforces transaction consistency: duplicate transactions from the same client sharing identical ID and nonce values must be bytewise identical. Violations of this constraint result in the client being flagged as Byzantine[1]. Upon completing a cycle, servers provide clients with cryptographic proof that their transactions have been incorporated into the global transaction order. This proof comprises the complete set of transactions for the cycle along with the corresponding

[1]This check, however, is not currently implemented in our prototype.

BG quorum certificates.

3.1.2 Server Actions

Servers perform four principal functions within the Polaris protocol. First, transaction reception and ordering ID assignment – servers receive transactions from clients and assign an ordering ID to each block of transactions received during the first phase of each cycle. The ordering ID is computed as a cryptographic hash of the block's contents, providing a unique and non-manipulable identifier for the block's position in the global order.

Second, intra-SL consensus – during the first phase of each consensus cycle, servers participate in an intra-SL Crash Fault Tolerant (CFT) replication protocol to establish a total ordering of all transactions received within the SL. Transactions are ordered by increasing ordering ID. Upon completion of this protocol, all operational servers within the SL reach agreement on the transaction order. If a server fails before replicating its state to at least one peer server within its SL, its state becomes inaccessible and its transactions are lost to the system. However, affected clients will subsequently retransmit these lost transactions (refer Section 3.1.1).

Third, client notification – at the conclusion of each cycle, servers return the ordered transaction set computed during that cycle to their respective clients. Fourth, monitor liveness tracking – servers within each SL continuously monitor the liveness of their designated monitor node. Upon detecting monitor failure, servers invoke a leader election protocol to select a new monitor, which then assumes BG membership.

3.1.3 Monitor Actions

The monitor within each SL serves as the primary coordinator for establishing a BG-wide transaction ordering during the second phase of each cycle. At the conclusion of this phase, each monitor obtains a total ordering of client transactions accompanied by a quorum certificate containing $2f_i + 1$ signatures from BG_i that validates this ordering. Byzantine monitor behavior can potentially impact both Phase 1 and Phase 2 operations. However, since Phase 2 employs Byzantine Fault Tolerant (BFT) consensus, any attempt by a Byzantine monitor to manipulate the outcome of these phases will trigger its replacement through the leader election mechanism. Recall that clients may send their transactions to multiple servers in multiple SLs, which causes duplicate transactions to enter the system. These duplicates are identified using the per-transaction client-generated nonce and must

be eliminated during BG quorum certificate computation. Since client-initiated transactions may arrive at different servers at varying times and thus participate in different Polaris cycles, monitors maintain a buffer containing transaction hashes from at least the most recent k cycles during the BG consensus protocol, removing duplicates as necessary.

3.1.4 Representative and emulator actions

A designated subset of servers within each SL are elected as SL representatives, responsible for retrieving remote BG state during Phase 3 of every cycle. Representatives query emulators to discover the transaction ordering at remote BGs. Since emulators may exhibit Byzantine behavior and fail to respond, representatives implement a fault-tolerant request strategy. However, because emulators return cryptographically signed results, they cannot equivocate in their responses.

To maintain performance under emulator failures, representatives initially contact a single emulator in each target BG. They send a local quorum certificate for the requested cycle as proof that the BG has progressed to cycle C. This message is termed as BCP ($BFTCompletionProof$). A retry timer is started concurrently. Upon timeout expiration, the representative contacts f_i additional emulators from f_i distinct SLs within BG_i. If all emulators for a particular BG become inaccessible, for instance, due to network partition, the representative enters a stalled state.

Upon receiving a state request, an emulator responds with either: (1) a state response containing the requested state and an accompanying quorum certificate, or (2) a null response indicating that the desired state is not yet available, with a commitment to send the complete response once it becomes available[2]. The quorum certificate, obtained as output from the BFT protocol, consists of a signature set verifying the hash of the underlying transactions concatenated with the current cycle number. This construction provides proof of both the specific transaction ordering and the completion of that cycle.

On receiving a non-null response, if the quorum certificate contains at least $2f_i + 1$ signatures (the minimum quorum size for the underlying BG), the representative broadcasts this response to all peer servers within its SL. Other representatives attempting to obtain responses from the same BG abort their pending requests upon receiving such a broadcast. The monitor tracks the operational status of all representatives, categorizing each as live, stalled, or failed. When a representative fails, the monitor promotes a new server to the

[2]This null response and future commitment represent a skip scenario where the concerned BG has skipped the cycle and the BFTCompletionProof serves as commitment response.

representative role, which assumes all responsibilities of the failed representative. If all representatives become stalled, for example, due to network partition, the monitor also stalls, eventually causing Phase 3 of the current cycle to stall and sacrificing liveness to maintain safety.

3.1.5 Node/SL Join/Re-join Protocol

A new or a newly-live node makes its presence known to the monitor in its SL, which sends announcements to monitors in peer SLs proposing that this node now be considered alive. If a quorum of monitors in the BG agree, then the node's status is updated by other monitors in the BG and in the global membership service. The new node is given all the state that it missed by one of its peers, which is the set of its missed transactions along with their quorum certificates.

3.2 Optimistic Global Coordination

The BG envoy extension in RCanopus reduces inter-BG bandwidth consumption by restricting Round 3 fetching to a designated subset of $2f + 1$ envoy SLs rather than all SLs in the BG. Following the notation of RCanopus, suppose we have z BGs each with $N = 3f+1$ SLs and transaction batch size T. The best-case WAN bandwidth consumption of RCanopus with $R = 1$ representative per envoy SL is:

$$bw_{\text{RCM}}^{\text{best}} = z \cdot (2f + 1) \cdot (z - 1) \cdot T \tag{3.1}$$

and in the worst case, where each envoy must contact $f + 1$ emulators per remote BG:

$$bw_{\text{RCM}}^{\text{worst}} = z \cdot (2f + 1) \cdot (z - 1) \cdot (f + 1) \cdot T \tag{3.2}$$

Since $N = 3f + 1$, the best-case bandwidth grows as $O(z^2 f)$, meaning both horizontal scaling (adding more BGs) and vertical scaling (more nodes per BG) compound the inter-BG bandwidth cost. This creates two concrete problems. First, the $2f + 1$ envoys are all placed on the critical path simultaneously and as a result in the common case where no failures occur it still incurs the full redundancy cost. Second, envoy selection during Phase 2 is arbitrary with no consideration of historical reliability, meaning Byzantine or slow SLs are equally likely to be selected as reliable ones. Together, these limitations establish a bandwidth bottleneck that prevents RCanopus from scaling horizontally without proportional WAN bandwidth growth.

Polaris introduces the Optimistic Global Coordination Mechanism (OCM), which restructures inter-BG coordination around a single primary envoy per remote BG on the critical path, with f secondary envoys activated only upon failure. In the optimistic case, OCM requires exactly one fetch per remote BG per cycle, reducing WAN bandwidth consumption to:

$$bw_{\text{OCM}}^{\text{best}} = z \cdot (z-1) \cdot T \tag{3.3}$$

representing a reduction by a factor of $2g+1$ over RCanopus in the common case. In the worst case, where the primary envoy fails and all f secondary envoys must fetch:

$$bw_{\text{OCM}}^{\text{worst}} = z \cdot (z-1) \cdot (f+1) \cdot T \tag{3.4}$$

Figure 3.2 illustrates the three operating conditions of OCM. In the optimistic case (*left panel*), the primary envoy sends a BCP message to a single emulator in the remote BG and receives the BFT result $[\{T\}, QC]$ (where $\{T\}$ is list of transactions and QC is corresponding quorum certificate) in return, keeping the inter-BG critical path lean with a single message exchange. When the contacted emulator is faulty (*center panel*), the primary envoy contacts f additional emulators within the remote BG before issuing a signed negative response, triggering the fallback. Under primary envoy failure (*right panel*), the f secondary envoys activate independently, each contacting the remote emulator pool at random until a valid BFT result is recovered.

3.2.1 Convergence Score

The effectiveness of OCM in the steady state depends on the quality of the primary envoy selection. A poorly selected primary envoy triggers a fallback mechanism that undoes the bandwidth savings OCM provides. Ideally, the primary envoy for a BG g should be the SL that is most likely to obtain the correct remote BFT result for g quickly and without triggering a fallback mechanism. This requires a selection criterion that captures two orthogonal parameters of reliability: First correctness which defines whether the envoy obtains the right result, and second timeliness, which handles whether it fetches that result fastest as compared to other envoys. Neither dimension alone is sufficient. An envoy that always obtains the correct result but only after a long delay still stalls the pipeline and reduces effective pipelining depth. Conversely, an envoy that responds quickly but with incorrect results is Byzantine and must be penalized regardless of its speed. For each envoy SL s of BG g, the Convergence Score $C(s,g)$ is computed by the BG leader at the end of each cycle as a weighted combination of two components:

$$\mathcal{C}(s,g) = \alpha \cdot \mathcal{A}(s,g) + (1-\alpha) \cdot \mathcal{T}(s,g) \tag{3.5}$$

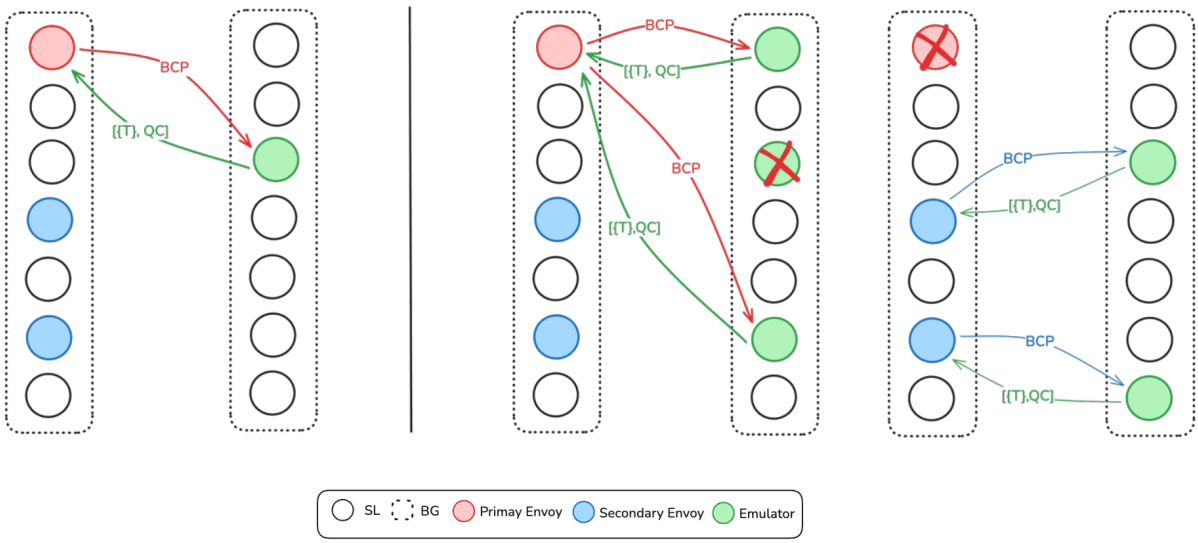


Figure 3.2: OCM under three operating conditions. *Left:* Optimistic path where the primary envoy fetches $[\{T\}, QC]$ from a remote emulator via a single BCP exchange. *Center:* Emulator omission where the primary envoy falls back to an alternative emulator within the same remote BG. *Right:* Primary envoy failure where f secondary envoys independently contact the remote emulator pool to recover the BFT result.

where $A(s, g)$ is the agreement rate of SL s with the final BFT result of BG g and $T(s, g)$ is the normalized timeliness of SL s in obtaining results from BG g, defined as:

$$\mathcal{T}(s, g) = 1 - \frac{\tau(s, g)}{\tau_{\max}} \tag{3.6}$$

where $\tau(s, g)$ is the observed fetch latency of SL s for remote BG g and $\tau_{\max}$ is the cycle timeout. The weighting parameter α controls the relative importance of correctness versus timeliness and is a system configuration parameter[3].

[3]In the current prototype, $\alpha = 0.5$ is utilized to represent a neutral weighting between the agreement rate $\mathcal{A}(s, g)$ and normalized timeliness $\mathcal{T}(s, g)$.

3.3 BFTCompletionProof as a Cross-Layer Metadata Carrier

In RCanopus, the BFTCompletionProof serves a single purpose: to prove that a remote BG has completed Round 2 for a given cycle. The proof is constructed by augmenting the SHA256 digest of the committed BFT result along with two additional fields `lastCycle` and `currentCycle`

$$\text{Digest}(x) = \text{SHA256}(x) \;||\; x.\texttt{lastCycle} \;||\; x.\texttt{currentCycle} \tag{3.7}$$

This construction is elegant in its simplicity where by leaking the cycle number fields into the digest, the proof can be verified without transmitting the full transaction payload, keeping the message size to 72 bytes (40-byte digest plus 32-byte threshold signature). The quorum certificate backed by $2f + 1$ BLS threshold signatures ensures the proof is cryptographically non-repudiable and tamper-evident.

However, this design treats the BFTCompletionProof purely as a completion certificate which serves two primary purposes: First, as a binary signal that a cycle has finished and second, to provide count of skip cycles. It carries no information about the internal state of the producing BG beyond the cycle boundary, and provides no visibility into the BG's current processing position, Phase 1 state, or pipeline utilization. As a result, any protocol layer that receives a BFTCompletionProof knows only that cycle C is done and it cannot infer where the BG is now, what cycle it will finalize next, or whether it is falling behind the rest of the system.

Polaris reconceptualizes the BFTCompletionProof from a completion certificate into a cross-layer metadata carrier which serves as a cryptographically signed state snapshot that propagates the internal execution state of a BG across protocol layers and across BG boundaries at every cycle, at zero additional communication cost. The enhanced BFTCompletionProof extends the digest to carry an additional field beyond the original `lastCycle` and `currentCycle` which tells the local pipeline utilization of the BG. The enhanced BFTCompletionProof is the information substrate that enables all of Polaris' cross-layer optimizations (refer Section 3.4 and 3.5). The key insight underlying this design is that the BFTCompletionProof already flows between every pair of interacting BGs at every cycle as part of the normal Round 3 protocol path. Embedding additional metadata into the digest costs nothing in terms of extra messages or round-trips because the metadata piggybacks on communication that would occur regardless. The result is that every BG maintains an up-to-date, cryptographically verified view of every remote BG's internal

state at every cycle, enabling the full suite of Polaris' adaptive optimizations without any coordination overhead beyond what the base protocol already incurs.

3.4 PredictiveSkip Design

This section introduces PredictiveSkipCycle optimization which is designed to minimize the synchronization overhead and performance penalties that arise when constituent BGs operate at varying progress speeds. By leveraging cross layer state information, the protocol maintains high throughput even in heterogeneous environments.

The Straggler Problem and Existing Approaches

In geo-distributed systems, when a BG falls behind due to regional congestion, hardware asymmetry, or transient network faults, faster BGs are prevented from advancing the global cycle counter beyond the straggler's current state plus the pipeline depth (the additional execution window). Consequently, system-wide throughput is throttled to the processing rate of the slowest member. This bottleneck is further compounded in modular-hierarchical consensus models like Polaris, where in a real world deployment we deploy each SL in a different datacenter, so the latency between them will be different and the global finality floor becomes sensitive to aggregate variance.

Existing approaches to mitigate this issue fall into three categories, each with fundamental limitations in the geo-distributed setting. First, *no-op* mechanisms, as used in Mencius [36] and GeoBFT [24], fill a straggler's pending slot with an empty placeholder. However, empty cycles are pipeline bubbles that do not reduce the straggler's incoming transaction queue. Thus, the straggler wastes phase time processing no transactions while its backlog continues to grow, forcing larger message sizes and longer BFT latency in subsequent cycles. Second, *timeout-based* leader replacement, as used in HotStuff and DiemBFT [15], rotates the straggler out after a timeout, but each rotation incurs a full timeout penalty and view synchronization cost, draining the pipeline of in-flight transactions. Finally, *reputation-based* selection, as used in Aptos [9], avoids electing historically slow participants but relies on lagging indicators computed over past epochs and cannot react to transient congestion within a single cycle.

RCanopus SkipCycle and its Limitations

RCanopus introduced a different SkipCycle design (we will be referring to it as OneSkip) to address the straggler problem directly by decoupling the BFT sequence number inside a BG from the global RCanopus cycle number. When a straggler BG receives a fetch request from a remote BG for a cycle it has not yet completed, the emulator submits the BFTCompletionProof of the highest requested cycle to its local SBFT leader. The leader then sets:

$$currentCycle = max(lastCycle + 1, maxRequestedCycle) \tag{3.8}$$

allowing the straggler to skip ahead in cycle numbers without committing transactions for the skipped cycles. For skipped cycles, the BFTCompletionProof of the next non-skipped cycle serves as proof of their vacancy. While SkipCycle is a meaningful improvement over no-op bubbles and avoids the high cost of a full View Change, it has three fundamental limitations. First, it is purely reactive in nature where a skip is only issued after and according to the fetch request arrives from a remote BG, meaning it avoids an immediate stall but does not provide a structured recovery path for the straggler. Without an optimized feedback loop, skip requests act as temporary bypasses rather than a mechanism that allows the straggler to efficiently clear its backlog and reintegrate into the high-speed consensus flow. Second, the skip mechanism is subject to head-of-line blocking of the pipeline. OneSkip relies on static skip increments, which lead to pipeline under-utilization. In a geo-distributed setting with high variance in transaction processing rate, a fixed skip amount is often either too conservative causing unnecessary waits and premature pipeline saturation for faster BGs. Third, lack of global pipeline awareness, the decision to skip is typically made in isolation, lacking global pipeline awareness. Without a holistic view of the execution window, the protocol may trigger a skip that is locally optimal but globally suboptimal, resulting in jitter where the system oscillates between idling and skipping.

PredictiveSkip Design

Polaris addresses all three limitations through the PredictiveSkip mechanism, which anticipates straggler drift before it manifests using cross-layer metadata already flowing through the normal protocol path. In Polaris, the BFTCompletionProof carries three fields: the current cycle being finalized, the last executed cycle, and last Prepare message cycle number[4]. This richer state snapshot gives the straggler BG precise visibility into the faster

[4]The last committed Phase 1 cycle serves as a functional replacement for the last Prepare message cycle number to accommodate the lack of native pipelining support in the Phase 2 BFT layer.

BG's state across all three protocol layers without any additional communication. Following the phase 3 global coordination scheme, when a fast BG at cycle C_{fast} attempts to fetch a result from a straggler BG (C_{slow}), the fast BG's emulator attaches its own signed BFTCompletionProof for C_{fast}. Upon receiving this proof, the straggler does not merely attempt to resolve the current fetch; instead, it uses the embedded Last Prepared Cycle to estimate the fast BG's future trajectory. The straggler then issues an Optimal Skip Request backed by the BFTCompletionProof for C_{fast}. This eliminates reactive lag by resolving the skip within the same round-trip as the initial fetch request. Because the skip target is derived directly from the $2f + 1$ signed state of the fast path, the straggler can jump to the most advanced speculative cycle safely, ensuring it reintegrates at the head of the pipeline rather than falling behind again during the skip's processing time.

Unlike OneSkip, which only reacts after a fetch request fails, Polaris utilizes the BFTCompletionProof to synchronize the straggler pre-emptively. By providing the straggler with a signed snapshot of the fast BG's Last Prepared Cycle, the straggler identifies the required jump target during the initial fetch-response round-trip. This ensures that the recovery process begins before the straggler has inhibited global progress, effectively aligning the regional state with the global execution horizon in real-time. Also, OneSkip frequently suffers from reactive skipping, where the straggler jumps to a cycle that is already stale by the time the skip is finalized. Polaris solves this by targeting the leading edge of the fast BG's Phase 1 window rather than just the currently requested cycle. By using the Last Prepared metadata as a look-ahead anchor, the straggler calculates a skip target that accounts for the latency of the skip resolution itself. This ensures the straggler lands at the head of the global pipeline, preventing the stochastic jitter and repeated window saturations characteristic of reactive systems.

The transition to a predictive model fundamentally shifts the recovery objective from pipeline bubble management to execution window alignment. While OneSkip improved upon static no-ops by decoupling local BFT sequences from global cycles, its incremental skip logic effectively acted as a performance barrier, forcing the faster BGs to throttle their transaction processing rate to match the straggler's reactive recovery pace. Polaris removes this constraint by decoupling the skip target from specific cycle requests and instead anchoring it to the speculative pipeline utilization of the fast BGs. Rather than throttling the fast BGs to accommodate a straggler, Polaris forcefully accelerates the straggler toward the maxima of global pipeline utilization, ensuring that the majority maintains its peak processing capacity. A detailed empirical verification of this design choice is provided in the evaluation in Section 7.8.

Skip Design Mathematical Interpretation

We formalize the straggler recovery dynamics established above. To provide a rigorous mathematical comparison, first we define the variables governing the skip mechanism of two BGs (one straggler and one fast BG scheme):

t_{fast}: Pipelined phase duration for faster BG (Phase 1 ingestion window duration)

t_{slow}: Pipelined phase duration for straggler BG (Phase 1 ingestion window duration)

t_{bft}: BFT phase processing time for straggler BG

X: The round-trip WAN latency between BGs

K: Pipeline depth

T: Current cycle lag (difference of current cycle at straggler and fast BG)

Axiom 3.1.
In any practical geo-distributed deployment of Polaris, the following inequality holds:

$$t_{slow} \ll t_{bft} \tag{3.9}$$

This inequality is justified by the disparate computational and communication complexities of the two protocol phases: t_{slow} (Phase 1 Ingestion): Represents a single pipelined phase (In this specific case, the duration required for a BG to ingest transactions to form TBs) t_{bft} (Phase 2 Finality): Represents the total duration of the Phase 2 BFT consensus process. This involves multiple rounds of inter-SL voting to reach a $2f + 1$ quorum.

First, we will analyze the recovery time for the models where the faster BG is T cycles ahead the straggler BG. For this specific case we are assuming the system is stalled and there are no more client requests served by the concerned deployment. In PredictiveSkip, the skip decision is decoupled from the straggler's local BFTCompletionProof queue. Since the skip is triggered by the state information from BFTCompletionProof attached to the fetch-response, the recovery occurs within a single round-trip plus local BFT phase:

$$T_{rec_pred} = X + t_{bft} \tag{3.10}$$

Notably, T_{rec_pred} is independent of the pipelined phase duration of straggler BG because it leverages the metadata from BFTCompletionProof. This allows Polaris to recover even when the fast path is running at maximum frequency. In the reactive OneSkip model, the skip request is inherently coupled to the cycle requests received by the straggler. The recovery time is not a single round-trip, but an accumulation of delays governed by the

rate at which fetch requests arrive and the time required to process them through the straggler's already-lagging BFT layer. For a drift of T cycles, the reactive recovery time is modeled as:

$$T_{rec_reac} = \frac{T \cdot (2t_{fast} + X) \cdot (t_{bft} + X)}{2t_{slow}} \tag{3.11}$$

The mathematical contrast between T_{rec_pred} and T_{rec_reac} reveals a fundamental architectural divide: First, The Polaris recovery time is constant relative to the drift T. Whether the straggler is behind by 1 cycle or K, the predictive mechanism resolves the gap in $X+t_{bft}$ time. In contrast, the reactive recovery time scales linearly with the drift (T), meaning larger gaps become exponentially harder to close as the faster BG continues to move forward. Second, T_{rec_reac} is inversely proportional to t_{slow}. If the straggler's local processing speed drops, its recovery time increases, creating a Recursive Failure Loop. PredictiveSkip breaks this loop by using the cross layer metadata to pull the straggler forward to match the fast BG's state. While T_{rec} defines the efficiency of the recovery mechanism, the global liveness of the protocol depends on whether this recovery can be finalized before the fast BG exhausts the remaining pipeline depth. To ensure the system remains fluid where the fast BGs never pauses processing cycles due to straggler, the recovery time must be strictly less than the Time-to-Stall (T_{stall}) where:

$$T_{stall} = (K - T) \cdot t_{fast} \tag{3.12}$$

In PredictiveSkip, if we substitute the recovery time for the predictive model ($T_{rec_pred} = t_{bft} + X$), the fluidity condition becomes:

$$(K - T) \cdot t_{fast} > t_{bft} + X \tag{3.13}$$

Solving for the ingestion window t_{fast}, we derive the minimum pipelined phase duration required for stability:

$$t_{fast} > \frac{t_{bft} + X}{K - T} \tag{3.14}$$

This result reveals a critical architectural advantage where the stability of Polaris is linearly dependent on the pipeline depth K. By increasing the execution window, the system can tolerate higher WAN latencies (X) or larger drifts (T) without inducing a stall. The headroom provided by K acts as a buffer that effectively facilitates for the $O(1)$ recovery to complete. In the reactive OneSkip model, recovery is not a single round but an incremental process. To prevent a stall, the drain rate of the skip mechanism must exceed the fill rate of the fast-path ingestion. Applying the reactive recovery time to the fluidity condition,

we observe a much more restrictive bound:

$$\begin{aligned} \frac{1}{t_{fast}} &> \frac{\frac{2t_{slow}}{2t_{fast}+X} - \frac{t_{bft}+X}{t_{fast}}}{t_{bft}+X} \\ t_{fast} &> \frac{X(t_{bft}+X)}{t_{slow}-2(t_{bft}+X)} \end{aligned} \tag{3.15}$$

The stability of the inequality depends on the sign of the coefficient $[t_{slow} - 2(t_{bft} + X)]$. This leads to the definition of the WAN Latency Threshold (X^*):

$$X^* = \frac{t_{slow}}{2} - t_{bft} \tag{3.16}$$

Stable Region ($X < X^*$): If $t_{slow} > 2(t_{bft} + X)$, the system can recover by adjusting the ingestion rate t_{fast}. Breakdown Region ($X \geq X^*$): If $t_{slow} \leq 2(t_{bft} + X)$, the denominator becomes non-positive. In this state, the Fluidity Condition fails and the reactive skip can no longer prevent pipeline from stall at the faster BG. In the predictive model, any increase in WAN latency X can be compensated for by a corresponding increase in pipeline depth K. On the other hand, In the reactive model, once X exceeds X^*, the system enters a state of irrecoverable drift. No amount of pipeline depth K can prevent a stall; K only serves to delay the inevitable halt by $(K - T)t_{fast}$ time.

3.5 Dynamic Pipeline

In planetary-scale distributed systems, the pipeline depth (N)[5] is a critical performance parameter. However, static configurations represent a fundamental performance bottleneck. If N is too shallow, the system cannot fill the BDP of cross-continental links between BGs, leading to network starvation. If N is too deep, the system suffers from buffer bloat of BFT results queuing behind network congestion, causing latency to spike exponentially without any gain in throughput. Furthermore, the variance in transaction load across geographically distant BGs creates regional drift, necessitating a pipeline that can dynamically expand to accommodate this skew. To navigate this, Polaris employs a state-aware control loop that treats pipeline depth as a dynamic equilibrium between physical network

[5]The number of concurrent consensus cycles in progress.

capacity and stochastic regional load skew. Each BG in Polaris deployment autonomously calculates the target pipeline depth (N_{new}) by combining global network telemetry with regional progress metadata. The depth is governed by the following formula:

$$N_{new} = \underbrace{N_{old} \cdot \frac{RTT_{ref}}{RTT_{obs}}}_{\text{Congestion Control (Part A)}} + \underbrace{(cycle_{fast} - cycle_{slow})}_{\text{Skew Resilience (Part B)}} \tag{3.17}$$

where:

N_{old}: current pipeline depth (execution window size)

RTT_{ref}: reference RTT calibrated to target bandwidth at 95% utilization

RTT_{obs}: observed round-trip time for the current cycle

$(cycle_{fast} - cycle_{slow})$: quorum-certified cycle gap between fastest and slowest BG

The formula contains two parts addressing orthogonal concerns. The multiplicative term governs inter-BG network condition. The additive term governs execution fairness across BGs. In congestion control, when the (RTT_{obs}) changes due to network-layer congestion, Part A contracts or expands the pipeline base. This prevents the BG from ingesting cycles faster than the degraded network can finalize them or underfilling cross-continental links between BGs respectively, keeping the system at the knee of the performance curve. While the skew resilience part addresses the condition where a BG lags due to local processing variance[6], the drift $\Delta = (cycle_{fast} - cycle_{slow})$ increases. Part B expands the window to incorporate this drift, providing the necessary headroom for the PredictiveSkip to trigger without stalling the faster BGs.

The BG primary operates on a five-cycle sampling epoch to dampen the effects of transient network micro-jitter. Within each cycle, the leader aggregates the Phase 3 (RTT) from all correct Envoys to compute a mean observed latency, R_{obs}. Simultaneously, the regional cycle-gap is derived from the $BFTCompletionProof$ metadata. Unlike the continuous averaging of RTT, the cycle-gap is sampled from every fifth $BFTCompletionProof$. In the event of a missing proof from a BG, the system treats the most recently executed cycle as a proxy for that BG's Phase 1 progress.

Next, we will examine the behaviour of the dynamic pipeline calculation under different conditions:

Case A Network Condition Change In the first case we analyze the dynamic pipeline optimization under variable network conditions while keeping the load drift across the

[6]It can be heterogenous load across the BGs.

BGs zero. The initial setup of the pipeline depth N and reference RTT RTT_{ref} is based on the base bandwidth[7] so that the overall performance of the protocol approaches close to the knee point. And to maintain this condition throughout under different network conditions the pipeline depth controller should satisfy two conditions: queue length of outstanding Phase 2 result (Q) should be zero and the link between the BG should not be underutilized. Since in this case the pipeline depth change is triggered due to network condition and not due to load drift we have $\Delta_{cycle} = 0$. Now to ensure that the pipeline operates at the optimal throughput-latency knee Part A of the formula functions as a State-Aware Regulator grounded in Little's Law ($L = \lambda W$). In the context of Polaris:

L (Items in system) represents the Pipeline Depth (N)

λ (Arrival rate) represents the Throughput (cycles/sec)

W (Wait time) represents the Observation RTT (RTT_{obs})

When network congestion occurs, RTT_{obs} increases. According to Little's Law, if the pipeline depth (N) remains static while the wait time (W) increases, the system must either decrease its arrival rate (λ) or allow an unmanaged queue to form. By proactively shrinking N as RTT_{obs} rises, Part A enforces a Zero-Queue Invariant. This ensures that the system does not ingest new transactions faster than the congested or degraded network can egress them, effectively preventing the exponential latency spikes characteristic of buffer bloat.

Conversely, when network conditions improve or RTT_{obs} approaches the reference floor (RTT_{ref}) or goes further down the reference RTT, the ratio-based formula allows N to expand. This ensures that the system maintains enough in-flight cycles to saturate the BDP of the inter-BG links. By ensuring the pipeline remains deep enough to ensure that the execution window always matches the physical capacity of the inter-BG links, Polaris prevents network starvation where correct BGs halt cycle progression not due to consensus failure but due to an artificially shallow execution window.

Case B Load Variance and Straggler Protection In the second case, we analyze the dynamic pipeline optimization under conditions of Regional Load Variance, where the underlying network is unchanged ($RTT_{obs} \approx RTT_{ref}$) but BGs exhibit drift in the pipeline utilization due to processing delays or variable load. In a traditional static pipeline, a performance bottleneck occurs whenever a BG lags behind the majority. This lag can be absolute, caused by local resource contention such as CPU spikes or disk I/O bottlenecks, or relative, where a BG is perceived to be lagging simply because other regions are processing a sudden surge in load more rapidly. Regardless of the cause, the Fast-Path majority[8] will

[7] In implementation that base bandwidth is 250 mbps.

[8] BGs with higher loads or better transaction processing capabilities.

eventually consume all available pipeline slots. Once the drift (T) between the fastest and slowest BG equals the static depth (N), the entire system hits a hard limit and must halt ingestion. This creates a Head-of-Line (HoL) Blocking effect, where the global throughput of the entire planet-scale network is strictly capped by the performance of its slowest regional member.

To prevent this stall and binding the overall protocol's performance with the slowest BG, Part B of the Polaris formula introduces an Additive Buffer based on the observed cycle-gap ($\Delta_{cycle} = cycle_{fast} - cycle_{slow}$). While Part A regulates the base of the pipeline based on the underlying inter-BG network links, Part B regulates its elasticity. By adding the drift directly to the depth, Polaris satisfies the Fluidity Condition. This maintains a constant drift buffer, where the number of available slots for the faster BGs remains approximately equal to the network's BDP capacity (N):

$$(N_{throttled} + \Delta_{cycle}) - T \approx N_{throttled} \tag{3.18}$$

This skew resilience term ensures that the majority of BGs can continue processing at peak throughput while the PredictiveSkip mechanism works in the background to coordinate a recovery jump for the straggler. Without this additive term, the fast BGs would hit a pipeline stall before the skip-cycle could even be proposed, leading to the throughput drops seen in static pipeline based protocols.

When Cases A and B occur simultaneously if they are aligned then the protocol has a definitive direction for pipeline to expand or shrink but it becomes interesting when both parts of the controller signal different outcomes for the pipeline for instance a regional link failure that both increases RTT and slows down a BG in this case the formula finds a Dynamic Equilibrium. Part A contracts the window to save the system from the latency increase of buffer bloat, while Part B expands it just enough to prevent a stall from the straggler.

After analysis of dynamic pipeline design one natural question that follows is how this design handles the pipeline contraction safely. In Polaris, this is addressed by treating the pipeline not as a storage buffer, but as a speculative execution window. Rather than managing the capacity of a queue, the window defines the maximum allowable concurrency of active consensus cycles. When the controller dictates a reduction in pipeline depth, the primary node simply suspends the ingestion of new cycles. Because the system does not function as a buffer, there is no requirement for explicit state draining of the pipeline. In-flight cycles are permitted to reach finality normally, while the issuance of new cycles is deferred until the number of active-unfinalized cycles falls within the reduced depth.

3.6 Chained Finality

Within deterministic protocols[9], finality semantics diverge considerably in how and when they are exposed to clients. Single-slot protocols such as PBFT [7] commit one batch at a time, providing immediate finality upon quorum acknowledgment but limiting throughput to one consensus round-trip per batch.

Pipelined protocols such as HotStuff [58] amortize this cost by overlapping multiple consensus instances but introduce a finality lag: due to its three-chain commit rule, a block proposed in view v is only committed once two additional consecutive views complete successfully. Consequently, clients must wait for at least two subsequent rounds before a transaction achieves finality. Parallel leader protocols such as Mir-BFT [51] extend throughput by assigning independent PBFT instances to multiple concurrent leaders, but require all in-flight instances within the watermark window to be reconciled before the global sequence can advance, tying finality to the watermark bound. DAG-based protocols such as Bullshark [50] derive a total order by locally interpreting a directed acyclic graph of certified proposals; finality is achieved once a transaction's vertex is causally ordered by a committed anchor, which typically requires two DAG rounds.

In geo-distributed hierarchical protocols, finality decomposes into two scopes: local finality, where a transaction is committed within a single BG, and global finality, where that commitment is acknowledged across all participating BGs. This two-tier scope introduces a layer of complexity absent in flat-network protocols. In RCanopus, global finality for cycle C requires every BG to collect BFT results from all other BGs before advancing. Under a pipeline of depth K, this forces the system to wait for $C + K$ cycles to confirm cycle C. This tight coupling induces Head-of-Line (HoL) Blocking, where a single slow BG delays the global finality horizon for all participants (an architectural bottleneck that scales linearly with pipeline depth).

Polaris addresses this through a relaxed global ordering model that reduces the finality requirement from $C + K$ to $C + 1$. Rather than enforcing strict sequential arrival for BFTCompletionProofs (the inter-BG coordination artifacts), Polaris enforces ordering on the underlying BFT responses. When a BG receives results for cycles $C' > C - 1$ out of order, Polaris buffers these results locally rather than stalling the coordination layer. As soon as the result for cycle $C - 1$ arrives, the buffer is immediately flushed, and all subsequent cycles are replicated and finalized in order. This ensures the global finality baseline advances as soon as the immediate predecessor cycle is confirmed, decoupling finality from the total pipeline depth.

[9] Consensus models that provide deterministic transaction finality.

A natural question is whether the $C+1$ requirement can be relaxed. We consider two candidate relaxations and show that both are insufficient.

Strawman 1: Finalize on Local Finality and Global BFTCompletionProof. The strongest relaxation would allow a BG to declare global finality for cycle C upon committing it local after Phase 2 and receiving BFTCompletionProofs from all other BGs, without waiting for the actual BFT responses. A BFTCompletionProof carries a quorum certificate that establishes the relative position and commitment of a remote BG's BFT result in the global order, but it does not carry the underlying transactions themselves. A BG that has received only the BFTCompletionProof for cycle C from a remote BG knows that a valid BFT result exists and has been committed, but cannot reconstruct the transaction list or verify the relative ordering of individual transactions within that BFT result. Finalizing cycle C without this information would violate the completeness requirement of finality as clients could not retrieve or execute their transactions even though the system has declared them final.

Strawman 2: Finalize on receiving BFT result of cycle C A weaker relaxation would allow a BG to declare global finality for cycle C as soon as it has collected the full BFT results (BFTResp) from all other BGs for that cycle. Now this semantic can guarantee the finalization of that cycle for the concerned BG but cannot guarantee the same for all the participating BGs. This fails under the following partition scenario. Consider BG A which has successfully collected BFT results from all BGs for cycle C and declares it globally final. Meanwhile BG B, due to a network partition, has sent its own BFT result to other BGs but has not yet received BFT results from some remote BGs for cycle C. From BG B's perspective, cycle C is incomplete. This creates a split finality condition where cycle C is globally final at BG A but not at BG B, violating the uniform agreement property that all correct BGs must reach the same finality decision for the same cycle.

Both strawmen fail because they attempt to finalize cycle C without a globally consistent anchor. The $C+1$ requirement resolves this by requiring that cycle $C-1$ has been fully confirmed across all BGs before cycle C can be finalized. Since $C-1$ serves as a common prefix that all correct BGs have agreed upon, it provides a consistent global baseline from which all BGs advance in lockstep Phase 3, ensuring consistent transaction finality across all the BGs regardless of transient network partitions.

Chapter 4

Safety And Liveness Analysis

In this chapter, we analyze the safety and liveness properties of Polaris and proposed cross-layer optimizations. We define the system output, establish the formal safety and liveness requirements, and prove that Polaris maintains established safety and liveness requirements despite the introduction of chained-finality, SkipCycles and a dynamic pipeline. We begin by establishing the foundational assumptions and definitions that underpin our theoretical framework.

Assumption 4.1 (Static System Configuration)**.**
We assume that the system configuration remains fixed throughout execution. Specifically, the following components are static: the set of participating nodes, the designated monitors, the appointed representatives, the constituent BG instances, and the emulators.

Next, we formalize the failure scenarios of the individual hierarchical components.

Definition 4.1 (Superleaf (SL) Status)**.**
For any Superleaf comprising N_{sl} replicas, its operational status is defined as follows:

- Failed SL: An SL is considered failed if it contains any Byzantine replica or more than $\lfloor (N_{sl} - 1)/2 \rfloor$ crashed replicas.
- Correct SL: An SL is considered correct if it is not failed.

Definition 4.2 (Byzantine Group (BG) Status)**.**
For any BG comprising N_{bg} Superleafs, its operational status is defined by the failure threshold of its internal BFT protocol:

- Failed BG: A BG is considered failed if more than $f = \lfloor (N_{bg} - 1)/3 \rfloor$[1] of its constituent Superleafs are failed.
- Correct BG: A BG is considered correct if it is not failed.

Assumption 4.2 (Individual BG Correctness)**.**
We assume that each BG_i independently satisfies both safety and liveness properties. In practical terms, this requires that the operating conditions of each BG's underlying BFT protocol remain within acceptable parameters, including assumptions on network synchrony and bounds on SL failures.

Assumption 4.3 (Partially-synchronous inter-SL communication)**.**
We assume the network is partially synchronous. There exists an unknown Global Stabilization Time (GST) and a known bound Δ. For any time $t \geq GST$, a message sent at time t is delivered by time $t + \Delta$.

Following this, we proceed to define the output of the protocol for cycle C and establish the fundamental safety properties of Polaris.

Definition 4.3 (Polaris Output)**.**
In a Polaris deployment consisting of z Byzantine Groups $(BG_1, BG_2, ..., BG_z)$, the Polaris Output (o_C) for a specific cycle number C is defined as an ordered, immutable collection of transactions derived from the non-nil results of all participating BGs returned by the procedure $getCycleResults(C)$ to a correct client.

We further define the commit and finality conditions for a cycle.

Definition 4.4 (Commit (BG Persistence))**.**
A cycle C is said to be committed within a Byzantine Group BG_i if there exists a valid Quorum Certificate $QC_{i,C}$.

Definition 4.5 (Finality (Client-Side Immutability))**.**
A cycle C achieves finality or is finalized if and only if a correct client p can verify a complete set of proofs $\mathcal{P}_C$ that "lock" the state across all z Byzantine Groups. Finality is reached when:

- Completeness: The client receives a response $resp$ (refer Definition 4.3) containing exactly one valid result (either a $BFTResult$ or a $BFTCompletionProof$) for every $BG_i \in \{1, \ldots, z\}$.

[1] Based on the current underlying BFT protocol for Polaris.

- Forward-Link Dependency ($C+1$): There exists a proof $m \in resp.cycleProof$ such that $CoversCycle(m, C+1)$ (Here we define $CoversCycle(m, C+1)$ as the cryptographic verification of a Quorum Certificate for height $C+1$) is true refer Algorithm 1 line 9. This indicates that the network has reached a consensus threshold on the subsequent state, making the transactions in cycle C cryptographically immutable.

Algorithm 1: Client Method for Strict Ordering Consensus Fetch

```
1  Function getCycleResults(C) is
2      if C > 0 ∧ ¬isCommitted(C − 1) then
3          return nil                                   // dependency check
4      foreach SL ∈ 𝒮𝒢 do
5          n ← randNode(SL)
6          resp ← Request(n, ClientFetchMsg(id, C))[a]
7          if C ∉ resp ∨ size(resp) ≠ |𝒮𝒢| then
8              continue
           ▷ Verify Proofs and Signatures
9          if ∃m ∈ resp.cycleProof | ¬CoversCycle(m, C + 1) ∨ ¬Verify(m) then
10             continue
11         foreach b ∈ resp.result do
12             K ← GetPublicKey(b.M.b)
               ▷ Verify based on Cycle State
13             if b.M.lc < C < b.M.cc then
14                 if ¬Verify(b.M, K) ∨ |b.M.R| > 0 then
15                     continue at line 4
16             else if C = b.M.cc then
17                 if ¬Verify(CalcDigest(b.M), K) then
18                     continue at line 4
19                 ReqSentOut ← ReqSentOut \
                     {c | C ∈ b.M.R ∧ c.id = id}
                   ▷ Remove client request c for cycle C
20             else
21                 continue
22         markCommitted(C)
23         return resp.result
24     return nil
```

[a] ClientFetchMsg includes $\{clientId, nonce, cycleRange\}$.

Definition 4.6 (Polaris Safety Properties)**.**
A protocol execution σ is safe if for every cycle C, the output O_C (as defined in Definition 4.3) satisfies the following formal requirements:

- Validity: If a correct client accepts a valid output O_v for cycle C, then O_v contains exactly the set of transactions committed by each Byzantine Group (BG) in cycle C
- Agreement: If any two correct clients, p and q, accept outputs O_p and O_q for the same cycle C, then $O_p = O_q$.

Algorithm 2: Intra-BG Block Verification and Ordering

```
1 Function VerifyAndSortSL(slReq, bgId, C) is
2     if ¬ IsMemberOf(slReq.slId, bgId) then
3         return fatal    ▷ SL membership validation failed
4     verifiedBlocks ← ∅
5     foreach blk ∈ slReq.blocks do
6         if SHA256(blk.transactions) ≠ blk.orderingId then
7             return fatal    ▷ Hash mismatch: tampered block content
8         verifiedBlocks ← verifiedBlocks ∪ {blk}
9     return DeterministicSort(verifiedBlocks, key = λ blk ⇒ blk.orderingId)
```

4.1 Safety Proofs of Polaris

To establish the safety of Polaris, we must demonstrate that the hierarchical coordination of BGs preserves global consistency despite the presence of stragglers or malicious actors[2]. The following lemmas and theorems bridge the gap between our system assumptions and the formal security properties defined in Definition 4.6. We first establish the deterministic ordering and reliable inter-BG sharing of the protocol through two core lemmas.

Lemma 4.1.
In a Polaris deployment D with z Byzantine Groups, the merge operations at every hierarchical phase are deterministic. Given an identical set of inputs, all non-faulty replicas and clients will produce a bit-wise identical global transaction order.

[2]Under the prescribed limit of faults to uphold Assumption 4.2.

Proof. Determinism is enforced through a fixed, hash-based sorting hierarchy at three distinct architectural levels. In Phase 1, replicas within each SL receive client requests and form ordered batches of these transactions called transaction blocks. Each transaction block is assigned an ordering ID as the hash of its contents. Servers exchange these blocks to the monitor of the SL and monitors sort them by their hash values to create the local ordering, which monitors then encapsulate into $SLReqMsg$[3] structures. In Phase 2, BFT leaders within each BG collect $SLReqMsg$ structures from all SLs monitors (Algorithm 2 line 8) and sort them by hash to form the BG-level ordering (Algorithm 2 line 9), which is agreed upon via SBFT and encapsulated in $BGRespMsg$[4] structures with quorum certificates. In Phase 3, representatives collect $BGRespMsg$ structures from all BGs and sort them by embedded random numbers (which are themselves hashed and part of the signed BFT output) to construct the global ordering refer Algorithm 3 line 16. If a Byzantine actor (e.g., a malicious monitor or envoy) attempts to reorder transactions or substitute content at any level, the resulting root hash of the batch will change. Since the Quorum Certificate (QC) is cryptographically bound to the original hash, any modification will invalidate the associated hashes and signatures, leading to rejection by non-faulty clients. □

[3] A set of ordered transaction blocks $\{blk_1, blk_2, ...\}$ with Phase 1 quorum certificate QC_{P1} from a Superleaf.

[4] Contains $\{cycleId, bgId, transactions, QC\}$, where QC is the quorum certificate for the batch.

Algorithm 3: System Output Validity Check and Global Merge

Function *ValidateAndMerge(*$BGRespMsg,\ C$*)* **is**
  $seenBGs \leftarrow \emptyset$
  foreach $R \in BGRespMsg$ **do**
    if $\neg$ *ValidateSignature*$(R, sysConfig)$ **then**
      return fatal ▷ Invalid signatures or malformed message
    if R *is BFTCompletionProof* **then**
      if $R.lc \geq C \vee R.cc \leq C$ **then**
        return fatal ▷ Cycle C not within skip window $[lc, cc]$
    else if $R.cc \neq C$ **then**
      return fatal ▷ BG ordering window mismatch
    if $R.bgId \in seenBGs$ **then**
      return fatal ▷ Duplicate response from BG $R.bgId$
    $seenBGs \leftarrow seenBGs \cup \{R.bgId\}$
  if $seenBGs \neq \mathcal{BG}$ **then**
    return fatal ▷ Missing results from one or more BGs
  $sortedR \leftarrow \text{DeterministicSort}(BGRespMsg, \text{key} = \lambda r \Rightarrow r.id)$
  $globalTxList \leftarrow [\text{flatten}(R.payload.R) \mid R \in sortedR, R.payload \neq \perp]$
  return $globalTxList$ ▷ Final ordered transaction sequence for cycle C

Lemma 4.2.
For any two Byzantine Groups BG_i and BG_j, assuming partial synchrony with a bounded message delay Δ, every correct replica in BG_j will eventually receive the certified response M_{C,BG_i} from BG_i for cycle C.

Proof. The inter-BG message exchange in Polaris operates through a two-stage request-fetch protocol with built-in redundancy to tolerate Byzantine failures on both the requester and responder sides. Without loss of generality, we focus on BG_j fetching results from BG_i for cycle C; the symmetric case follows by an identical argument. The representative of the primary envoy SL in BG_j sends π_{C,BG_j}, attesting that Phase 2 for cycle C has completed, to a single emulator of BG_i. Upon receiving π_{C,BG_j}, the contacted emulator responds with $M_{BG_i,C}$, containing BG_i's ordered transaction list and quorum certificate for cycle C. Once fetched, $M_{BG_i,C}$ is replicated throughout BG_j using the intra-BG replication mechanism. Since BG_j is non-faulty by Assumption 4.2, all correct replicas in BG_j eventually receive copies of $M_{BG_i,C}$. We now consider the two failure scenarios that could prevent successful message delivery.

Case A: Faulty Primary Envoy If the designated envoy in BG_j fails or provides nil results, the protocol falls back to f backup envoys. Since a normal BG contains at most f Byzantine SLs, at least one correct envoy is guaranteed to complete the fetch. Also, Byzantine envoys cannot forge results due to the $2f + 1$ signature requirement in quorum certificates.

Case B: Faulty Emulator If the contacted emulator at the remote BG_i is Byzantine or unresponsive, the representative contacts f additional emulators across distinct SLs. The $f + 1$ SL contact ensures that at least one honest SL will provide a valid, signed response. Because the fetch logic is f-resilient on both the requester side (envoys) and the responder side (emulators), the certified state of BG_i is guaranteed to reach BG_j as long as f is not exceeded (Assumption 4.2).

□

Algorithm 4: SBFT Preprepare Validation with Skip-Cycle

```
1  Function ValidatePreprepare(ppm) is
2      if ∃tx ∈ ppm.R | ¬VerifySig(tx) then
3          return false    ▷ Invalid client signature
4      cc ← ppm.cc
5      cc_old ← last(this.log).cc
6      if cc ≤ 0 ∨ cc ≤ cc_old then
7          return false    ▷ Enforce monotonicity
8      K ← getSystemPipelineDepth()
9      if cc − cc_old − 1 > K then
10         return false    ▷ Skip exceeds K
11     if cc > MAX_CYCLE then
12         return false    ▷ Prevent overflow
13     this.last_phase1_cycle ← cc
14     return true
```

Building upon these lemmas, we now prove that Polaris satisfies the fundamental safety requirements as defined in Definition 4.6.

Theorem 4.3.
For any cycle C, let O_C be output of Polaris (Definition 4.1). Then O_C is valid, meaning it contains exactly the set of transactions committed by each normal BG in cycle C.

Proof. Suppose, for the sake of contradiction, that a correct client accepts an output O_C that contains a transaction batch $T'_{i,C}$ for BG_i which differs from the batch actually com-

mitted by BG_i. Using Lemma 4.1, we know any change in transaction order $T'_{i,C}$ must result in a different cryptographic hash. And in Algorithm 1 (line 17) the client calculates the digest over the received result and verifies the threshold signature. For the client to accept $T'_{i,C}$ while the network has committed $T_{i,C}$, there must exist two valid QCs for the same cycle C in BG_i: QC_C (for the correct batch) and QC'_C (for the adversarial batch). And each QC requires signatures from a quorum of $n-f$ (i.e., $2f+1$) Superleaf (SL) monitors. In a non-Byzantine BG (Assumption 4.2) where $n \geq 3f+1$, any two such quorums must intersect in at least $f+1$ nodes. Since there are at most f Byzantine SLs, at least one correct (honest) monitor M_{honest} exists in the intersection. To form both certificates, M_{honest} would have had to sign two different Preprepare messages for the same cycle C. However, as defined in Algorithm 4 (line 6), an honest monitor maintains a strict state and refuses to sign a message if the cycle number has already been used or does not follow the monotonic $cc > cc_{old}$ rule. Thus, the existence of QC'_C would require M_{honest} to violate the safety of the underlying BFT protocol or require the adversary to control $> f$ SLs, both of which contradict our system assumptions. Thus, the accepted batch $T'_{i,C}$ must be identical to the committed batch $T_{i,C}$.

□

Theorem 4.4.
If any two correct clients, p and q, or the same correct client at different points in time, accept valid outputs O_p and O_q for the same cycle C, then $O_p = O_q$.

Proof. Suppose, for the sake of contradiction, that $O_p \neq O_q$ for cycle C. This divergence must originate from one of two sources: either a single BG_i successfully committed two different, certified results for cycle C, or the global merge of identical BG results produced different sequences. First, we consider the possibility of conflicting BG results. As established in Theorem 4.3, this is impossible because it would require at least one honest monitor to sign two different proposals for the same cycle, violating the strict monotonicity and state management rules of the underlying BFT protocol. Thus, the set of inputs from all BGs for cycle C must be identical for both clients.

Second, we consider the possibility that the global merge operation produced a different transaction order despite having identical inputs. However, this scenario results in a Deterministic Merge Contradiction because according to Lemma 4.1, the merge operations at every hierarchical phase are strictly deterministic. The protocol enforces a bit-wise identical global order by sorting transaction blocks by hash at the SL level, sorting SLReqMsg structures by hash at the BG level, and merging all BFTRespMsg by hash at the global level. As any modification to the transaction order would necessarily change the associated cryptographic hashes and invalidate the threshold signatures verified by the clients, a

different order $O_p \neq O_q$ is impossible under our deterministic merge rules. Since both the possibility of conflicting BG results and the possibility of non-deterministic merging lead to contradictions of established protocol invariants, the initial assumption $O_p \neq O_q$ must be false. Therefore, $O_p = O_q$.

□

Theorems 4.3 and 4.4 establish that Polaris satisfies the fundamental requirements outlined in Definition 4.6, fulfilling the properties of Validity and Agreement, respectively.

4.1.1 Safety Proof of SkipCycle Optimization

The SkipCycle optimization is designed to be safety-preserving while significantly enhancing liveness by allowing straggler BGs to synchronize with the global pipeline. While the mechanism permits a BG to advance its cycle count without processing new transactions, every skip remains strictly bound by the BG's BFT quorum agreement. Safety is preserved because the "null" nature of a skipped cycle is cryptographically attested within the Quorum Certificate (QC), preventing an adversary from perpetual skipping.

Theorem 4.5.
The introduction of the SkipCycle extension preserves the properties defined in Definition 4.6 and avoids the trivial attack of perpetual skipping.

Proof. Suppose, for the sake of contradiction, that the SkipCycle extension leads to a violation safety requirement of Definition 4.6 and allows an adversary to perform perpetual skips. This implies that for a specific cycle C, a correct client accepts an output O_C where BG_i has two conflicting committed states: one as an ordinary cycle containing transactions $T_{i,C}$, and one as a skipped cycle containing no transactions. This scenario implies that BG_i has successfully formed two conflicting QCs for the same cycle C. However, this directly contradicts Theorem 4.3, which proves that a normal BG cannot commit more than one valid result for a given cycle number. As established in Theorem 4.3, any two quorums in a normal BG must intersect at at least one honest monitor, M_{honest}. Because M_{honest} maintains a monotonic local state and strictly refuses to sign any transaction batch or skip proof for a cycle number that has already been utilized, the network cannot generate the threshold of $2f + 1$ signatures required for both results. Furthermore, the decision for the skip is linked with the availability of a valid QC this contradicts our initial assumption where the introduction of SkipCycle allow adversary to perform perpetual skips. Since the

Algorithm 5: Message Handling and Strict Ordering Enforcement

Function *HandleIncoming(msg)* **is**
  switch *msg.type* **do**
    case *STATE_REQUEST* **do**
      ▷ Server Acting as an Emulator
      if $\neg$*CheckStateRequest*(*msg*) **then**
        return fatal
      resp $\leftarrow$ GetOrWaitBGStateResponse($msg.cycleId$)
      if *resp* = *nil* **then**
        StartCycle(*msg*) ▷ Register pending request & trigger SkipCycle
      $addr \leftarrow serverTCPAddrMap[msg.senderAddr]$
      $tcpManager.Send(addr, \text{STATE_RESPONSE}, resp)$
    case *STATE_RESPONSE* **do**
      ▷ Server Acting as an Envoy representative
      if $\neg$*CheckQC*(*msg*) **then**
        return fatal
      $strictOrderBuffer[msg.bgId][msg.cycleId] \leftarrow msg$
      $C \leftarrow msg.cycleId$
      while
       $(C = 0 \ \vee \ isGloballyCommitted(C-1)) \ \wedge \ \forall bg \in \mathcal{BG}, buffer[bg][C] \neq \perp$
       do
        $allResp \leftarrow \{\text{GetOwnBGStateResponse}(C)\} \cup \{buffer[bg][C] \mid bg \in \mathcal{BG} \setminus \{\text{localBG}\}\}$
        if $\exists r \in allResp \mid \neg VerifyQuorumCert(r)$ **then**
          return fatal ▷ Safety violation detected
        $globalOrder \leftarrow$ DeterministicSort($allResp$) ▷ Prop. IV.1
        if *IsLeader*() **then**
          $raftLayer.CommitGlobalOrder(C, globalOrder)$
        else
          ForwardToMonitor($C, globalOrder$)
        MarkGloballyCommitted(C)
        EvictBuffer(C) ▷ Garbage collect cycle C
        $C \leftarrow C + 1$ ▷ Try next cycle in pipeline
    case *STATE_RESPONSE_TO_MONITOR* **do**
      if *IsLeader*() **then**
        $raftLayer.AddStateResponse(msg)$
      else
        ForwardToLeader(*msg*) ▷ Forward to current Monitor leader

existence of conflicting QCs is impossible, the SkipCycle extension maintains Validity and the core requirements of Definition 4.6.

□

4.1.2 Safety Proof of Chained Finality

The Chained-Finality property ensures that once a client observes a valid output for cycle C, the entire preceding history of the ledger is finalized (Definition 4.5).

Theorem 4.6.
For any cycle number C, if Polaris has produced a valid output O_C, then every Byzantine Group (BG_i) has already finalized (Definition 4.5) all cycles up to and including C.

Proof. Suppose, for the sake of contradiction, that a correct client accepts an output O_C (per Definition 4.3) but there exists at least one Byzantine Group BG_i that has not yet finalized one or more cycles in the range $[1, C]$. And According to Algorithm 1 line 9, a correct client only returns a non-nil response for cycle C after verifying a CycleProofMsg that covers cycle $C + 1$ for every $BG \in \mathcal{BG}$ (where $\mathcal{BG}$ is set of all BGs in the concerned Polaris deployment). And with a strict ordering precondition enforced in Phase 3 inter-BG sharing condition at Algorithm 5[5] line 17 which can't be satisfied unless C has been finalized across all the BGs and cascading the same for all the preceding cycles. Thus we encounter a contradiction and existence of a valid $C + 1$ proof across all BGs thus serves as an immutable cryptographic anchor, proving that cycle $[1, C]$ is fully finalized. □

4.1.3 Safety Analysis of Global Coordination Optimization

The safety of the inter-BG request-fetch protocol is anchored in the cryptographic immutability of the data and the redundancy of the f-fallback mechanism, this mechanism ensures that by querying $f+1$ SLs, a correct actor is mathematically guaranteed to interact with at least one honest monitor. Because this SL provides a cryptographically bounded state of BG_i for a particular cycle C, the f-fallback mechanism bridges the gap between

[5]The asynchronous timeout and retransmission mechanisms are managed by the underlying networking layer such that if a STATE_RESPONSE is not received within the designated Δ interval following the Global Stabilization Time (GST), or if the primary envoy fails to finalize a STATE_REQUEST within the prescribed timeout, the system invokes the f-fallback strategy by triggering f secondary envoys to independently initiate fetch operations from the remote emulator pool, ensuring that the certified Byzantine Group state is retrieved even in the presence of a faulty or slow primary coordinator.

regional consensus and global finality without introducing a single point of failure. Under this optimization, a single Primary Envoy or Emulator is initially contacted to minimize network overhead; however, safety is never compromised by this "optimistic" path because of two distinct protections. First, any attempt by a Byzantine actor to provide forged or tampered data is caught during the verification phase, as they cannot produce a valid QC without finding a cryptographic collision or compromising the threshold signature. Second, to prevent a "nil-result" or omission attack, the protocol utilizes an f-fallback strategy. Per Assumption 4.2 ($n \geq 3f+1$), by retrying the fetch across $f+1$ total SLs, the requester is guaranteed to interact with at least one honest monitor who possesses the certified state.

4.1.4 Safety Analysis of Dynamic Pipeline Optimization

The dynamic pipeline optimization allows simultaneous incomplete cycles outstanding at any point in time, but it does not redefine the safety boundary of Polaris. Instead, it adjusts the window of progress available to each BG while remaining strictly bound by the requirements of Definition 4.6. Thus, the introduction of this optimization is primarily a Liveness concern rather than a safety one.

4.2 Liveness Analysis of Polaris

Liveness in Polaris ensures that the system makes perpetual progress, specifically guaranteeing that client transactions are eventually processed and that cycle results are obtained as Polaris output as defined in Definition 4.3. This section formalizes these properties under the foundational assumptions of bounded failures (Assumption 4.2) and partial network synchrony (Assumption 4.3).

Definition 4.7.
Protocol execution σ is considered live if it satisfies the following three conditions:

- Transaction Termination: Every transaction submitted by a correct client eventually terminates with a success or failure reply.
- Output Termination: For any cycle number C, a call to the $getCycleResults(C)$ function eventually returns a non-nil output O_C.
- Eventual Inclusion: If a transaction tx is successfully submitted by a correct client, there eventually exists a cycle C such that $tx \in O_C$.

Theorem 4.7.
Polaris satisfies the properties of Transaction Termination, Output Termination, and Eventual Inclusion as specified in Definition 4.7

Proof. We analyze the system's progress in two phases defined by the network state. For all $t < GST$, the network may experience arbitrary delays or partitions, potentially causing protocol timers to expire; however, the quorum requirements $(2f + 1)$ ensure that safety is never violated during these intervals. Liveness is guaranteed for $t \geq GST$, as the known bound Δ ensures that all consensus phases and recovery mechanisms complete within a predictable window. Suppose, for the sake of contradiction, that the system fails to maintain liveness in the post-GST phase. This implies that either a client's $getCycleResults(C)$ call never returns or a submitted transaction tx is permanently excluded from the ledger. This leads to the following contradictions:

Case A Global Stall due to Straggler: If the client never receives O_C, it implies at least one BG has stalled permanently. However, each BG independently satisfies liveness under sufficient synchrony. Even if a BG is a "straggler" due to heterogeneous regional loads, the SkipCycle optimization allows it to synchronize by issuing a quorum-certified "null" result for that cycle. Since every BG is guaranteed to eventually produce either a transaction batch or a skip result, the client must eventually receive a response for every BG_i. Furthermore, the dynamic pipeline depth K adjusts to accommodate regional drift, ensuring faster BGs are not blocked by a static window.

Case B Result Reachability: In this case the BGs are not able to retrieve the Phase 2 result from other BGs, it contradicts the f-fallback strategy established in Lemma 4.2. Because the fetch protocol retries across $f+1$ total SLs, and at most f are failed, the client is guaranteed to interact with at least one honest monitor who possesses the certified state. Since the inter-BG network eventually restores sufficient synchrony after any partition, the BG will eventually successfully fetch the required z proofs.

Case C Fair Inclusion and View-Changes: At the SL level, if an SL leader ignores a submitted transaction, the client will broadcast tx to f other SLs, ensuring it reaches the honest monitors. At the BG level, the underlying BFT protocol utilizes a view-change mechanism that ensures a Byzantine leader who suppresses SLs transaction blocks is eventually replaced by a correct leader. Since cycle results are guaranteed to terminate (as proven in Case A), tx must eventually be proposed by an honest leader and appear in a finalized output O_C. □

4.2.1 Liveness Analysis of Dynamic Pipeline

The correctness of the dynamic pipeline calculation can be established by demonstrating that it prevents the global pipeline from stalling due to regional load drift. Assume, for the sake of contradiction, that a BG is permanently unable to propose a new cycle because it has hit a pipeline limit. This would imply that the gap between its current cycle and the global minimum has exceeded K. However, the dynamic formula directly contradicts this possibility. By including the group drift term $(\max - \min)$, the pipeline depth automatically expands to accommodate the exact skew currently present in the network. As long as at least $2f+1$ monitors are reachable to issue a BFTCompletionProof, the values of local pipeline utilization[6] will eventually be disseminated via the f-resilient fetch logic established in Lemma 4.2. Furthermore, the base depth D ensures that there is sufficient buffer to keep the network saturated while the first cycle in the window is still in flight. Because K is derived from quorum-certified proofs, every correct replica will eventually calculate a K that permits continued progress. This ensures that a faster BG is never forced to wait indefinitely for a straggler, provided the straggler eventually produces a skip or a commit result. Thus, the dynamic pipeline depth maintains global liveness.

4.2.2 Liveness Under Network Partition

Polaris is designed to prioritize consistency over availability during total network partitions. In the event of an intra-BG partition that prevents a $2f + 1$ quorum, the affected group will naturally stall to prevent state divergence until connectivity is restored. Similarly, if an inter-BG partition isolates one or more groups, the protocol stalls global finality (as a complete set of results from all z BGs is required for output) and fast-moving BGs will eventually hit the dynamic pipeline limit K. Liveness is automatically restored once the partition heals, at which point the f-resilient fetch logic and SkipCycle mechanism allow the isolated groups to rapidly synchronize and resume consensus.

[6] Phase 1 cycle numbers are employed in implementation due to lack of Phase 2 pipeline.

Chapter 5

Analytical Evaluation of Consensus Properties

In this chapter, we present a comprehensive analytical comparison of Polaris against state-of-the-art topology-aware BFT protocols, including Steward, GeoBFT, and LT-DBFT. We characterize the performance and security trade-offs from two aspects:

- Communication Complexity: This is defined as the total number of messages exchanged to reach a globally consistent ordered decision for a specific cycle (or batch of transactions). We distinguish between intra-region complexity (local coordination within a cluster) and inter-region complexity (global coordination across geographically distributed clusters).
- Fault Tolerance Thresholds: A comparison of how protocols manage under different failure scenarios. Unlike flat systems where a single threshold ($f < n_t/3$) applies globally, we analyze how failures are isolated within specific clusters and the impact of correlated regional disruptions on the system's global state.

5.1 Communication Complexity Analysis

To understand the scalability of Polaris, we must derive the communication complexity across its three phases and compare it to existing clustered BFT protocols. In traditional BFT systems, the message complexity is a function of the total number of replicas (n_t). In Polaris, the complexity is decoupled across hierarchical layers, focusing on the number of

participating Byzantine Groups and the number of nodes per cluster. Assume for a Polaris deployment D there are z BGs, where each BG is implemented with $n_{SL} = 3f_{SL} + 1$ SLs, and each such SL has $n = 2f + 1$ replicas.[1]

Table 5.1: Comparative Complexity and Architectural Characteristics †

Protocol	Local Comm.	Global Comm.	Decisions	Centralized
Steward	$O(zn_t^2)$	$O(z^2)$	1	Yes
GeoBFT	$O(zn_t^2)$	$O(z^2 n_t)$	z	No
LT-DBFT	$O(zn_t^2)$	$O(z^3)$	z	No
Polaris	$O(zn_t)$	$O(z^2)$	z	No

†These complexity bounds are derived for failure-free executions.

5.1.1 Intra-BG Communication Complexity (Local Communication)

Intra-BG consensus consists of Phase 1 CFT replication and Phase 2 BFT replication. In Phase 1, the $n - 1$ replicas in a Superleaf (SL) forward client transactions to the SL monitor. The monitor orders these transactions to create a Transaction Block (TB) and transmits the TB back to all $n - 1$ replicas. For a single Byzantine Group (BG) with n_{SL} Superleafs, the total communication for this intra-SL phase is $n_{SL}(2n - 2)$, which is $O(n \cdot n_{SL})$ or $O(n_t)$.

Next, the protocol commences Phase 2 BFT replication delegated to the n_{SL} monitors of the SLs. In the pre-prepare phase, the primary BG node sends a message to the other $n_{SL} - 1$ monitors in the group, resulting in $n_{SL} - 1$ communications. In the prepare phase, each monitor node verifies the pre-prepare message and sends a prepare message to the collector node. All n_{SL} nodes send their prepare messages to the collector resulting in n_{SL} communications. The collector then aggregates $2f_{SL} + 1$ prepare messages into a prepare certificate using threshold signatures and broadcasts this certificate to all n_{SL} nodes, resulting in another n_{SL} communications. In the commit phase, each node receives the prepare certificate and sends a commit message to the collector, resulting in n_{SL} communications. The collector aggregates $2f_{SL} + 1$ commit messages into a commit certificate and broadcasts it to all n nodes, resulting in n_{SL} communications.

[1]Overall a single BG contains $n_t = n_{SL} \times n$ nodes.

The final communication complexity for Phase 2 for a single BG is $(n_{SL}-1)+n_{SL}+n_{SL}+n_{SL}+n_{SL} = 5n_{SL}-1$, which is $O(n_{SL})$. Therefore, the total communication complexity for a single BG's Intra-BG consensus is $O(n) + O(n \cdot n_{SL})$, and the total protocol complexity across all groups is $O(z \cdot n_{SL}) + O(z \cdot n \cdot n_{SL})$ which can be further simplified to $O(z \cdot n_t)$.

5.1.2 Inter-BG Communication Complexity (Global Communication)

Inter-BG coordination phase shares the certified BFT results of each BG to establish a global transaction order. This phase uses a request-fetch protocol involving envoys and emulators. For a single BG_i, when its designated primary Envoy sends BFTCompletionProof to one Emulator in each of the other $z-1$ groups. Upon receiving a request, each Emulator responds with its group's signed transaction list and the corresponding Quorum Certificate (QC). This exchange results in $(z-1)$ request messages and $(z-1)$ response messages, totaling $2(z-1)$ communications for one BG to synchronize with the rest of the network.

Since every *BG* in the system must independently perform this fetch operation to maintain a consistent local state, the process is repeated z times. The total communication count for Phase 3 across all groups is $z[2(z-1)]$, which simplifies to $2z^2 - 2z$. In the case of an unresponsive primary Envoy or Emulator, the protocol retries the fetch through f fallback nodes, introducing a constant factor increase while maintaining the same scaling behavior.

The final communication complexity for the Global Coordination phase is $O(z^2)$. When combined with the Intra-BG phases, the total complexity for Polaris is $O(z^2) + O(zn_t)$. As depicted in Table 5.1, Polaris exhibits superior local communication compared to Steward, GeoBFT, and LT-DBFT for the same number of Byzantine Agreements. The two-phase intra-BG consensus allows Polaris to perform more decisions ($z \cdot n_{sl}$ compared to z or 1 for the others) by leveraging parallel transaction ingestion and processing through Phase 1 CFT. This design reduces BFT complexity by limiting the number of participating nodes in the BFT process to SL monitors and utilizing SBFT's inherent linear complexity.

In contrast, GeoBFT and LT-DBFT perform z Byzantine Agreements with $O(z \cdot n_t^2)$ complexity. While LT-DBFT attempts to limit local communication by forming a committee of c nodes, it employs a trust-score-based mechanism where replicas must pass a certain threshold to join the committee. Consequently, in a stable protocol execution, almost every node eventually participates in the BFT consensus, degrading its complexity. Finally,

Steward follows a similar pattern but makes only a single decision due to its centralized design where a single primary cluster is responsible for the Byzantine agreement.

Similarly, Polaris maintains a more efficient global communication profile compared to other topological-aware protocols. To begin with, GeoBFT's global coordination requires each primary to share results with primary and $\lfloor \frac{n_t-1}{3} \rfloor$ additional replicas in every other cluster, leading to $O(z^2 n_t)$ complexity. In contrast, Polaris leverages the Envoy-Emulator model to isolate inter-BG traffic to a single primary envoy per group, achieving a global complexity of $O(z^2)$. On the other hand, LT-DBFT's round-robin global sharing mechanism results in $O(z^3)$ complexity. Finally, Steward achieves the similar theoretical global complexity at $O(z^2)$ because of its centralized site coordination.

5.2 Fault Tolerance Analysis

In this section, we will analyze failure scenarios representative of geo-distributed deployments to assess the resilience of Polaris against its peers.

Table 5.2: Comparative Fault Tolerance Thresholds

Protocol	**Max BFT Faults**	**Max CFT Faults**	**Additional Requirement**
Steward	$\lfloor \frac{n_t-1}{3} \rfloor$	$\lfloor \frac{n_t-1}{3} \rfloor$	Re-election if a cycle fails
GeoBFT	$\lfloor \frac{n_t-1}{3} \rfloor$	$\lfloor \frac{n_t-1}{3} \rfloor$	None
LT-DBFT	$\lfloor \frac{n_t-1}{3} \rfloor$	$\lfloor \frac{n_t-1}{3} \rfloor$	Majority of primary site must be honest
Polaris	$\lfloor \frac{n_t-1}{3} \rfloor$	$\lfloor \frac{n_t}{n} \rfloor \cdot \lfloor \frac{n-1}{2} \rfloor$	Localized Byzantine failure isolation

5.2.1 Case 1: Normal Node Failure

Polaris uses Raft-based CFT replication. If an SL node crashes, the system remains operational provided a majority ($\lfloor n_{SL}/2 \rfloor + 1$) are functional. This CFT layer addresses real-world deployments where crash faults (power loss, disk failure, OS panic) occur far more frequently than Byzantine faults [45, 19, 20, 12]. In contrast, other topology-aware

protocols lack this localized CFT layer, which reduces their crash fault tolerance to the same threshold as f, as summarized in Table 5.2. As a result, any node failure in such protocols often escalates immediately to the BFT layer, potentially triggering[2] expensive view changes or signature re-collections across the entire cluster.

5.2.2 Case 2: SL Monitor Failure

In Phase 1, if an SL monitor fails, Polaris triggers a monitor re-election within that SL. During this period, the SL may be temporarily unable to participate in Phase 2 BFT; however, the remaining $2f_{SL} + 1$ honest monitors within the BG continue the consensus process. At the BG level, Polaris is designed to tolerate up to f_{SL} total SL failures. Because consensus is delegated to monitors in the BFT layer, any Byzantine behavior from a monitor is treated as an SL failure. Consequently, the loss or corruption of an entire SL is considered a single Byzantine failure within the $3f_{SL} + 1$ threshold of the BG. Non-Monitor replicas in each SL also track the status of the monitor in Phase 2 and on the detection of monitor failure trigger the leader election[3] to elect a new monitor.

In contrast, the equivalent failure in other protocols is treated as a standard node failure during the BFT phase. For Steward and GeoBFT, this is handled by the intrinsic Byzantine Fault Tolerance properties of their underlying BFT protocols. LT-DBFT operates differently: it selects a committee of c nodes from the cluster, where fault tolerance is defined as f_c (given $c \geq 3f_c + 1$). If a committee node exhibits Byzantine behavior (either through non-participation or malicious voting) its trust score is reduced according to the LT-DBFT trust model. This decrease in trust directly lowers the probability of the affected node being selected for future BFT committees.

5.2.3 Case 3: BG Primary Failure

During the BFT Phase, if the BG Primary fails, the surviving SL monitors detect the timeout and initiate a local BFT view change. This mechanism for handling Primary failure during the BFT replication phase is largely consistent with other topology-aware protocols which similarly rely on regional view changes to maintain liveness. For LTDBFT the selection of new primary is dependent on the trust score of the node.

[2]Based on the BFT replication mechanism employed in the concerned topological-aware protocols.
[3]Here Byzantine failure of Phase 2 translates to crash abstraction of Phase 1.

5.2.4 Case 4: Envoy Representative Failure

Global coordination in Polaris relies on Envoys to fetch the BA. To ensure the system doesn't stall during the inter-BG phase, we select $f + 1$ Envoy SLs, with one node per SL acting as a representative[4]. Within this group, one is designated as the primary representative. In this phase, the primary Byzantine risk is an act of omission (refusal to respond or share data). Because any shared transaction list must be backed by a valid Quorum Certificate (QC), any attempt to manipulate the data itself would be immediately detected and rejected. If a representative times out or sends a negative response, a new representative is elected. Simultaneously, f fallback SL representatives are triggered to fetch the result. Since we assume at most f Byzantine SLs, at least one honest representative is guaranteed to successfully retrieve the data. To further secure this process, Polaris maintains convergence logs for each SL. If a primary SL's results frequently diverge from the f fallback Envoys, its convergence score is penalized. This reduced score decreases its likelihood of being selected as a representative in future rounds, effectively self-cleaning the pool of global-coordinators. Furthermore, if a representative fails to provide any response, we continue the replacement process until a response is received, relying on the fact that an entire SL infrastructure is unlikely to crash simultaneously.

In the case of GeoBFT, the primary is responsible for sharing BA to $f + 1$ replicas in every other BG. If the primary behaves maliciously, the receiving clusters must trigger a view-change request back to the source cluster's $f+1$ replicas, leading to high cross-region communication overhead. Next in Steward, the protocol relies strictly on primary-to-primary interaction for global sequencing. If a primary behaves Byzantinely, the cycle cannot complete and on global timeout for the cycle it triggers a global view change across all participating clusters which is a significantly more expensive recovery process than Polaris's local representative swap. LT-DBFT utilizes a round-robin mechanism for primary-to-primary coordination. However, its safety is highly fragile: if more than one-third of the clusters have malicious primaries, the global order fails. This adds an extra layer of vulnerability on top of the standard BFT conditions that Polaris, Steward, and GeoBFT do not share.

[4]For environments with high transient node failure rates, Polaris can be configured to select multiple active representatives per SL. This avoids the latency penalty of timeouts and re-elections by allowing redundant fetch operations to proceed in parallel.

5.3 Case 5: Emulator Failure

The Emulator role, unique to Polaris, serves as the repository for regional consensus results during Phase 3. Because every transaction list served by an emulator must be accompanied by a valid QC, a malicious emulator cannot provide fraudulent data and any tampering would be immediately detected by the fetching Envoy during verification. The only viable Byzantine behavior for an emulator is suppression (omission or refusal to respond). To maintain liveness and avoid the high latency of repeated local retries, Polaris utilizes a parallel fetch strategy where if the primary Envoy representative does not receive a response from the designated Emulator, it does not waste cycles on local retries within that same potentially partitioned SL. Instead, it immediately broadcasts the fetch request to f other SLs within the remote BG. Since the protocol assumes at most f Byzantine SLs per group, at least one of these $f + 1$ total requests is guaranteed to reach an honest SL that holds the certified result.

Chapter 6

Implementation

This chapter covers the implementation of Polaris, detailing the component design, technology choices and engineering challenges encountered during development. We translated the theoretical design described in Chapter 3 into a working system that can be used as an emulation of coordination service for geo-scale deployments.

6.1 Implementation Overview

We implemented Polaris as a full featured prototype built on top of the RCanopus codebase. The system is primarily written in Go and consists of approximately 7600 lines of code. Continuing the design principle of RCanopus, Polaris architecture emphasizes a modular design where replication protocols at various layers are treated as independent modules and as long as a specific module satisfies the interface requirements and fault tolerance assumptions of its respective layer it can be integrated into the protocol. The major components of Polaris and their interactions are shown in Figure 6.1[1].

6.2 System Component Design

The transaction input queue serves as the entry point for client transactions into the Polaris system. Every server maintains a transaction input queue that stores incoming transactions

[1]Not all interactions are active on every replica, since behavior depends on the replica's assigned role. For example, communication with the BFT coordination layer is required only on monitor nodes and therefore does not occur on Raft follower replicas.

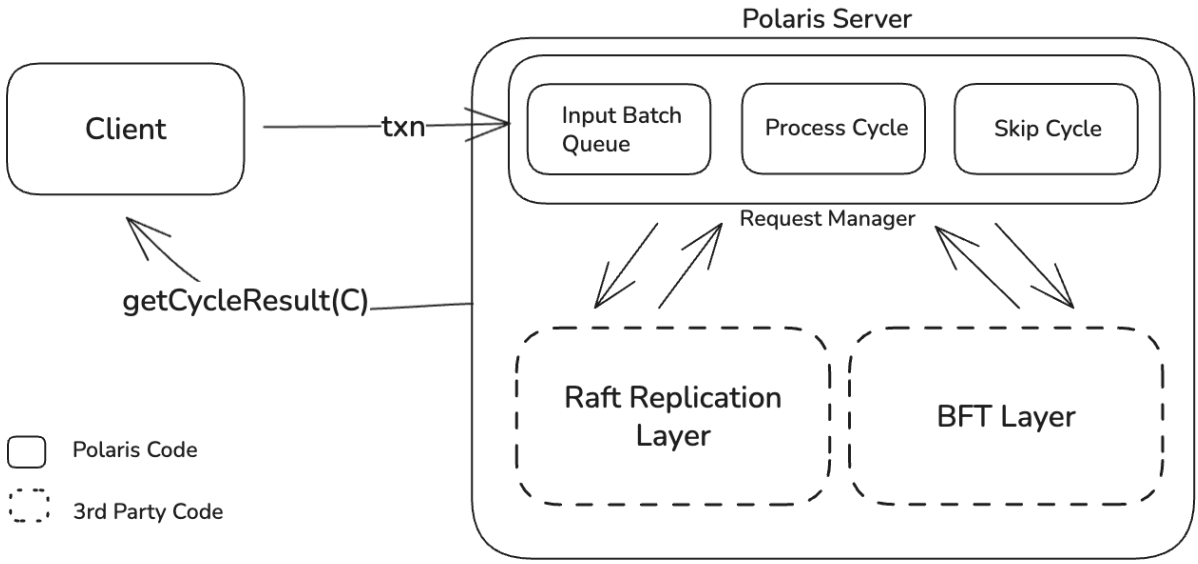


Figure 6.1: Main components of Polaris

from clients. The queue management differs based on the replica's role assigned within the SL. For instance, when transactions arrive at Raft follower nodes, these transactions are immediately forwarded to the current Raft leader within the Super Leaf. Follower nodes do not directly process[2] transactions but act as proxy entry points. This design provides better load distribution for client connections and improves availability, even if the leader fails and a new leader is elected, clients connected to followers experience no disruption.

The Raft leader's transaction input queue serves as the collection point for all transactions within the Super Leaf. Transactions remain in the input queue until they are successfully committed within a transaction block in the SL's Raft consensus. Once Raft commits a block containing specific transactions, those transactions are removed from the queue. This removal policy ensures that the queue size remains bounded even under sustained high load. This design also solves the shortcoming that we encounter due to the inability of SBFT (underlying consensus protocol for BFT layer) but we will discuss this in the subsequent sections. As the SL leader constructs the TB, it assigns each batch a unique ordering identifier that establishes the batch's position within the following BFT transaction sequence. The order id generation logic is straightforward: it sequences the transaction in the block and just takes its digest for the ordering id.

[2]They can batch the incoming client requests.

For the Raft replication layer, we employ etcd-raft implementation (etcd Version 3.3.0) [43, 20]. The choice of Raft rather than a Byzantine fault tolerant protocol for this layer reflects Polaris's infrastructure-aligned fault model discussed in Chapter 4 that within a single datacenter crash fault tolerance is sufficient. Once the SL leader has created the TB and assigns ordering IDs, it invokes the ProcessCycle operation on the replication layer. This operation disseminates the transaction block to all nodes within the Super Leaf, ensuring that every node receives and acknowledges the block. The Raft protocol guarantees that once a majority of Super Leaf nodes have replicated the block to their local logs, it is durably stored and will survive failures of minority nodes.

Next, the TBs are forwarded from all the monitors to the BFT layer where SBFT (Concord v0.5) serves as the underlying consensus service. Since SBFT is implemented in C++, a gRPC acts as a proxy enabling the Polaris monitor sends transaction blocks to gRPC wrapper, the wrapper forwards blocks to the SBFT replica running as a separate process, SBFT processes blocks and produces ordering decisions with cryptographic certificates, and the wrapper returns results to Polaris via gRPC. The results from this phase (Phase 2) are then replicated across all nodes in the Super Leaf using Raft, ensuring that followers obtain the Phase 2 output even though they do not participate directly.

The Request Manager is also responsible for keeping track of cycles and handles the SkipCycle request that deals with stragglers. Internally, the monitor asks the server in its SL to skip cycles. This is achieved using the RequestManager that creates a SkipCycle request with information, including the quorum certificates, skip cycle number, BG information, etc. This SkipCycle request information is appended to the block header and used to set the next cycle number in its BG.

6.3 Implementation Limitations and Design Trade-offs

The primary objective of our implementation is to demonstrate the feasibility of the theoretical Polaris model and to determine how this positions against other protocols in geo-distributed environments. As a research prototype, our implementation makes certain pragmatic trade-offs that simplify development while preserving the essential characteristics necessary for performance evaluation. This section discusses the key limitations of our current implementation and the design decisions made to address them.

6.3.1 Transaction Deduplication

The current Polris Implementation doesn't implement transaction deduplication mechanisms to eliminate redundant transaction submissions. In a complete production system, malicious or faulty clients might submit identical transactions multiple times, either accidentally (due to network timeouts causing clients to retry) or maliciously (attempting to execute the same operation multiple times). Detecting such duplicates requires maintaining transaction history and performing existence checks before accepting new transactions. To address the same we leverage two complementary assumptions and mechanisms:

First, non-malacious client assumption, For the purposes of experimental evaluation, we assume clients behave non-maliciously. This assumption is reasonable for consortium or permissioned blockchain deployments where participants are known, authenticated with reputational and economic incentives to behave correctly. In such settings, intentional duplicate transaction submission by clients is very unlikely, as Byzantine behavior is expected primarily from infrastructure failures or compromised nodes rather than from authenticated client applications.

Second, setting high retransmission timeouts, to prevent accidental duplicate submissions, we configure client retransmission timeouts to be significantly higher than typical consensus completion times. Clients wait for explicit phase 1 CFT acknowledgment that their transaction has been committed before considering retransmission. The retransmission timeout is set to significantly $(2\times - 3\times)$ higher than the expected end-to-end latency, substantially exceeding the time required for consensus to complete under normal operation. This conservative timeout ensures that clients do not prematurely retry transactions that are still progressing through the consensus pipeline, effectively eliminating accidental duplicates in the common case.

However, this limitation doesn't materially affect our performance comparisons with other topologically-aware BFT protocols, as the baseline systems (GeoBFT, Steward, LT-DBFT) similarly lack comprehensive deduplication in their publicly available implementations and rely on client-side duplicate prevention. Our experimental methodology ensures that client request generators do not submit duplicates, creating a level playing field for performance comparison.

6.3.2 SBFT Pipeline Limitations and Batching Workarounds

The SBFT implementation we employ for the phase 2 BFT layer (i.e. Concord v0.5) lacks the native support for the request pipelining. In a pipelined BFT protocol, the system can

process multiple concurrent consensus instances. For example, while instance i is in the prepare phase, instance $i + 1$ can be in pre-prepare, and instance $i + 2$ can be initiating. This overlapping execution amortizes protocol latency across many operations and substantially increases throughput. Without pipelining support, SBFT processes consensus instances sequentially where it first accepts a request, runs the complete consensus protocol to commit it, and only then accepts the next request. Furthermore, the SBFT implementation enforces a strict per-client request limit where each client can have at most one outstanding request at any time. A client must wait for explicit confirmation that its previous request has been executed before submitting a new request. This restriction creates a fundamental throughput bottleneck, as the system's transaction processing rate becomes limited by the number of concurrent clients (SL monitors) multiplied by the inverse of per-request latency. Thus, to maintain acceptable performance in comparison with the theoretical model despite these SBFT limitations, we implemented several compensating mechanisms that can help to mimic the proposed model with the given layer 2 shortfalls.

First, we implement bounded input queue with backpressure where we limit the size of the transaction input queue at each monitor to prevent unbounded memory growth when transaction arrival rate exceeds phase 2 processing capacity. When the queue reaches its capacity threshold, the system applies backpressure by temporarily refusing new transactions from clients or slowing transaction acceptance. Monitors track their queue occupancy and the current SBFT consensus cycle completion rate. When queue occupancy is high (indicating that SBFT cannot keep pace with incoming transaction blocks), monitors reduce their transaction acceptance rate, signaling to clients (through explicit backpressure messages) to slow submission rates.

Second, to maximize SBFT's utilization despite its single request-at-a-time limitation, we aggregate multiple transaction blocks at the gRPC communication layer before submitting to SBFT. From SBFT's perspective, it processes one request per cycle, but that request contains many transaction blocks, each of which contains many individual transactions. This batching approach transforms the system behavior from processing individual transaction blocks sequentially to processing comparatively large batches of transactions sequentially. While each batch experiences the full SBFT consensus latency, the batch contains sufficient transactions that overall throughput remains high.

Also, because the SBFT layer cannot provide the last Prepare message cycle number the SkipCycle and Dynamic Pipeline mechanisms are re-anchored to the Phase 1 complete cycle number. Instead of tracking BFT pipeline status, we leverage the most recently committed Raft cycle from the ingestion layer as the primary indicator for state forwarding.

This limitation affects absolute latency numbers in our evaluation but does not com-

promise the validity of comparative performance analysis. The batching-induced latency overhead affects Polaris and would similarly affect any system using the same SBFT implementation. Moreover, the SBFT bottleneck is orthogonal to Polaris's core contributions (hierarchical coordination, straggler tolerance, optimized inter-region communication), integrating a pipelined BFT implementation would improve absolute performance without changing the relative advantages of Polaris's architectural innovations.

6.4 Implementation of Baseline Protocols and Polaris Variants

To conduct performance comparisons and evaluate the impact of individual architectural features, we implemented three baseline topology-aware BFT protocols (GeoBFT, Steward, and LTDBFT) alongside multiple Polaris variants with different optimization strategies enabled or disabled. This section describes our implementation approach for baseline systems and the architectural variations we evaluated.

For GeoBFT[3] and LTDBFT[4], we leverage the publicly available implementation code repository based on their published specification. However, in the case of GeoBFT rather than deploying the complete ResilientDB system, we extract and use only the consensus service component that implements GeoBFT's core protocol. In the case of Steward, we implemented the protocol from scratch in Go, following the protocol specification from the original publication. Steward's two-tier architecture with centralized coordination through a primary cluster (or site as specified in paper) is conceptually straightforward, making reimplementation feasible. Our implementation[5] includes cluster-based partitioning with local PBFT consensus within the primary cluster, primary cluster selection, and global coordination where non-primary clusters submit transactions to the primary cluster, which establishes global ordering and broadcasts results.

To isolate the performance impact of individual implemented features and optimizations, we implemented multiple Polaris variants with different combinations of features enabled. This ablation study allows us to quantify the contribution of each feature to overall system performance. The baseline variant (Polaris-Base) represents the base architecture without our optimizations, using the original transaction finality rules (waiting

[3]https://github.com/apache/incubator-resilientdb
[4]https://github.com/Lihuatx/LT-DBFT
[5]https://git.uwaterloo.ca/s6joshi/Steward

Table 6.1: Polaris variants implemented for ablation study.

Variant	**Features**
Polaris-Base	Base protocol
Polaris-ChainedFinality	Polaris-Base with new finality rules
Polaris-RCM	Polaris-ChainedFinality
Polaris-OCM	Polaris-ChainedFinality with optimistic global coordination and performance-based envoy selection
Polaris-OneStepSkip	Polaris-OCM
Polaris-PredictiveSkip	Polaris-OCM with Predictive skip
Polaris-StaticPipeline	Polaris-PredictiveSkip
Polaris-DynamicPipeline	Polaris-PredictiveSkip with Dynamic pipeline
Polaris-Full	Polaris-DynamicPipeline

next K cycles), static pipeline depth, and single-step skip cycle mechanism. we then implemented variants enabling specific optimizations: First, Polaris-ChainedFinality changed the transaction finality rules from $C + K$ waiting to $C + 1$ waiting for finality of cycle C. Then, Polaris-OCM implements optimistic execution requiring only 1 envoy interaction with fallback to f other envoys interactions when conflicts are detected. And Polaris-RCM prioritizes safety by requiring $f+1$ envoys interactions for inter-BG result sharing. Polaris-PredictiveSkip implements our redesigned skip cycle mechanism with predictive decision. Following, Polaris-OneStepSkip uses the original single-step skip mechanism for comparison. Polaris-DynamicPipeline adds adaptive pipeline depth adjustment based on system load. Next, Polaris-StaticPipeline version includes optimized finality rules but maintains fixed pipeline depth. Finally, Polaris-Full[6] represents the complete system with improved transaction finality rules, predictive skip cycles and redesigned inter-BG communication with performance-based envoy selection. All these implemented versions are summarized in Table 6.1. This ablation study allows us to quantify the contribution of each optimization to overall system performance and understand the trade-offs between different design choices.

[6]For comparison with other topological-aware protocols we use Polaris-Full as Polaris.

Chapter 7

Evaluation

We evaluate our Polaris prototype through a deployment on a local cluster of computing nodes configured to mimic real-world wide-area network conditions using MaxiNet [55]. MaxiNet is a distributed network emulation framework that extends Mininet to enable large-scale Software-Defined Network (SDN) emulation across multiple physical machines. Our cluster infrastructure is organized hierarchically to simulate realistic geographical distribution: nodes within a superleaf (SL) are connected via high-speed local network switches to simulate co-location within the same data center, SLs within a Byzantine group (BG) are connected through MaxiNet's network emulation that introduces realistic inter-datacenter latencies and bandwidth constraints characteristic of connections within the same geographic region, and BGs are separated by emulation parameters that replicate inter-regional network conditions. This deployment follows our recommendation of organizing participants into multiple geographically-proximate Byzantine groups in order to take advantage of high-capacity regional network links.

Table 7.1 shows the emulated bandwidth and latency characteristics between nodes at different hierarchy levels, configured to match measured values from real-world wide-area deployments. In accordance with the realistic constraints, nodes within the same simulated region experience bandwidth of nearly 9 Gb/s with sub-millisecond latencies, while cross-region connections are throttled to between tens and hundreds of Mb/s with latencies ranging from 30ms to 120ms depending on the simulated geographical distance. MaxiNet allows us to enforce these network characteristics precisely across our distributed cluster, ensuring that our local deployment accurately reflects real-world network constraints.

While our experiments were performed on a local cluster with MaxiNet-emulated network conditions, a similar experimental methodology can be applied to actual distributed

Table 7.1: Available bandwidth (Gbps) and roundtrip latency (ms) between clusters.

Region	**Region 1**	**Region 2**	**Region 3**	**Region 4**
Region 1	9.8 Gbps†	30 ms	90 ms	60 ms
Region 2	0.25 Gbps	9.4 Gbps†	65 ms	100 ms
Region 3	0.25 Gbps	0.25 Gbps	9.8 Gbps†	120 ms
Region 4	0.25 Gbps	0.25 Gbps	0.25 Gbps	9.6 Gbps†

Note: Upper triangle = RTT (ms); Lower triangle = Bandwidth (Gbps).

†Latency between replicas in the same datacenter (Intra-DC) is under <1ms.
Latency between replicas on different hosts within the same region (Inter-DC) is <10ms.

deployments across real data centers or cloud infrastructures. In a realistic deployment, we expect participants to use different cloud providers to limit the damage of an attack based on a cloud-provider specific security vulnerability. Our MaxiNet-based cluster approach allows us to precisely control and reproduce network conditions while avoiding the costs and variability associated with actual multi-cloud deployments. However, we expect the bandwidth between cloud providers in the same region to be higher than the bandwidth within the same cloud provider across different regions. Therefore, our deployment strategy should largely remain unchanged even when deployed in actual multi-cloud environments. We focus our evaluation on state-of-the-art topology-aware protocols, omitting flat-network consensus approaches, as prior work [24, 54, 41] has demonstrated that hierarchical, cluster-based architectures significantly outperform flat designs at planetary scale.

Unless stated otherwise, in line with SBFT [23], our experiments use 240 byte transactions issued from clients that are co-located with our SLs. All the nodes in an SL are running on a single physical machine equipped with dual-socket Intel Xeon E5-2620 v2 processors (12 cores 2.10 GHz) and 64 GB DDR3 @ 1866 MHz memory, and each BG consists of four such SLs. Our cluster nodes are equipped with 10-Gbps network interfaces to avoid creating artificial bottlenecks, as Polaris is throughput-bound and requires high network capacity to achieve its performance potential. Polaris is configured with a pipeline depth of 16 and a maximum batch size of 5000 requests. We run each experiment for 240 seconds with a 60-second initialization period before collecting measurements for the subsequent 180 seconds. All reported results represent averages across three independent runs. Our evaluation is organized around five research questions. The first examines how horizontal scaling through additional geo-distributed Byzantine Groups (BGs) affects Polaris' performance relative to existing consensus protocols. The second investigates the

impact of topological grouping of replicas on Polaris' performance characteristics compared to competing approaches. The third explores how increasing the number of replicas per cluster influences throughput and latency across protocols. The fourth considers the impact of straggler BGs or clusters on Polaris' performance, and assesses how its straggler mitigation strategy compares to those of other protocols. Finally, the fifth uses a series of microbenchmarks to isolate and evaluate the contribution of each individual design choice incorporated in Polaris.

7.1 Impact of Horizontal Scaling

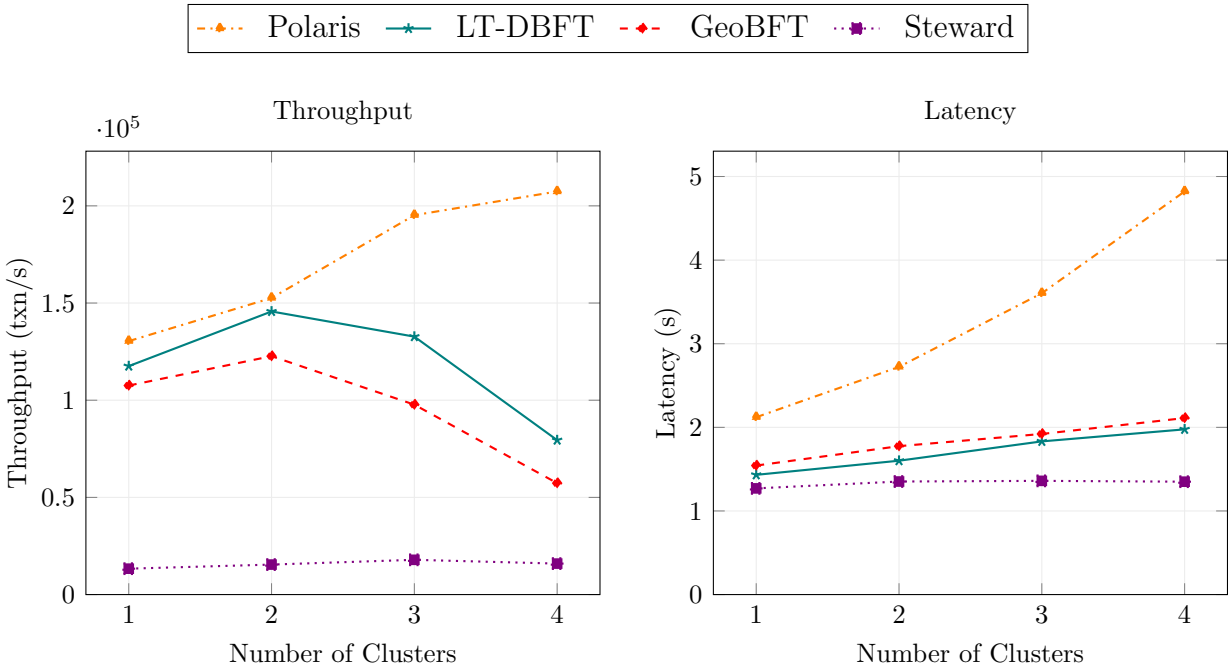


Figure 7.1: Throughput and Latency as a function of Cluster Count (z): Performance scaling results for $12z$ replicas.

First, to evaluate the horizontal scalability of Polaris against other topology-aware protocols, we measured throughput and latency as a function of the number of BGs or clusters (z), varying z from 1 to 4. In this configuration, the total replica count increases proportionally from 12 to 48 ($12z$). The results, illustrated in Figure 7.1, demonstrate

the trade-offs between increased computational resources and inter-cluster communication overhead.

Steward suffers from high computational costs, concentrated in a single primary cluster due to its primary-dependent centralized design where a single primary cluster is responsible for BFT consensus through "Assign-Sequence" and the global dissemination of results. This creates a performance bottleneck at the primary cluster, rendering the system unable to leverage additional clusters for performance gains. Consequently, throughput remains stagnant (approx. 14k to 17k txn/s) regardless of horizontal scaling.

While GeoBFT's decentralized design initially provides a performance boost as the cluster count increases from 1 to 2, throughput begins to decline as z progresses to 3 and 4. This performance degradation is primarily due to the inter-cluster communication complexity of $O(z^2 f)$ where z is the number of clusters each with $3f+1$ replicas. This overhead eventually outweighs the benefits of adding more nodes. Similar to GeoBFT, LT-DBFT shows an initial performance advantage, particularly in single-cluster configurations, due to its delegated committee-based local BFT process. However, it faces a sharp throughput dip at $z = 4$. The global sharing communication overhead for LT-DBFT is $O(z^3)$, which increasingly becomes prohibitive as the cluster count increases.

Finally, Polaris has the highest horizontal scalability, climbing steadily to a peak throughput of 207k txn/s at $z = 4$. This is achieved by leveraging a two-layer intra-BG consensus where Phase 1 (Raft) allows for parallel transaction collection and processing. Expensive BFT processes are then delegated to a smaller subset of replicas (SL monitors). Furthermore, the optimistic inter-BG sharing complexity of Polaris is $O(z^2)$, allowing it to outperform competitors by a factor of 2.6x to 3.6x. On latency, however, Polaris exhibits higher absolute base latency than its competitors, owing to its taller consensus hierarchy and the absence of pipelining support in the Phase 2 BFT consensus stage. Furthermore, latency grows as more BGs are added for two compounding reasons. First, finalizing a cycle requires waiting $C + 1$ cycles for cycle C to be finalized, a cost that scales with BG count. Second, while the $O(z^2)$ optimistic coordination reduces inter-BG bandwidth consumption, it shifts load onto inter-SL communication, increasing inter-SL bandwidth utilization as the number of clusters grows and contributing to the observed latency increase.

7.2 Impact of Parallel Byzantine Agreement

Next, we examine the performance improvement that Polaris obtains from grouping nodes into smaller geographically-proximate BGs, and performing agreement within each BG in

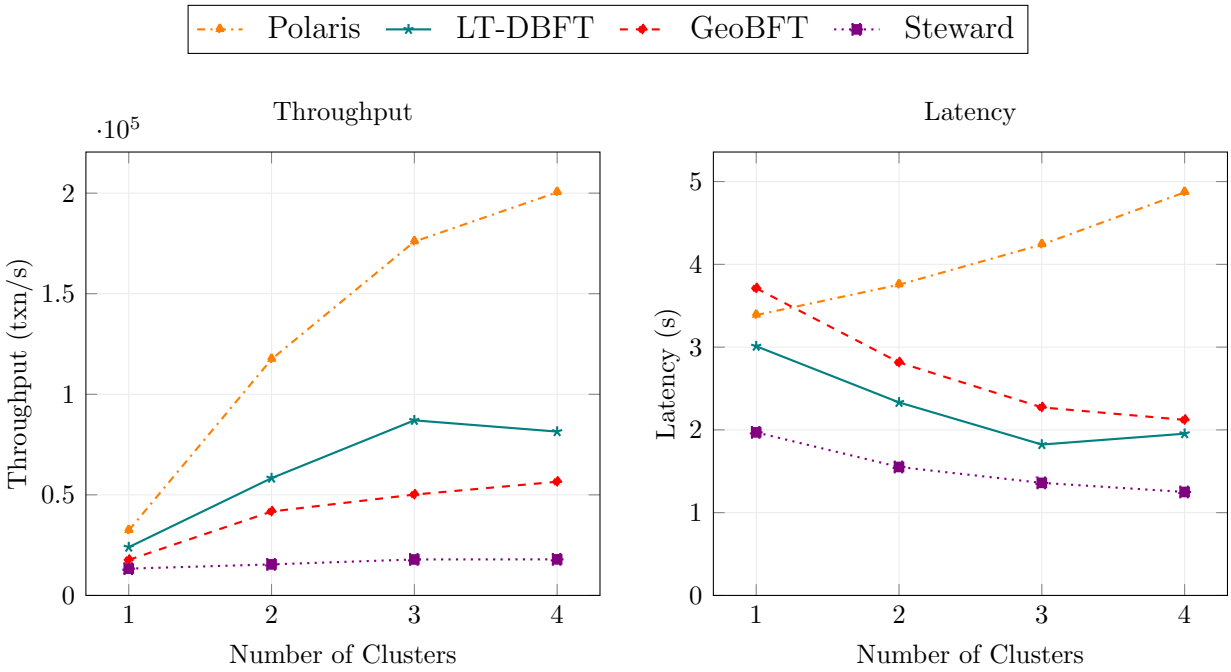


Figure 7.2: Throughput and Latency as a function of Cluster Count (z): Performance scaling results for a fixed total of $zn = 48$ replicas.

parallel. In this experiment, we deployed Polaris and other consensus protocols across all 48 of our replicas and configured it to have 1, 2, 3, and 4 clusters, with each BG or cluster consisting of 48, 24, 16, and 12 replicas respectively. The results are shown in Figure 7.2.

Again, as we can observe from the measurements, Steward serves as the baseline for sequential processing. Its centralized design and reliance on a single primary cluster prevent parallel Byzantine agreement, constraining throughput regardless of the number of clusters deployed. The addition of clusters merely introduces replication overhead without enabling performance gains, resulting in a flat throughput response.

With all the replicas in a single cluster ($z = 1$), Polaris outperforms all the other consensus protocols by at least 43% in terms of throughput. This advantage stems primarily from its use of a CFT protocol (Raft) within each SL, which reduces the number of nodes participating in Byzantine agreement from 48 to 16. LT-DBFT also demonstrates competitive performance in this configuration. However, its threshold-based trust scoring mechanism causes the committee size participating in BFT consensus to expand progressively as more nodes exceed the trust threshold. By the end of the experiments, effectively all nodes participate in BFT consensus, degrading its performance to levels comparable

with GeoBFT, which leverages simple PBFT.

Polaris' peak throughput increases to almost 117k tps with 2 BGs. The performance increase is due in part to the smaller BGs or clusters, but also due to the increased parallelism we obtain from concurrently performing Byzantine agreement across both groups. This similar trend is also observed in both GeoBFT and LTDBFT. With the further topological division of the replicas Polaris continues to outperform both GeoBFT and LTDBFT. As all three schemes distribute the transaction processing and thereby reduce global communication they continue to improve performance with the increase of BGs or clusters as we can observe from Figure 7.2. With 4 clusters, which is the maximum number of BGs that Polaris can have with a 48 node deployment, peak throughput jumps to over 200,000 tps representing at least a 2.5x improvement over competing protocols. This substantial gain is primarily attributable to significantly reduced cross-region traffic. With 4 BGs, each BG becomes small enough that SLs within a BG are predominantly selected from the same region. As shown in Table 7.1, intra-region bandwidth substantially exceeds inter-region bandwidth, making this reduction in cross-region communication critical for performance. While Polaris and GeoBFT maintain positive throughput scaling as BG count increases, LT-DBFT exhibits a decrease in throughput at $z = 4$. For LT-DBFT, the expensive inter-cluster communication overhead, which scales with $O(z^3)$, eventually overshadows the gains from parallel Byzantine agreement at higher cluster counts.

Regarding latency, Polaris exhibits an expected trade-off. With 1 BG, latency is dominated by the time required to reach Byzantine agreement across 16 geographically distributed SLs. At $z = 2$, latency increases to 3.7s as Phase 3 (inter-BG state dissemination) becomes necessary. With 4 BGs, latency reaches 4.8s, primarily due to the substantially higher throughput generating a larger volume of data that must be disseminated across BGs in Phase 3. In contrast, other protocols demonstrate a slight reduction in latency as BG count increases, since the replica-level interactions required for their BFT consensus phases incur higher latencies in single-cluster configurations than in distributed multi-cluster deployments. As we decrease the cluster size, the participants in the BFT phase get reduced and consequently the overall communication overhead of BFT phase.

7.3 Impact of Byzantine Group Size

In our third experiment, we tested the impact of the number of replicas per BG on the performance characteristics of Polaris and other consensus protocols. For this experiment, we measure the throughput and latency by different consensus schemes as a function of

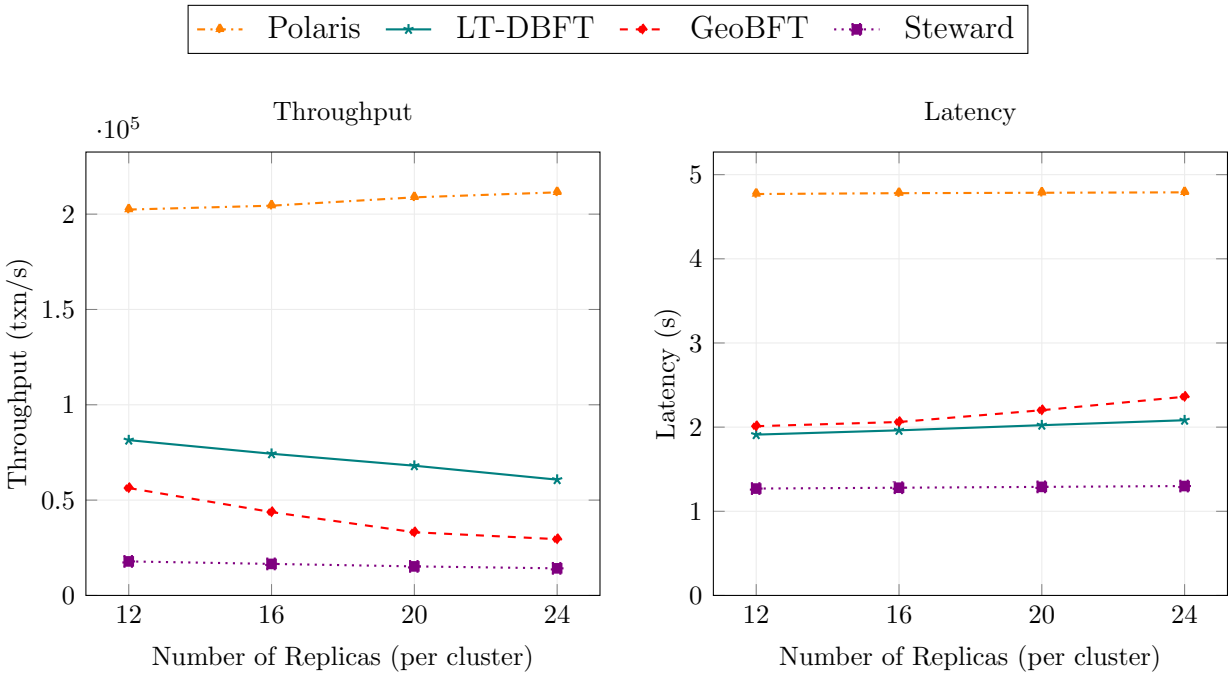


Figure 7.3: Throughput and Latency as a function of number of replicas per cluster: Performance scaling results for a fixed total $z = 4$ clusters.

number of replicas per cluster, varying it from 12 to 24 while keeping the overall number of clusters fixed to 4.

As illustrated in Figure 7.3 Steward's throughput remains flat regardless of cluster size. On the other hand, GeoBFT exhibits a noticeable performance degradation as the number of nodes per cluster increases. Throughput declines sharply from 56k txn/s at $n = 12$ to 30k txn/s at $n = 24$, representing a 48% performance loss. The significant contributors for this decline include: first, the size of certificates exchanged between clusters scales with f (the number of tolerable Byzantine failures per cluster, which increases with cluster size); second, each cluster must send certificates to $f + 1$ replicas in every other cluster.

As cluster size increases, GeoBFT suffers a sharp performance decline driven primarily by its $O(z^2 f)$ inter-cluster coordination complexity and the BFT overhead of additional replicas. While LT-DBFT benefits from cluster-size-independent global sharing, it still exhibits a moderate decline as its trust-scoring mechanism expands the committee size when nodes exceed specific thresholds. In contrast, Polaris throughput improves from approximately 202k txn/s at $n = 12$ to 211k txn/s at $n = 24$ by leveraging a fixed-

overhead hierarchical design that insulates the consensus critical path from replica scaling, maintaining high performance. The combination of increased parallel processing capacity in Phase 1 and fixed overhead in Phases 2 and 3 allows Polaris to leverage additional nodes effectively, achieving a 3.5× higher throughput than LT-DBFT and 7.2× higher throughput than GeoBFT at $n = 24$. Regarding latency, Polaris maintains stable latency across all cluster sizes, varying less than 0.5%. This stability confirms that the critical path of the protocol remains least affected with cluster size variations.

7.4 Impact of Straggler Groups

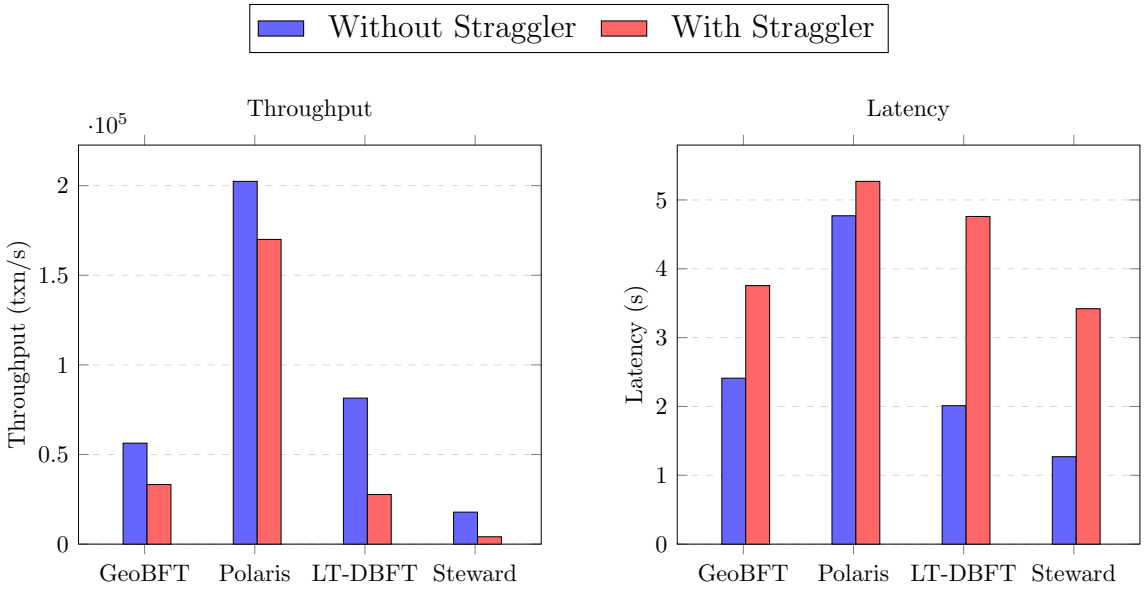


Figure 7.4: Impact of Straggler Cluster on Throughput and Latency.

Finally, we examine the impact of a straggler cluster on the performance of Polaris and other consensus protocols, along with the effectiveness of straggler mitigation techniques. In this experiment, we introduced artificial delays in one of the four clusters to simulate straggler behavior. Specifically, we injected sleep delays after transaction processing phases to slow down the cluster's consensus participation by approximately 3×. We then measured the throughput and latency of each protocol with and without the straggler cluster present. The results are shown in Figure 7.4.

Steward and LT-DBFT, which lack explicit straggler mitigation mechanisms, suffer significant performance degradation. Steward experiences a 77% throughput drop and a 2.8× latency increase (in this case delays are introduced in the primary cluster). Similarly, LT-DBFT faces a 65% throughput decline and a 2.35× latency increase. On the other hand, GeoBFT offers an indirect mitigation approach through its *no-op* proposal mechanism. When a cluster's results are not ready within the current consensus round, it can propose an empty (*no-op*) block to maintain protocol liveness. This mechanism was originally designed to handle periods of low client load, not straggler scenarios. The wait times for results from the straggler eventually inflate the average latency. Consequently, GeoBFT incurs a 41% loss in throughput and a 1.5x increase in average latency.

Polaris demonstrates high resilience through its specialized SkipCycle mechanism. By leveraging metadata shared during the global sharing phase, Polaris monitors determine the optimal cycle to skip. Rather than issuing a series of empty cycles, the straggler continues processing transactions to catch up with peer BGs, effectively skipping to the most appropriate cycle number. This proactive synchronization results in only a 16% loss in throughput and a 1.1× latency increase.

7.5 Impact of Optimistic Inter-BG Coordination

This section of the ablation study investigates the Inter-BG coordination mechanism. In this experiment, we are focusing on the performance disparity between the Optimistic Coordination mechanism and the Redundant Coordination Mechanism. As one of the recurring themes in our planetary-scale experiments against other topological models is that the protocols managing inter-BG bandwidth most efficiently consistently achieve superior performance. And OCM facilitates Polaris to achieve the same. To do so, we measure the throughput and latency attained by Polaris under variables while using OCM and RCM as inter-BG coordination mechanisms respectively.

The measurements (as depicted in Figure 7.5) show that OCM consistently outperforms RCM under all the load conditions. The gain from RCM is 1.6x for the lower loads and significantly increases for the higher loads 2.25x. The Resilient Path hits a throughput wall early because the $2f + 1$ redundant message exchanges saturate the available inter-BG bandwidth. In contrast, the Optimistic Path maintains a stable latency curve until it saturates the available inter-BG and inter-SL bandwidths. The OCM achieves a peak throughput of over 200,000 txn/s, effectively doubling the performance of the RCM, which plateaus at approximately 93,000 txn/s. At lower throughput levels, the latency between the two paths is comparable (approx. 1.8s to 2.2s). However, as load increases, the Resilient

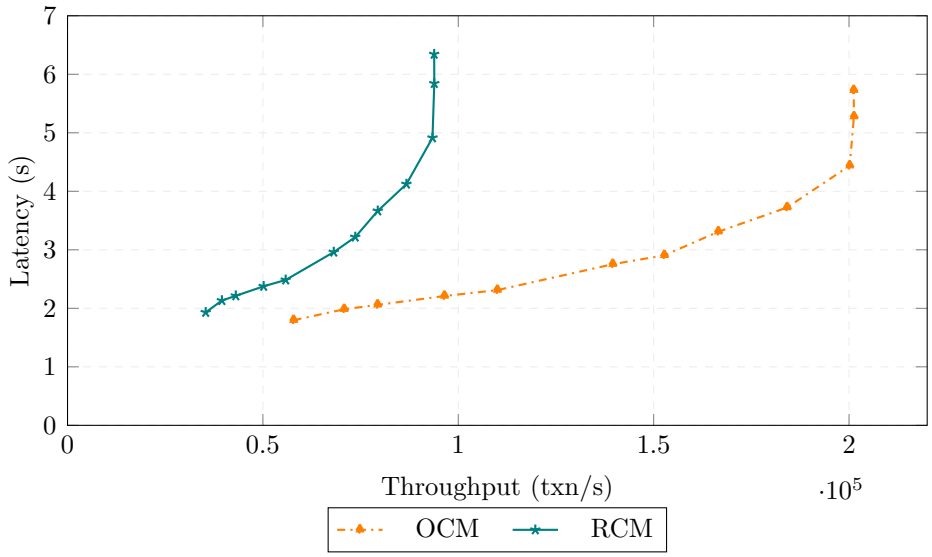


Figure 7.5: Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism.

Path suffers from exponential latency growth, crossing the 6s mark at less than half the peak throughput of the Optimistic Path.

While the OCM provides a significant throughput leap by eliminating redundant BFT result transfers during failure-free execution, its reliance on a single primary coordinator introduces a sequential recovery penalty when the optimistic path fails and fallback methods are triggered. To characterize this trade-off, we analyzed Polaris's resilience under three distinct failure modes: Emulator Omission, Envoy Jitter, and Envoy Failure. First, Emulator Omission: We introduced a 10% failure rate where primary Envoys interacted with a faulty Emulator that omitted all incoming BFTCompletionProof messages. Next, Envoy Jitter: We simulated lazy primary Envoys that delayed broadcasting BFT results until reaching 90% of the timeout duration (τ_{max}). This test demonstrates the impact of a straggling coordinator on global finality. Last, Envoy Failure: We introduced a 10% failure rate where the primary Envoy performed an act of omission, refusing to submit results from remote BGs.

In the case of Emulator Omission (refer Figure 7.6), OCM still achieves a peak throughput of approximately 171 ktxn/s, representing an 82% lead over the Resilient baseline (94 ktxn/s). While RCM is inherently immune to Emulator Omission by masking the failure through parallel redundant fetches, its baseline performance is severely capped by the con-

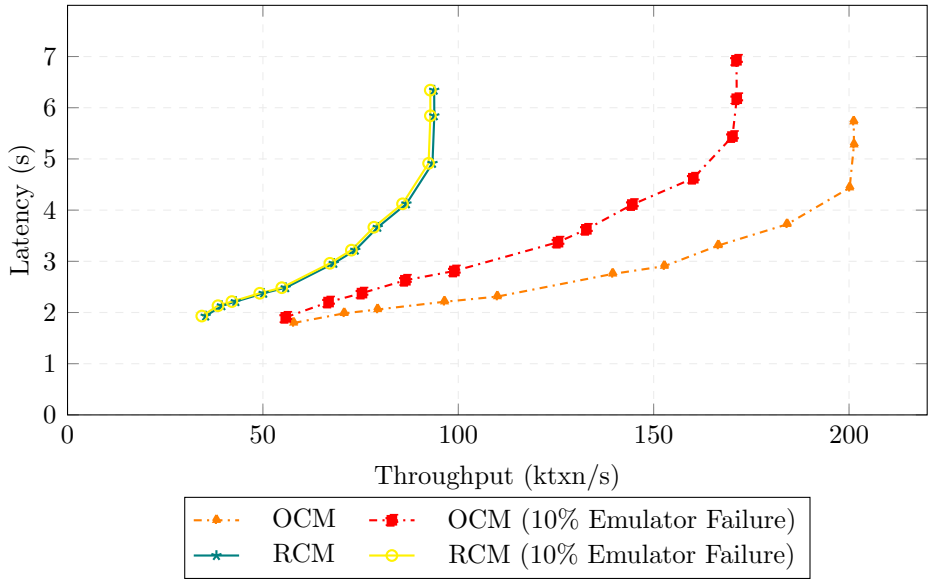


Figure 7.6: Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under 10% stochastic Emulator failure probability per cycle.

stant bandwidth overhead of $2f+1$ fetches per cycle. In terms of latency, RCM exhibits no meaningful change, as its redundant fetch mechanism absorbs the emulator failure. OCM, however, incurs a significant latency penalty driven by two factors: timeout delays at the primary envoy, and the accumulation of a transaction backlog that progressively reduces effective pipelining depth as load increases.

In case of Envoy jitter (refer Figure 7.7), we observe that OCM suffers from a throughput loss of around 5% at higher loads. This loss stems from the fact that the delay induced by the primary envoy leads to the underutilization of the pipeline where at higher loads it creates backlog due to pipeline capacity but that is not due to network condition or CPU bottleneck but due to late shifting of execution window, where delayed envoy responses prevent the pipeline from advancing its transmission window in time, leaving capacity underutilized at precisely the moments when load is highest. We also observe that OCM latency increases as the primary envoy delays result broadcasts up to 90% of the timeout limit. At lower loads, this increase closely tracks the delay introduced by the primary envoy and at higher loads it compounds further due to additional wait time from pipeline stalls. Conversely, RCM remains unaffected by this jitter, as it naturally accepts the fastest $f+1$ responses and bypasses the slow primary.

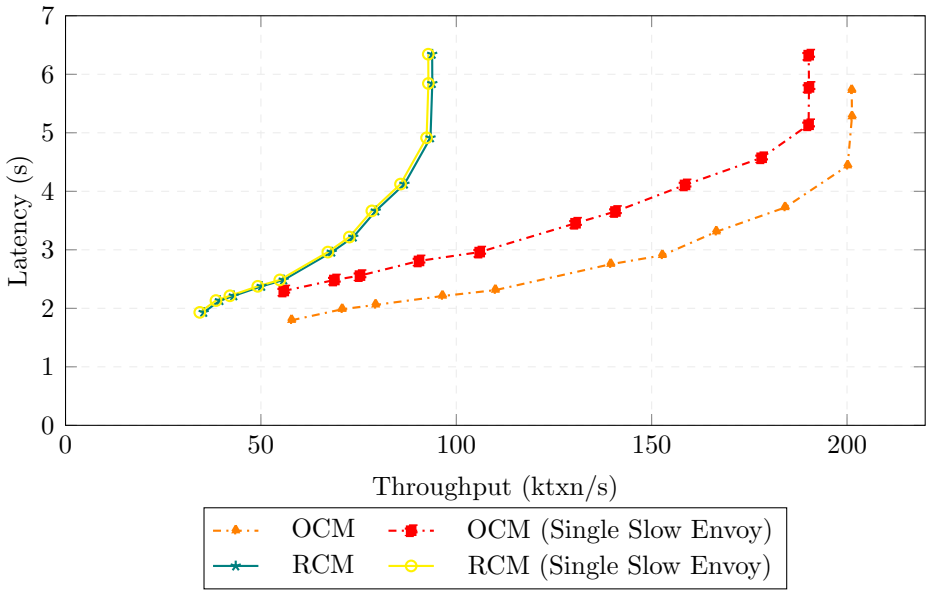


Figure 7.7: Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under a targeted straggling attack by a malicious Envoy inducing a sub-timeout propagation delay ($< \tau_{max}$).

Under Envoy Failure (Figure 7.8), OCM throughput drops to 141k txn/s. Intuitively, the throughput loss should be comparable to Emulator Omission, since the same number of copies must be fetched and same pattern of inter-BG bandwidth utilization. However, the additional degradation stems from the longer timeout[1], which saturates the pipeline earlier and compounds the performance loss. In terms of latency, we observe a trend similar to emulator omission where envoy failure induces latency penalty in OCM, as the primary's omission triggers a full sequential fallback to secondary coordinators. Every cycle must wait for a complete timeout, causing latency to climb toward 7s and drastically reducing the effective pipeline depth creating additional waiting time for lower level results.

Despite this sequential fallback, OCM still outperforms RCM's result sharing approach across all three failure scenarios. Nevertheless, the performance penalties observed across these scenarios motivate Polaris' Convergence Scoring mechanism, which mitigates such losses by promoting a more reliable SL to the primary role and tracking emulators health[2], effectively restoring the high-performance path and recovering the throughput and latency

[1] 750 ms

[2] Emulator health is monitored via the global membership service, which maintains an up-to-date registry of operational nodes.

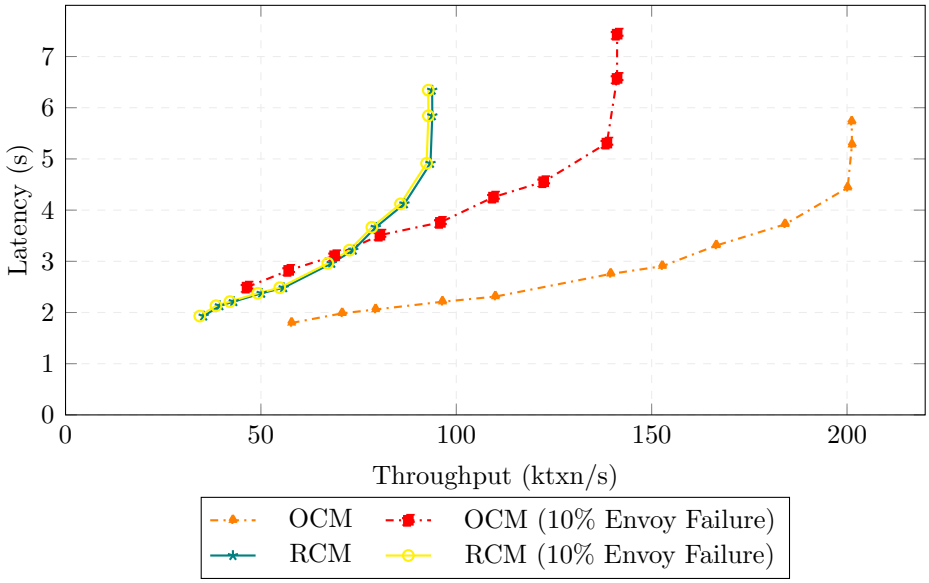


Figure 7.8: Comparison of Throughput-Latency trade-offs between the Optimistic Coordination Mechanism and the Redundant Coordination Mechanism under 10% stochastic Envoy failure probability per cycle.

to nominal levels. These results demonstrate that the bandwidth efficiency gained through optimism outweighs the transient latency penalties of sequential recovery in wide-area network deployments.

7.6 Impact of Pipeline Depth

In a planetary-scale system, throughput is a function of the Bandwidth-Delay Product (BDP). If the pipeline depth K is too shallow, the network sits idle while waiting for cross-continental acknowledgments; if it is too deep, latency explodes without throughput gain. Furthermore, the variance in transaction load across geographically distant BGs creates regional drift, necessitating a pipeline that can dynamically expand to accommodate this skew without inducing unnecessary buffer bloat. In this section, we determine the impact of the pipeline depth and inter-BG bandwidth on the performance of Polaris. To do so, we measure the throughput and latency attained by our prototype as a function of the pipeline depth d, which we vary between 8 and 36, across three different bandwidth configurations: 250 Mbps, 500 Mbps, and 750 Mbps. The results of our measurements can be found in

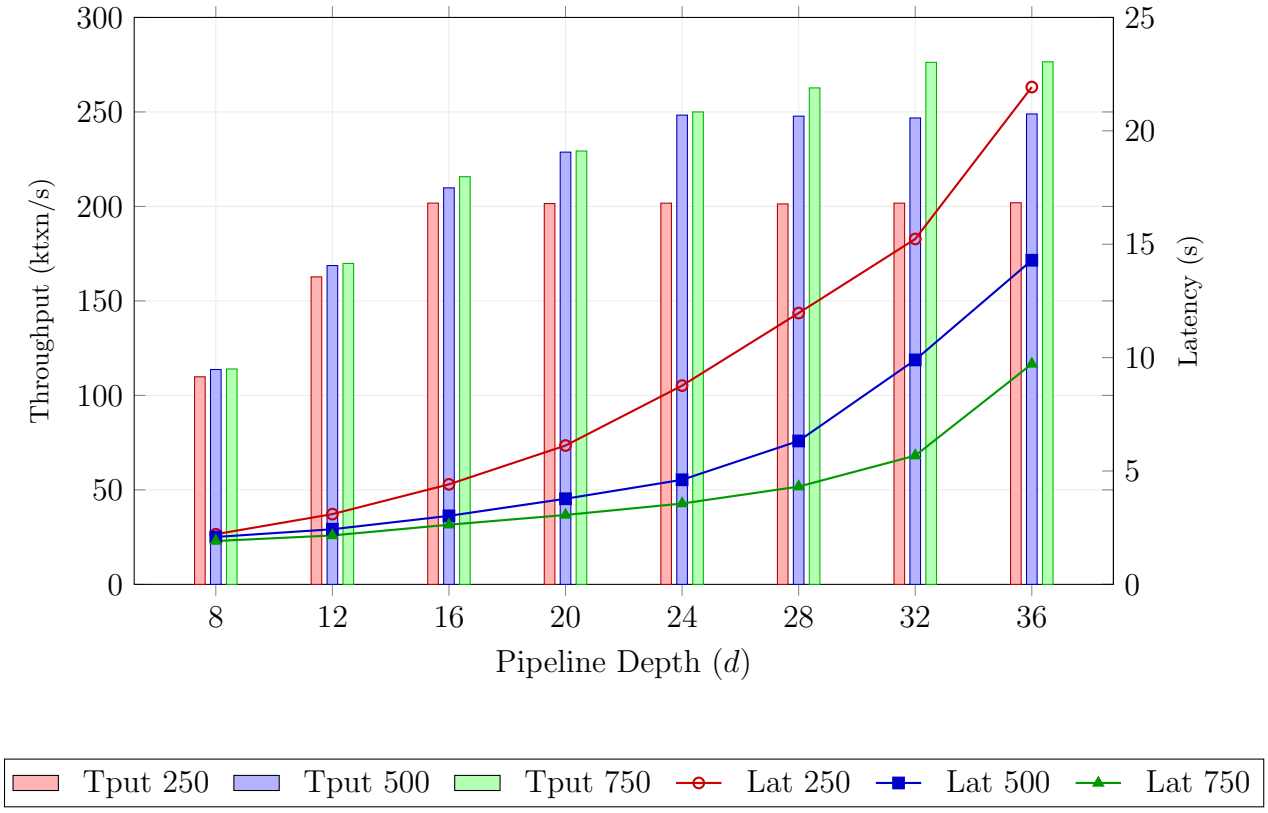


Figure 7.9: Throughput and Latency vs. Pipeline Depth across varying inter-BG Bandwidths (250, 500, 750 Mbps). Bars represent throughput; matching-colored lines represent latency.

Figure 7.9.

For the 250 Mbps configuration, throughput climbs linearly until $d = 16$, where it plateaus at approximately 201.7 ktxn/s. When bandwidth is increased to 500 Mbps and 750 Mbps, the saturation points shift to $d = 24$ and $d = 32$, respectively, reaching peak throughputs of approximately 248.9 ktxn/s and 276.5 ktxn/s. While throughput hits a physical ceiling defined by the bandwidth, latency continues to rise exponentially once the pipeline is full. For instance, at 250 Mbps, increasing the depth from $d = 16$ to $d = 36$ causes latency to spike from 4.4s to over 21s with no measurable throughput gain. Interestingly, one might expect the saturation point to scale proportionally with bandwidth since 250 Mbps saturates at $d = 16$, a doubling to 500 Mbps should ideally push saturation to $d = 3$. However, this is not observed; 500 Mbps saturates around $d = 24$ instead. This deviation is explained by the inter-SL bandwidth bottleneck, as inter-BG bandwidth increases, the system can sustain higher throughput per cycle, which in turn increases the volume of intra-BG result sharing among SL monitors. This inter-SL coordination overhead grows with load, creating a secondary bottleneck that prevents the saturation point from scaling linearly with inter-BG bandwidth.

These results highlight a fundamental challenge: the optimal pipeline depth is not a fixed system parameter but a function of prevailing network conditions and system load. At 250 Mbps, depths beyond $d = 16$ yield negligible throughput gains while incurring severe latency penalties, whereas at 750 Mbps the system continues to benefit from deeper pipelines well beyond that point. Since inter-BG bandwidth in geo-distributed deployments is inherently variable, and BGs may experience asymmetric or fluctuating load, statically configuring pipeline depth risks either under-utilizing available bandwidth or over-committing the pipeline and inflating latency. This motivates the need for a dynamic pipeline depth optimization mechanism that continuously adapts to observed network conditions and load distribution across BGs.

7.7 Impact of Finality Semantic

This section evaluates the performance impact of transitioning from traditional pipeline-based finality to our proposed Chained Finality model. We compare the throughput and latency of Polaris using Chained Finality ($C+1$) against a static Pipeline Finality baseline with depths of $k = 8$ and $k = 16$. The results are presented in Figure 7.10.

At low throughput levels, Pipeline Finality configurations ($C + k$) exhibit significantly higher latency compared to Chained Finality. This is due to the Pipeline Starvation where

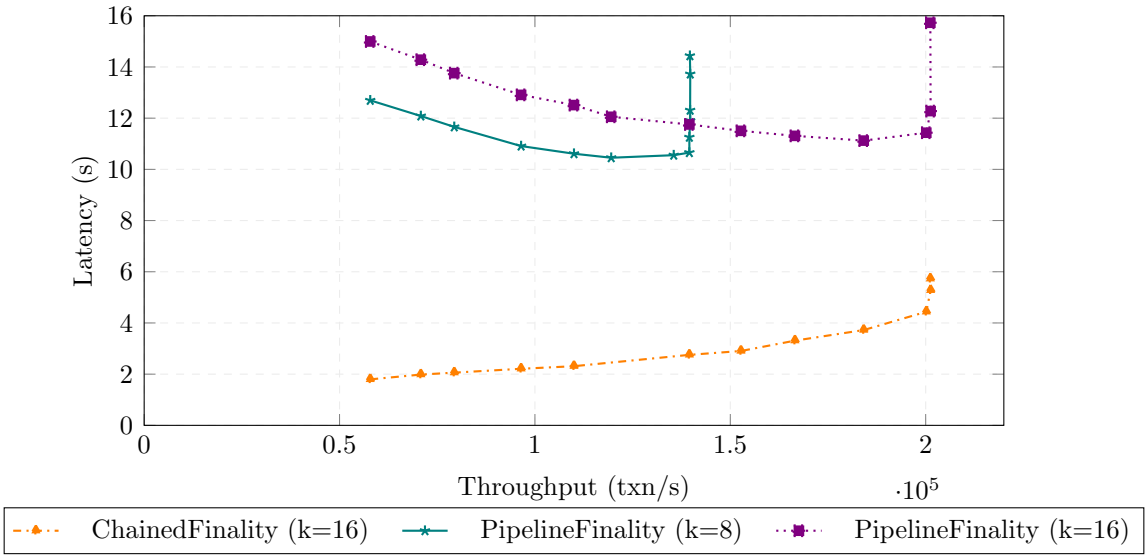


Figure 7.10: Comparison of Throughput-Latency trade-offs between the ChainedFinality and PipelineFinality setup.

the system must wait for k subsequent cycles to be initiated and filled before confirming cycle C, even if the replication for C was completed nearly a round-trip earlier. As throughput decreases, the time required to fill these k slots increases, causing the finality horizon to stretch significantly beyond the actual communication time. Our measurements show that increasing the pipeline depth from $k = 8$ to $k = 16$ under low load increases average latency by approximately 2 seconds without providing any throughput benefit. In contrast, Chained Finality mitigates this penalty by decoupling transaction finality from the total pipeline depth. By anchoring the finality of cycle C to the immediate progress of $C + 1$, Polaris provides near-optimal latency regardless of the system's current load.

Furthermore, the data proves that Chained Finality matches or exceeds the peak throughput of traditional pipeline-based models. This demonstrates that enforcing a $C+1$ anchor with strict ordering does not throttle the system's ability to process high-volume bursts. As shown by the orange line in Figure 7.10, Chained Finality maintains a stable latency curve across the entire throughput spectrum. It is important to note a specific edge case: during an Optimistic Path failure, Chained Finality may briefly exhibit higher perceived latency than $C+k$ Pipeline Finality. In a $C+k$ model, the sequential recovery delay for a missed cycle C can be masked behind k cycles of ingestion, effectively amortizing the stall across a larger window. In Polaris, the $C+1$ requirement forces the client to wait for a proof from the immediate successor, making the recovery delay more visible. However,

this is a conscious design choice where Polaris prioritizes deterministic, low-latency finality over the artificial masking of straggler events.

7.8 Impact of Predictive Skipping

In this section we investigate the effectiveness of the Predictive Skip-Cycle mechanism in mitigating the impact of straggler BGs. In a strictly ordered system, the global progress is typically gated by the slowest region. Polaris addresses this through its Skip-Cycle logic, which allows the network to bypass straggling BGs without halting or drifting the protocol's performance toward the slowest BG. We compare our proposed PredictiveSkip (which anticipates drift based on metadata shared by the BFTCompletionProof of previous cycle) against a reactive OneSkip baseline (which only skips after a cycle lag is detected).

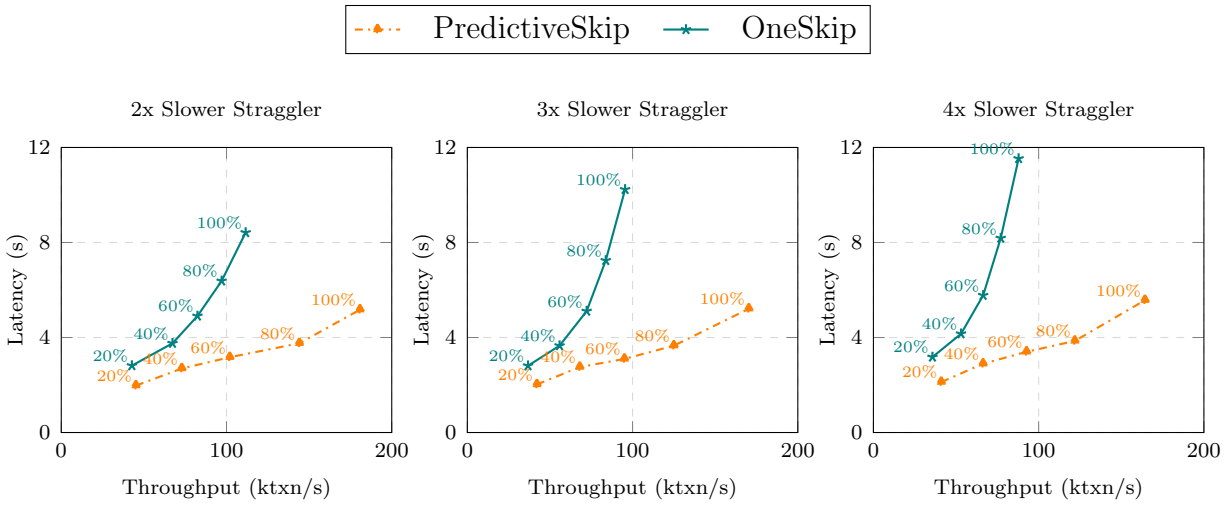


Figure 7.11: Comparison of Throughput-Latency trade-offs between the PredictiveSkip an OneSkip under three different straggler configurations with pipeline utilization for pipeline depth $k = 16$.

Next, we determine the impact of straggler severity and pipeline utilization on the performance of Polaris. To do so, we measure the throughput and latency of the system under three different straggler configurations: 2x, 3x, and 4x slower than the regional average in a 4 BG configuration. In the first experiment, we introduced artificial delays in one of the four BGs to simulate straggler behavior. Specifically, we injected sleep delays

after transaction processing phases to slow down the cluster's consensus participation. For the second experiment, we use a single 3x slower BG and compare the behaviour of both PredictiveSkip and OneSkip as a function of pipeline depth. The results of our measurements are presented in Figure 7.11 and Figure 7.12 respectively.

The measurements in Figure 7.11 demonstrate that PredictiveSkip consistently maintains a superior throughput-latency profile across all straggler intensities. PredictiveSkip exhibits a stable latency curve, maintaining sub-4s latency even as throughput approaches 125 ktxn/s in the 3x straggler case. In contrast, the OneSkip baseline suffers from a reactive skip, causing latency to climb toward 10s and beyond at much lower throughput levels. PredictiveSkip allows the system to reach significantly higher peak throughput. For example, in the 4x slower straggler scenario, PredictiveSkip peaks at 164 ktxn/s, whereas OneSkip plateaus near 88 ktxn/s. The disparity arises because PredictiveSkip preemptively adjusts skip decisions, preventing the Phase-3 stalls that afflict the reactive model.

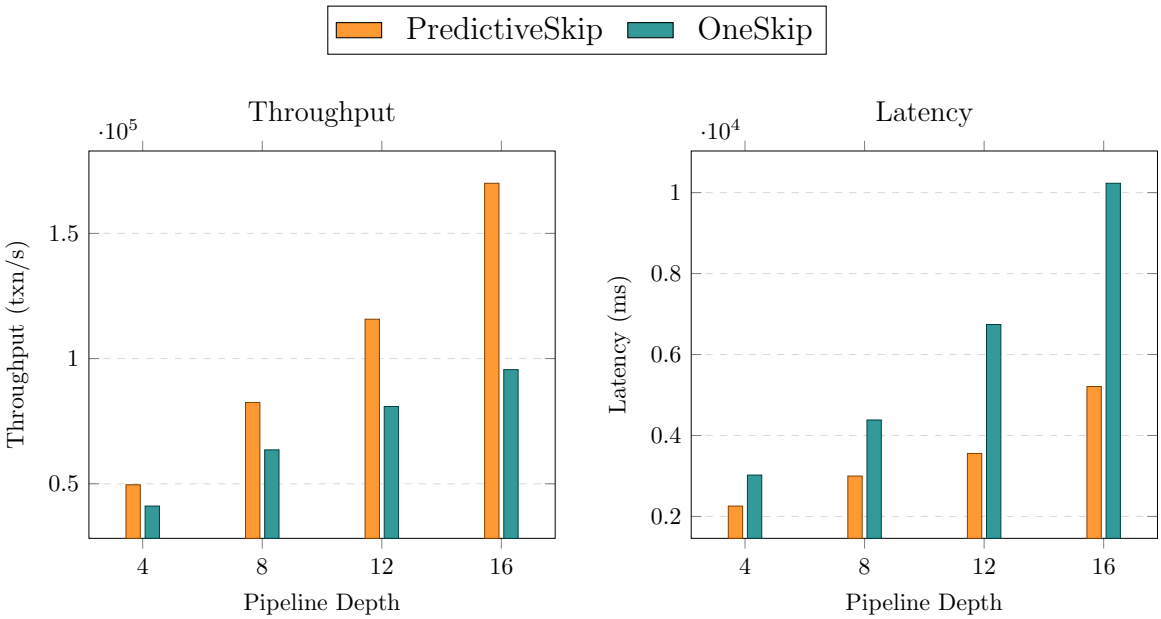


Figure 7.12: Throughput and Latency comparison for PredictiveSkip and OneSkip across different pipeline depths.

Specifically, OneSkip's single reactive skip forces all in-flight pipeline cycles to wait for an additional Phase-2 completion before the next skip can be issued, inducing Head-of-Line

(HoL) Blocking that stalls the pipeline with pending cycles from faster BGs. This bottleneck is also evident in Figure 7.12, where a deeper pipeline partially alleviates OneSkip's throughput by amortizing the stall cost. As shown in Figure 7.11, the performance gap between the two models widens as pipeline utilization increases (or when the load across the system increases). A similar trend is observed in Figure 7.12 where at a pipeline depth of 16, PredictiveSkip achieves nearly 1.8x the throughput of OneSkip (170 ktxn/s vs 95 ktxn/s), while maintaining roughly half the latency (5.2s vs 10.2s). These findings also align with the mathematical comparative analysis we provided in Section 3.3 between PredictiveSkip and OneSkip.

The poor performance of the OneSkip baseline stems from its reactive nature. Because it only skips to the required cycle from other BGs rather than the based on the protocol load (or pipeline utilization), the system's global finality floor is repeatedly dragged down to the straggler's processing time. PredictiveSkip mitigates this by using the BG state information extracted from BFTCompletionProof, which helps to perform optimal skips based on the requirements of the broader pipeline.

Chapter 8

Conclusion

The performance barrier in planetary-scale BFT consensus is primarily driven by network-oblivious consensus designs that fail to account for the hierarchical nature of global network infrastructure. Traditional protocols treat the network as a flat abstraction, leading to redundant communication over expensive inter-continental links and HoL blocking when regional members experience transient congestion. While hierarchical clustering is a known mitigation strategy, existing implementations lack the cross-layer visibility necessary to adapt to the stochastic variance of geo-distributed environments, often resulting in static configurations that either under-utilize bandwidth or inflate latency.

Polaris addresses these challenges by reconceptualizing the coordination fabric as an integrated, state-aware system. Polaris trades off a stronger fault model, assuming a pragmatic limit on the maximum number of faults in each geographic zone, to increase throughput by parallelizing BFT protocol operations. These architectural changes fundamentally shift the system-level implications of geo-distributed deployment. The Optimistic Global Coordination reduces inter-cluster traffic by a factor of $2f + 1$ during failure-free execution, allowing the system to scale horizontally without proportional WAN bandwidth growth. Mathematical and empirical analysis proves that Polaris maintains global liveness even when individual regions are 4x slower than the network average proving the effectiveness of skip mechanism in mitigating stragglers without stalling the faster majority. By treating pipeline depth as a dynamic parameter between network capacity and regional load skew, Polaris prevents the exponential latency spikes characteristic of buffer bloat in static systems.

A realistic future direction for this work involves integrating Polaris with more sophisticated underlying BFT protocols that support native pipelining. While the current

prototype uses a sequential SBFT implementation that imposes a base latency penalty and throughput loss, the hierarchical cross-layer optimizations developed in this thesis are orthogonal to the specific BFT consensus engine employed. Future research could also explore the application of enhanced convergence scoring to more granular participant selection in permissionless environments which can potentially bridge the gap between the high-performance bounds of permissioned clusters and the decentralization requirements of open networks. Exploring alternative communication overlays for global dissemination that could further reduce the bandwidth pressure on individual BGs could be another meaningful addition to this protocol design.

References

[1] 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, 2019.

[2] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Scaling byzantine fault-tolerant replication to wide area networks. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 105–114. IEEE, 2006.

[3] Rita Azzi, Rima Kilany Chamoun, and Maria Sokhn. The power of a blockchain-based supply chain. *Computers & Industrial Engineering*, 135:582–592, 2019.

[4] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

[5] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. Technical report, R3 CEV, 2016.

[6] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, volume 99, pages 173–186, 1999.

[8] Victor Chang, Patricia Baudier, Hui Zhang, Qianwen Xu, Jingqi Zhang, and Mitra Arami. How blockchain can impact financial services–the overview, challenges and recommendations from expert interviewees. *Technological Forecasting and Social Change*, 158:120166, 2020.

[9] Avery Ching, Mo Shaikh, et al. The Aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. Technical report, Aptos Labs, 2022.

[10] Rosanna Cole, Mark Stevenson, and James Aitken. Blockchain technology: implications for operations and supply chain management. *Supply Chain Management: An International Journal*, 24(4):469–483, 2019.

[11] ConsenSys. Quorum whitepaper v0.2, 2016.

[12] Pedro Costa, Marcelo Pasin, Alysson Bessani, and Miguel Correia. Byzantine fault-tolerant MapReduce: Faults are not just crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39. IEEE, 2011.

[13] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless byzantine consensus and its application to blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications*, pages 1–8, 2018.

[14] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün Sirer. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[15] Diem Association. DiemBFT v4: State machine replication in the Diem blockchain. Technical report, Diem Association, 2021.

[16] John R. Douceur. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[17] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.

[18] Pankaj Dutta, Tsan-Ming Choi, Surabhi Somani, and Richa Butala. Blockchain technology in supply chain operations: Applications, challenges and research opportunities. *Transportation Research Part E: Logistics and Transportation Review*, 142:102067, 2020.

[19] V. Enes et al. State-machine replication for planet-scale systems. In *Proceedings of the 15th EuroSys*, pages 1–15, 2020.

[20] etcd-io. etcd: Distributed reliable key-value store. https://github.com/etcd-io/etcd, 2020. Accessed: 2026.

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

[22] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[23] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

[24] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6):868–883, 2020.

[25] Christine V. Helliar, Louise Crawford, Elizabeth French, and Christopher I. Reutzel. Permissionless and permissioned blockchain diffusion. *International Journal of Information Management*, 54:102136, 2020.

[26] Charles Hoskinson. Why we are building Cardano. IOHK Whitepaper, 2017.

[27] Maged N. Kamel Boulos, James T. Wilson, and Kevin A. Clauson. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *International Journal of Health Geographics*, 17(1):25, 2018.

[28] Srinivasan Keshav, Wojciech Golab, Bernard Wong, S. Rizvi, and Sergey Gorbunov. RCanopus: Making Canopus resilient to failures and byzantine faults. *arXiv preprint arXiv:1810.09300*, 2018.

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

[30] Christine Kim. Ethereum 2.0: How it works and why it matters. Coindesk Report, 2020.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58, 2007.

[32] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

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

[34] Linux Foundation. Hyperledger fabric project. https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html, 2023. Accessed: 2026.

[35] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, and Marko Vukolić. XFT: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 485–500, 2016.

[36] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 369–384, 2008.

[37] Gonzalo Martínez, Javier Conde, Pedro Reviriego, José Alberto Hernández, and Paul Reinheimer. Round trip time (RTT) delay in the internet: Analysis and trends. *IEEE Network*, 38(2):280–285, 2023.

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

[39] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Egalitarian Paxos. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–375, 2012.

[40] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[41] R. Neiheiser, M. Matos, and L. Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 35–48, 2021.

[42] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the*

Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 8–17, 1988.

[43] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[44] Qifan Pu, Ankur Haque, et al. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[45] S. Rizvi, B. Wong, and S. Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of ACM CoNEXT*, pages 426–438, 2017.

[46] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 34(3):1156–1190, 2021.

[47] Fabian Schär. Decentralized finance: On blockchain- and smart contract-based financial markets. *Federal Reserve Bank of St. Louis Review*, 103(2):153–174, 2021.

[48] Ayesha Shahnaz, Usman Qamar, and Ayesha Khalid. Using blockchain for electronic health records. *IEEE Access*, 7:147782–147795, 2019.

[49] Joao Sousa and Alysson Bessani. From byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[50] 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*, pages 2705–2718, 2022.

[51] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-BFT: High-throughput robust BFT for decentralized networks. *arXiv preprint arXiv:1906.05552*, 2019.

[52] Philip Treleaven, Richard Gendal Brown, and Danny Yang. Blockchain technology in finance. *Computer*, 50(9):14–17, 2017.

[53] Ashish Vulimiri et al. Global analytics in the face of bandwidth and regulatory constraints. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336, 2015.

[54] Yang Wang, Xiaofei Xing, Peiqian Li, and Guojun Wang. LT-DBFT: A hierarchical blockchain consensus using location and trust in IoT. *IEEE Internet of Things Journal*, 12:19574–19589, 2025.

[55] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl. MaxiNet: Distributed emulation of software-defined networks. In *IFIP Networking 2014 Conference*, pages 1–9, 2014.

[56] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. White paper, Polkadot Network, 2016.

[57] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. Whitepaper, 2018.

[58] 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*, pages 347–356, 2019.