Implementing RSA Accumulators for Asynchronous and Permissionless Reliable Broadcasting

Eric Webb

A dissertation proposal paper submitted in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Cybersecurity Management

College of Computing and Engineering, Nova Southeastern University

5-7-2025

An Abstract of a Dissertation Submitted to Nova Southeastern University

in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Cybersecurity Management

Implementing RSA Accumulators for Asynchronous and Permissionless Reliable Broadcasting

by

Eric B. Webb

April 2025

Asynchronous consensus protocols are essential for decentralized and trustless environments such as decentralized finance (DeFi), supply chain management, and voting systems. These protocols eliminate centralized authority and timing assumptions while improving resilience and security. However as network sizes increase the communication overhead in these systems becomes a bottleneck that limits scalability and efficiency. One notable example is the Aleph protocol, which stands out as one of the first consensus protocols to be both asynchronous and permissionless while providing Byzantine Fault Tolerance. Unlike many existing asynchronous consensus mechanisms Aleph is permissionless in nature and does not rely on a trusted dealer or predefined set of participants making it well-suited for fully decentralized blockchain applications. Its permissionless and asynchronous nature allows it to function without synchronized clocks or fixed membership ensuring greater fault tolerance and robustness in adversarial conditions. These are key properties for blockchain networks that operate in open dynamic environments. Regarding the blockchain trilemma, this design strengthens decentralization and security but also introduces higher communication complexity which impacts scalability. This research acknowledges these trade-offs and aims to enhance asynchronous permissionless consensus by reducing communication overhead making it more scalable for decentralized applications.

Aleph’s consensus mechanism relies on a Chain Reliable Broadcast (ch-RBC) protocol for message dissemination. Despite its advantages the ch-RBC suffers from quadratic communication complexity leading to substantial overhead in large networks. This dissertation proposes an enhancement to Aleph’s ch-RBC protocol by replacing its Merkle tree-based transaction validation with Rivest-Shamir-Adleman (RSA) accumulators. The research aims to reduce the protocol’s communication complexity from O(Tr + N² log N) to O(Tr + N²), thereby improving performance and scalability while maintaining security guarantees. RSA accumulators offer compact and constant-sized proofs for transaction validation which can significantly reduce message complexity and bandwidth consumption.

The study employs an experimental approach by implementing both the original Merkle tree-based ch-RBC and the RSA accumulator-based version. Simulations will be conducted using Rust-based implementations on AWS EC2 instances with network sizes ranging from 32 to 104 nodes and transaction batch sizes scaling up to 131,072. Key performance metrics will include transaction throughput, latency, communication overhead, and resource utilization.

Eric B. Webb

These metrics will be analyzed to evaluate the impact of RSA accumulators on consensus efficiency.

This work aims to enhance performance as part of the broader goal of making decentralized networks more scalable and suitable for real-world applications. This work will explore the feasibility of RSA accumulators in permissionless asynchronous consensus protocols providing insights into their potential to improve blockchain scalability for high-performance and large-scale decentralized applications.

**Table of Contents**

**Chapters**

1. **Introduction** 1

1.1 Background 1

1.2 Problem Statement 4

1.3 Dissertation Goal 6

1.4 Relevance and Significance 8

1.5 Barriers and Issues 9

1.6 Assumptions, Limitations, and Delimitations 10

1.7 Definition of Terms 11

1.8 List of Acronyms 11

1.9 Summary 11

1. **Review of the Literature** 13

2.1 Overview 13

2.2 Justification of the Criteria for What is Included and Excluded as Part of the Review 13

2.3 Identification of What has Been Done Before Including the Strengths and Weaknesses

of Existing Studies 14

2.4 Identification of Gaps in Literature 16

2.5 Synthesis of the Literature and Perspective 16

2.6 Summary 17

1. **Methodology** 19

3.1 Overview of Research Methodology and Design 19

3.2 Specific Research Methods to be Employed 19

3.3 High-Level Methodology Stages 20

3.4 Stage 1: Design & Implementation of Merkle-Based Aleph 21

3.5 Stage 2: Design & Implementation of RSA-Based Aleph 31

3.6 Stage 3: Deployment & Validation of Each Variant 42

3.7 Stage 4: Evaluation & Analysis 45

3.8 Proposed Sample 49

3.9 Resource Requirements Summary 56

3.10 Summary 57

References 58

**Tables**

1. Summary of Research Method Inheritance 20
2. Validation Objectives Per Variant 44
3. Aleph ch-RBC Experiment Matrix 50

**Graphs**

1. HBFT Figure 6: TPS vs Batch Size 31
2. HBFT Figure 7: TPS vs Latency 32
3. ABFT Figure 1: Nodes vs Latency 33
4. Throughput vs BatchSize 52
5. HBFT Figure 6: TPS vs Batch Size 53
6. TPS vs Latency by Node Config 54
7. HBFT Figure 7: TPS vs Latency 55
8. Latency vs Node Count (RSA vs Merkle) 56
9. ABFT Figure 1 : Nodes vs Latency 56

**Chapter 1**

**Introduction**

**Background**

The presented research focuses on asynchronous consensus and permissionless systems that eliminate the need for centralized authority and timing assumptions. These systems improve resilience and security in applications and thus are essential for decentralized trustless environments such as DeFi, supply chain management, and voting systems (Hajian Berenjestanaki et al., 2024). Compared to synchronous permissioned models, asynchronous permissionless systems enhance decentralization and security by allowing open participation and tolerating unpredictable network delays making them more resilient in adversarial conditions. However, this comes at the cost of higher communication overhead which limits scalability as network size increases. While synchronous permissioned approaches optimize for efficiency through structured participation and timing coordination, they do so by sacrificing decentralization and increasing reliance on trusted nodes. This research acknowledges the trade-offs in the blockchain trilemma and seeks to mitigate the scalability limitations of asynchronous permissionless consensus by reducing communication complexity and thereby making it more viable for large-scale decentralized applications.

The Aleph permissionless asynchronous protocol’s implementation of a RBC protocol called ch-RBC degrades due to the communication complexity that increases quadratically with the number of nodes. Thus, to address this performance degradation an enhancement is proposed in this work by integrating RSA accumulators into Alephs ch-RBC to reduce communication complexity and improve scalability.

Decentralized systems in asynchronous and permissionless environments face significant scalability challenges particularly due to the communication complexity inherent in RBC protocols that are vital for achieving consensus. Traditional RBC designs often based on Merkle trees for transaction validation exhibit quadratic communication complexity as the number of participants grows (Miller et al., 2016; Gągol et al., 2019). This inefficiency places substantial strain on network resources such as bandwidth and computational capacity which hinders performance and scalability.

These challenges are especially pronounced in critical applications. In DeFi for example high transaction throughput and robust security are essential to maintaining liquidity and trust (Singh et al., 2022). Similarly traditional centralized supply chain systems often suffer from bottlenecks and opacity. In contrast consensus protocols enable stakeholders to access immutable and transparent transaction records that foster trust and operational efficiency (Manzoor et al., 2022). In voting systems where integrity and accuracy are paramount, consensus mechanisms ensure each vote is securely and verifiably recorded such that it safeguards the electoral process (Hajian Berenjestanaki et al., 2024).

The importance of addressing these limitations highlights the value of asynchronous permissionless systems. By tackling the blockchain trilemma of balancing scalability, security, and decentralization these systems enable greater efficiency and robustness in real-world applications (Principato et al., 2023).

Asynchronous consensus systems are particularly important because they do not require all participants to proceed in lockstep and have no timing assumption. The lack of dependency on synchronized clocks among nodes enhances the robustness of the network against delays and latency (Gao et al., 2022). In contrast, synchronous systems require all nodes to operate within a controlled timing regiment and can be a significant limitation in decentralized networks where communication delays are unpredictable (Miller et al., 2016).

Aleph is one of the first asynchronous consensus protocols to operate in a permissionless setting and eliminates the need for Distributed Key Generation (DKG) or a trusted dealer (Guo et al., 2022). Its innovative ch-RBC protocol enables decentralized operations with open participation and no timing assumptions making it a valuable solution for asynchronous decentralized applications. However, Aleph’s reliance on Merkle trees introduces significant communication overhead as the network scales. The quadratic complexity of message verification and broadcasting results in higher resource consumption and reduced efficiency. These scalability concerns limit its performance in larger networks where robust and efficient operations are critical. Addressing these challenges are essential to improving Aleph’s scalability and overall effectiveness in asynchronous and permissionless environments.

RBC protocols play a critical role in ensuring the correct and consistent delivery of messages in distributed systems. However, as the inherent message complexity in traditional RBC protocols such as those utilized by HoneyBadgerBFT, Dumbo, and BEAT presents significant scalability challenges as networks expand (Miller et al., 2016; Duan et al., 2018; Guo et al., 2020). These challenges are particularly pronounced in asynchronous environments where the growth in communication can lead to bottlenecks and reduce the overall efficiency and effectiveness of the network (Miller et al., 2016; Guo et al., 2020). Synchronous permissioned protocols are positioned in the blockchain trilemma to be generally more scalable due to their reliance on a fixed validator set and coordinated timing assumptions, asynchronous permissionless protocols prioritize decentralization and security in the trilemma by allowing open participation and tolerating unpredictable network delays and adversarial conditions. While permissioned approaches achieve higher efficiency by limiting participants and enforcing strict timing they sacrifice decentralization by requiring a trusted setup. In contrast permissionless systems strengthen decentralization and resilience but suffer from increased communication complexity making scalability a key challenge. As the number of participants increases these limitations highlight the need for more efficient communication mechanisms to maintain robust fault tolerance without sacrificing performance (Duan et al., 2018).

The research focus in this paper is the integration of RSA accumulators into Aleph's ch-RBC protocol to address the scalability and communication overhead challenges inherent in its Merkle tree-based design. By reducing the communication complexity this approach aims to enhance the protocol's performance and scalability in asynchronous and permissionless environments. The research contributes to the broader goal of improving decentralized networks by offering a more efficient asynchronous permissionless consensus mechanism lowering resource consumption and maintaining robustness. These improvements are particularly valuable for applications in DeFi, supply chain management, and secure voting systems which demand high efficiency, transparency, and fault tolerance.

**Problem Statement**

Aleph’s ch-RBC protocol is a critical component of its asynchronous and permissionless consensus layer but suffers from significant communication complexity as network size increases. This limitation is detailed in the Aleph paper where the authors demonstrate that the ch-RBC’s communication complexity scales as O(Tr + N² log N), where Tr represents the total number of transactions and N is the number of participating nodes (Gągol et al., 2019). This complexity stems from the use of Merkle trees that ensure consistency and fault tolerance but imposes substantial overhead due to the logarithmic growth in validation costs and the quadratic growth in node-to-node message exchanges.

As the number of participants approaches 1,000 nodes the communication overhead in Aleph’s ch-RBC escalates dramatically reaching nearly 10 million message exchanges. This overhead arises primarily during the propose, prevote, and commit phases which requires each node to communicate with every other node multiple times. Such intensive messaging places a heavy burden on network bandwidth and node processing capabilities creating bottlenecks that degrade performance. These challenges are especially detrimental in permissionless environments where the number of nodes can grow unpredictably (Gągol et al., 2019; Duan et al., 2018).

Aleph’s high communication complexity not only affects throughput but also limits scalability making the protocol less suitable for real-world applications such as DeFi and supply chain management. Both domains demand consensus protocols capable of handling large transaction volumes while ensuring performance and reliability. For instance, DeFi platforms require high throughput to maintain liquidity and prevent market instability while supply chain networks rely on transparency and efficiency to track and verify goods across multiple stakeholders (Zhou et al., 2020; Manzoor et al., 2022). In Aleph’s current state the communication overhead hampers its ability to scale effectively in such scenarios.

When compared to other consensus protocols, Aleph’s O(Tr + N² log N) complexity is particularly high. For example, protocols employing threshold cryptography or gossip-based dissemination achieve lower complexities such as O(N²) (Guo et al., 2020). These techniques reduce the need for direct communication between all nodes and instead rely on smaller subsets of nodes or probabilistic communication strategies to disseminate information. In contrast, Aleph’s design necessitates extensive message exchanges among all participants and leads to inefficiencies in large-scale networks.

Among permissioned asynchronous protocols Aleph’s communication overhead is notable. HoneyBadgerBFT and Dumbo face similar challenges with quadratic message complexities resulting in bottlenecks as network size increases (Miller et al., 2016; Duan et al., 2018). While synchronous or semi-synchronous protocols such as Bitcoin’s Nakamoto Consensus and Hashgraph exhibit lower complexities O(N). These systems benefit from their less stringent assumptions and controlled environments (Gencer et al., 2018; Baird & Luykx, 2020). In contrast, Aleph’s fully asynchronous and permissionless design makes it inherently more complex and challenging to optimize (Guo et al., 2020).

The extensive message exchanges required by Aleph’s ch-RBC result in network congestion, increased latency, and higher resource consumption making it unsuitable for large-scale deployment. Addressing this communication complexity is crucial to improving the protocol’s scalability and ensuring its viability for real-world decentralized applications.

**Dissertation Goal**

The research aims to implement a RBC protocol that incorporates RSA accumulators as a replacement for the Merkle tree validation currently used in Aleph’s ch-RBC. Aleph’s current protocol has a communication complexity of O(Tr + N² log N), where Tr represents the total number of transactions and N is the number of participating nodes. By integrating RSA accumulators the goal is to reduce this complexity to O(Tr + N²) thereby addressing the scalability and efficiency challenges inherent in the existing design. Prior studies have shown that RSA accumulators are effective in reducing message complexity and validation overhead making them a promising approach to improving the performance of Aleph’s ch-RBC protocol in permissionless networks (Gągol et al., 2019; Hussein & Al-Gailani, 2022; Reddy, 2021).

The research involves implementing Aleph’s ch-RBC proof in a controlled programming environment and replacing the Merkle tree structure with an RSA accumulator-based design. This process includes developing both the Merkle tree-based control protocol and the RSA accumulator-based protocol and comparing their performance in identical simulated environments. To ensure reliability the study will follow established methodologies for benchmarking asynchronous consensus protocols including HoneyBadgerBFT, Dumbo, and ABFT and will simulate diverse network conditions and workloads to evaluate performance.

To validate the effectiveness of the RSA-based RBC protocol the research will measure key performance metrics including throughput, latency, communication overhead, and resource utilization. Throughput will be used to evaluate the protocol’s capacity to handle large transaction volumes by measuring the number of transactions processed per second. Latency will provide insight into the system’s responsiveness by assessing the time taken from transaction submission to its finalization. Communication overhead will be analyzed by quantifying the total number of messages exchanged during the consensus process which reflects the efficiency of the protocol’s design. Resource utilization will include CPU and memory consumption and will help determine the protocol’s ability to operate efficiently under varying workloads. These metrics are standard in asynchronous consensus protocol evaluations and have been used in studies of HoneyBadgerBFT, BEAT, and Dumbo (Miller et al., 2016; Duan et al., 2018; Guo et al., 2020).

Scalability will be demonstrated by observing how these metrics behave as the number of nodes and transaction volumes increase. This work will explore whether RSA accumulators can reduce communication overhead and improve throughput and latency in larger network environments compared to Merkle tree implementations. The analysis aims to demonstrate the potential scalability benefits of the proposed approach without compromising performance.

By isolating the effects of RSA accumulators on communication complexity the study aims to provide a detailed performance comparison between the two designs. The goal is to provide an implementation of RSA accumulators that is targeted to enhance scalability and overall performance in permissionless asynchronous networks. The potential reduction in Aleph’s communication overhead may improve the value of the protocol for real-world decentralized applications in areas such as decentralized finance, supply chain management, and other domains requiring high scalability, transparency, and efficiency

**Relevance and Significance**

The scalability of permissionless consensus protocols is a critical concern for decentralized networks particularly in high-transaction environments such as DeFi, supply chain management, and secure voting systems where high throughput and low latency are essential. Traditionally scalability within the blockchain trilemma has favored synchronous and permissioned models due to their more predictable communication overhead and tighter coordination. However Aleph’s ch-RBC reliance on Merkle tree-based validation limits its ability to scale effectively. As the number of nodes grows the communication complexity increases leading to network congestion and reduced efficiency particularly in large-scale applications.

This study fundamentally shifts the scalability balance by integrating RSA accumulators as a replacement for Merkle trees and addressing these bottlenecks directly. Unlike Merkle trees which require extensive proof propagation and verification, RSA accumulators allow for compact non-interactive membership proofs significantly reducing message complexity. This reduction in communication overhead enhances throughput making permissionless asynchronous consensus protocols more scalable without the usual trade-offs in security and decentralization.

The implications are broad such as in DeFi improved throughput ensures market stability and liquidity. In supply chain management reduced validation overhead enhances transparency and efficiency. While in secure voting systems lower latency enables faster and more reliable vote validation at scale. By demonstrating that permissionless asynchronous consensus can achieve competitive scalability through improved communication complexity this research challenges the traditional trilemma assumptions. It advances the field of asynchronous permissionless consensus mechanisms and paves the way for a new position in the blockchain trilemma that allows increased scalability while maintaining security and decentralization.

**Barriers and Issues**

Implementing and validating the original Merkle tree-based and RSA accumulator-based RBC protocols presents several challenges that must be carefully addressed to ensure its feasibility and effectiveness. One of the primary challenges is algorithmic complexity as integrating RSA accumulators into the ch-RBC protocol requires maintaining security and correctness while only modifying the transaction validation process. Another significant barrier is experimental validation which involves deploying large-scale simulations to empirically measure the performance improvements of RSA accumulators compared to the existing Merkle tree-based implementation. Ensuring that these simulations accurately reflect decentralized environments is crucial for validating the proposed approach. Additionally computational overhead is a concern as the benefits of reducing message complexity through RSA accumulators must be balanced against the potential increase in cryptographic verification costs. Addressing these trade-offs will involve using RSA operations to maintain efficiency without introducing new bottlenecks. These barriers will be systematically tackled throughout the research by implementing rigorous testing, performance benchmarking, and security validation to demonstrate the practical applicability of the proposed solution in permissionless asynchronous consensus protocols.

**Assumptions, Limitations and Delimitations**

This study assumes a BFT network where the number of malicious nodes does not exceed the tolerated threshold. The analysis focuses on reducing communication complexity rather than handling missing data caused by byzantine nodes. The research is limited to evaluating Aleph’s ch-RBC in a controlled simulated environment meaning that deployment factors such as adversarial behavior and unpredictable network conditions are not explicitly tested. Additionally, the study is restricted to analyzing the impact of RSA accumulators within Aleph’s ch-RBC protocol and does not compare them against alternative data structures like vector commitments. These assumptions, limitations, and delimitations define the scope of the research and ensure that the study remains focused on addressing communication complexity while maintaining consistency in evaluation.

*Definition of Terms*

* Reliable Broadcast (RBC): A consensus mechanism ensuring that all nodes in a distributed system receive and agree on the same messages.
* Merkle Tree: A cryptographic data structure used for efficient and secure transaction validation.
* RSA Accumulator: A cryptographic primitive enabling compact proofs of set membership, reducing communication complexity.
* Asynchronous Consensus: A consensus model where nodes do not rely on synchronized clocks to reach agreement.
* Permissionless Network: A decentralized system where participants can join and leave freely without requiring pre-approval.

*List of Acronyms*

* HBFT - The HoneyBadgerBFT protocol.
* ABFT - An Asynchronous BFT protocol named eponymously
* BEAT - The Broadcast, Encryption, Agreement, and Threshold protocol
* RBC – Reliable Broadcast
* BFT – Byzantine Fault Tolerance
* DeFi – Decentralized Finance
* ch-RBC – Chain Reliable Broadcast
* RSA – Rivest–Shamir–Adleman

**Summary**

This chapter introduced the motivation for the study outlining the challenges associated with Aleph’s ch-RBC protocol in terms of scalability and communication complexity. The problem statement identified the inefficiencies of Merkle tree-based transaction validation while the research goal proposed RSA accumulators as a solution to reduce message complexity. The research questions and hypotheses were formulated to guide the investigation, and the significance of the study was highlighted in the context of real-world applications such as DeFi, supply chain management, and voting systems.

Furthermore, this study positions Aleph within an asynchronous and permissionless more scalable paradigm enhancing consensus scalability without sacrificing decentralization or security. By leveraging RSA accumulators, the protocol achieves a lower communication overhead enabling higher transaction throughput while maintaining Byzantine Fault Tolerance. This shift enhances scalability in fully asynchronous networks making permissionless consensus more viable for large-scale applications.

The next chapter will review the existing literature on asynchronous consensus mechanisms, communication complexity in RBC protocols, and the role of cryptographic accumulators in optimizing blockchain scalability. This literature review will provide the necessary theoretical foundation for understanding the proposed improvements and situating them within the broader research landscape.

**Chapter 2**

**Review of the Literature**

**Overview**

This chapter reviews the existing research on asynchronous consensus and permissionless systems focusing on the challenges of communication complexity in RBC protocols and the potential improvements offered by RSA accumulators. It examines the role of RBC in ensuring message reliability in decentralized networks, the communication overhead in current RBC implementations, and the use of Merkle trees for transaction verification. The chapter also explores RSA accumulators as an alternative discussing their cryptographic properties and how they can reduce communication complexity in blockchain consensus. A comparative analysis of consensus protocols including Aleph, HoneyBadgerBFT, BEAT, Dumbo, and ABFT others highlights their scalability and efficiency. Finally, the review identifies gaps in the literature outlining the limitations of existing approaches and areas for further research. This foundation supports the study’s goal of modifying Aleph’s ch-RBC protocol by integrating RSA accumulators to enhance scalability and efficiency.

**Justification of the Criteria for What is Included and Excluded as Part of the Review**

This literature review focuses on relevant sources by selecting studies that directly relate to asynchronous consensus and permissionless systems. The review includes peer-reviewed journal articles and conference papers that analyze the communication complexity of RBC protocols in decentralized networks, research on Merkle trees and RSA accumulators in blockchain consensus, and performance evaluations of alternative transaction validation methods. Studies were excluded if they focused only on synchronous or permissioned consensus mechanisms, lacked quantitative assessments of communication complexity, or were outdated and replaced by more recent findings. This selection ensures that the review aligns with the research objectives and supports the development of the proposed methodology*.*

***Identification of What has Been Done Before Including the Strengths and Weaknesses of***

***Existing Studies***

RBC protocols are crucial for maintaining message consistency in asynchronous networks, but existing implementations face scalability challenges due to high communication complexity. HoneyBadgerBFT (Miller et al., 2016) was one of the first practical asynchronous BFT protocols using threshold encryption to achieve reliability but its quadratic message complexity O(N²) limits its scalability. BEAT (Guo et al., 2020) introduced batching to reduce message complexity while maintaining fault tolerance yet it still experiences quadratic growth in communication. Dumbo (Zhou et al., 2020) improved upon HoneyBadgerBFT by incorporating erasure coding, reducing redundant message transmissions and enhancing bandwidth efficiency while retaining BFT. ABFT (Knudsen et al., 2021) is designed as a high-performance asynchronous BFT protocol but is primarily suited for permissioned environments due to its reliance on predefined validator sets and threshold ECDSA without a Distributed Key Generator (DKG). While it improves scalability through optimized erasure coding and cryptographic precomputations it still requires further modifications to support dynamic membership and true permissionless participation. Aleph (Gągol et al., 2019) removed the need for a trusted dealer making it one of the first permissionless asynchronous consensus protocols but its reliance on Merkle tree-based validation results in O(Tr + N² log N) communication complexity restricts its ability to scale efficiently.

The primary issue affecting RBC scalability is message complexity as traditional designs require extensive node-to-node communication leading to quadratic growth in message exchanges. Merkle trees used for transaction validation in Aleph’s ch-RBC introduce additional logarithmic complexity by further increasing verification overhead. As networks expand this overhead worsens latency and finality time making it difficult to sustain high transaction throughput.

Merkle trees are effective for proof validation but have limitations in consensus protocols. Each transaction verification requires multiple hash operations increasing communication overhead. Additionally storing and maintaining Merkle roots for every batch adds storage complexity. When combined with node-to-node communication this results in quadratic message complexity making large-scale deployment inefficient.

RSA accumulators provide an alternative that significantly reduces message complexity by enabling constant-sized membership proofs instead of exchanging multiple hash values. Unlike Merkle trees, RSA accumulators allow nodes to verify transactions without needing multiple proofs and potentially reducing communication complexity from O(Tr + N² log N) to O(Tr + N²). However, RSA accumulators require modular exponentiation operations which introduce computational overhead though recent optimizations have improved their efficiency. Empirical studies (Reddy, 2021; Hussein & Al-Gailani, 2022) show that RSA accumulators reduce verification latency and bandwidth usage making them a promising solution for improving RBC protocols in asynchronous permissionless networks.

**Identification of the gaps in the literature**

Despite advancements in optimizing asynchronous consensus protocols several gaps remain in the literature. RSA accumulators have been studied primarily for proof-of-membership applications but their impact on RBC communication complexity has not been fully explored. Additionally, performance benchmarks for RSA accumulators in large-scale networks with high transaction throughput are lacking making it difficult to assess their scalability in real-world decentralized systems. Furthermore, alternative set membership proofs such as vector commitments and polynomial commitments (Belling, Soleimanian, & Ursu, 2024) offer different validation approaches but their trade-offs compared to RSA accumulators have not been thoroughly analyzed. Addressing these gaps will help determine the feasibility of RSA accumulators for improving the efficiency of asynchronous consensus protocols.

**Synthesis of the literature**

The reviewed literature highlights the scalability limitations of existing RBC protocols, particularly in asynchronous and permissionless environments. HBFT and BEAT, both permissioned, and Aleph, a permissionless protocol, achieve BFT asynchronously, they struggle with high communication complexity (Miller et al., 2016; Duan et al., 2018; Gągol et al., 2019; Guo et al., 2020). The reliance on Merkle trees for transaction validation in Aleph’s ch-RBC further increases this complexity adding logarithmic verification overhead to an already quadratic message cost. As a result, these protocols face significant inefficiencies as network size grows. RSA accumulators have emerged as a possible alternative offering constant-sized proofs and batch verification which could drastically reduce communication complexity. However, existing research on RSA accumulators has primarily focused on cryptographic applications rather than their direct integration into consensus mechanisms. This study bridges that gap by implementing RSA accumulators in Aleph’s ch-RBC and empirically evaluating their impact on communication complexity, throughput, and scalability. The research contributes new benchmarks and performance insights while positioning RSA accumulators as a viable solution for improving asynchronous permissionless consensus.

The broader perspective on asynchronous consensus reveals a fundamental trade-off between decentralization, security, and scalability, often referred to as the blockchain trilemma (Principato et al., 2023). RBC protocols optimize for security and correctness but struggle with message overhead, making scalability a challenge. While Merkle trees have traditionally been used for transaction validation their computational and communication costs make them increasingly impractical in high-throughput environments. The key takeaway from this review is that reducing communication complexity rather than focusing solely on computation or storage efficiency is the most effective way to enhance RBC scalability. This shift in focus suggests that consensus research should move toward cryptographic techniques like RSA accumulators which allow for lightweight validation without increasing message complexity. By incorporating these findings into Aleph’s ch-RBC this study challenges traditional assumptions about RBC design and introduces a novel approach that balances security, decentralization, and performance, paving the way for more scalable permissionless consensus mechanisms.

**Summary of the Chapter**

This chapter provided an overview of existing research on asynchronous consensus and RBC protocols and identified the limitations of current approaches and the potential improvements offered by RSA accumulators. The next chapter will detail the research methodology, including experimental setup, data collection techniques, and performance evaluation criteria.

**Chapter 3**

**Methodology**

**Overview of Research Methodology and Design**

This study employs an experimental research design to evaluate the impact of RSA accumulators on the communication complexity and scalability of Aleph’s ch-RBC protocol. The research follows a comparative approach by implementing and testing two versions of ch-RBC: the existing Merkle tree-based ch-RBC protocol, which uses Merkle trees for transaction validation, and a modified version where Merkle trees are replaced with RSA accumulators for transaction verification. Both implementations will be deployed and tested in a simulated AWS-based environment allowing for controlled experimentation under varying network conditions. The study will measure specific performance metrics including average transaction throughput (TPS) across nodes, communication overhead (messages per node in the final round), CPU and memory utilization, and total consensus latency, to evaluate whether RSA accumulators can reduce communication complexity and improve scalability compared to the baseline Merkle tree-based implementation.

**Specific Research Methods to be Employed**

This document presents the complete design, implementation, deployment, and analysis methodology for evaluating the Aleph ch-RBC protocol with two transaction validation mechanisms: Merkle trees and RSA accumulators. The goal is to compare their impact on scalability and communication overhead in asynchronous permissionless networks.

**High-Level Methodology Stages**

* *Stage 1: Merkle-Based Aleph Implementation*  
   - Develop, validate, and test the Aleph protocol using Merkle tree-based transaction validation. This stage follows the design outlined in the original Aleph protocol and the HBFT, where Merkle proofs are used to ensure data availability and validation efficiency. The modularity of the Aleph design, inherited from prior asynchronous BFT protocols like BEAT and DUMBO, allows for straightforward integration of cryptographic validation mechanisms.
* *Stage 2: RSA-Based Aleph Implementation*  
   - Modify and extend the protocol to replace Merkle tree validation with RSA accumulators. This approach is informed by cryptographic research on succinct proofs, such as that by Hussein & Al-Gailani (2022) and Reddy (2021), and allows for constant-sized proof generation and verification. The modular validation structure of Aleph, inspired by HBFT and BEAT, supports this transition without altering the core consensus or DAG-based commitment logic.
* *Stage 3: Deployment & Runtime Execution*  
   - Provision a cloud-based testbed using AWS CDK to deploy both protocol versions. The deployment architecture is based on experimental setups from HBFT and similar ABFT systems such as BEAT and DUMBO using EC2 instances within a VPC to replicate an asynchronous communication environment. This ensures consistent runtime conditions for comparative analysis.
* *Stage 4: Experimental Validation & Analysis*  
   - Design and execute experiments to measure key performance metrics, including transaction throughput, latency, communication overhead, and resource utilization. These metrics are aligned with those used in BEAT, HBFT, and the original Aleph protocol, ensuring comparability with prior evaluations. The results will highlight trade-offs between Merkle and RSA validation mechanisms and their impact on scalability in asynchronous networks.

*Summary of Research Method Inheritance*

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Inherited From** | **Impact on This Methodology** |
| Modular validation design | HBFT and Aleph | Enables flexible swapping of Merkle trees with RSA accumulators |
| Experimental performance metrics | BEAT, HBFT, DUMBO, and ABFT | Standardized evaluation of throughput, latency, and communication |
| DAG + ch-RBC round structure | Aleph | Maintains asynchronous commitment model for fair comparison |
| Use of RSA accumulators | Cryptographic literature Hussein & Al-Gailani (2022); Reddy (2021) | Supports constant-sized proofs for improved scalability |
| Deployment architecture | HBFT, BEAT, ABFT, and DUMBO’s EC2 testbed | Guides reproducible AWS EC2-based deployment for evaluation |

**Stage 1: Design & Implementation of Merkle-Based Aleph**

*1.1 Architecture Overview*

* Language: Rust.
* Consensus Validation: Merkle Tree + Reed-Solomon Erasure Coding.
* Communication Protocol: JSON over HTTP (port 30333).
* Deployment Environment: Distributed EC2 network (VPC).
* Logging: tracing + chrono for structured logs and timestamping.
* Storage: File-based JSON output for finalized DAG and a text file for logs that sync to a S3 bucket.

*1.2 System Components*

Merkle Based Core Protocol Engine

* src::aleph\_rbc: Drives the three-round protocol: Propose, Prevote, and Commit.Processes messages from other nodes and updates the local DAG. Triggers Merkle proof verification and quorum-based finalization.

*Creating Transaction Data Algorithm*

|  |
| --- |
| Function CreateTransactionData(txs: Vec<Vec<u8>>, k: usize, n: usize)  Input:  - txs: List of raw transactions  - k: Number of data shards  - n: Total number of shards  Output:  - List of (shards: Vec<Vec<u8>>, root: [u8; 32], proofs: Vec<Vec<[u8; 32]>>)  Steps:  results = []  For each tx in txs:  Pad tx to fixed length (e.g., 250 bytes) → padded\_tx  Apply Reed-Solomon encoding to padded\_tx with parameters (k, n) →  → Shards: Vec<Vec<u8>>  For each shard in shards:  Compute SHA-256 hash → shard\_hashes: Vec<[u8; 32]>  Build Merkle tree from shard\_hashes → root: [u8; 32]  For j in 0 to k - 1:  Generate Merkle inclusion proof for shard j → proof\_j:Vec<[u8; 32]>  Append (shards, root, [proof\_0, ..., proof\_k-1]) to results  Return results |

*Construct Merkle Root Algorithm*

|  |
| --- |
| Function ConstructMerkleRoot(leaves: Vec<[u8; 32]>)  Input:  - leaves: List of leaf hashes for shards  Output:  - root: Merkle root hash ([u8; 32])  Steps:  If leaves is empty:  Return [0u8; 32]  current\_level = leaves  While current\_level.length > 1:  If current\_level has odd length:  Duplicate last hash to make even  next\_level = []  For i from 0 to current\_level.length / 2 - 1:  left = current\_level[2\*i]  right = current\_level[2\*i + 1]  parent = SHA256(left || right)  next\_level.append(parent)  current\_level = next\_level  Return current\_level[0] |

*Generate Merkle Proof Algorithm*

|  |
| --- |
| Function GenerateMerkleProof(leaves: Vec<[u8; 32]>, index: usize)  Input:  - leaves: List of leaf hashes for shards  - index: Index of the leaf to prove  Output:  - proof: List of sibling hashes ([u8; 32]) required to reconstruct the Merkle root  Steps:  levels = [leaves]  current = leaves  While current.length > 1:  If current has odd length:  Duplicate last hash to make even  next\_level = []  For i from 0 to current.length / 2 - 1:  left = current[2\*i]  right = current[2\*i + 1]  parent = SHA256(left || right)  next\_level.append(parent)  levels.append(next\_level)  current = next\_level  proof = []  current\_index = index  For each level in levels[:-1]: // exclude root level  sibling\_index = current\_index ^ 1  If sibling\_index < level.length:  proof.append(level[sibling\_index])  current\_index = current\_index // 2  Return proof |

*Verify Merkle Proof Algorithm*

|  |
| --- |
| Function VerifyMerkleProof(leaf\_hash: [u8; 32], proof: Vec<[u8; 32]>, index: usize, expected\_root: [u8; 32])  Input:  - leaf\_hash: Hash of the shard to verify  - proof: List of sibling hashes from the Merkle tree  - index: Original index of the leaf  - expected\_root: The Merkle root to compare against  Output:  - Boolean: True if proof is valid, else False  Steps:  current\_hash = leaf\_hash  For each sibling\_hash in proof:  If index % 2 == 0:  current\_hash = SHA256(current\_hash || sibling\_hash)  Else:  current\_hash = SHA256(sibling\_hash || current\_hash)  index = index // 2  Return current\_hash == expected\_root |

Inter-node Communication

* controllers::api\_routes: Exposes endpoints for message passing between nodes using JSON over HTTP (port 30333).

*Proposal Message Format -* Contains transaction shards, corresponding Merkle proofs, and the root hash, allowing nodes to verify data integrity before proceeding to prevote.

|  |
| --- |
| {  "round": 0,  "proposer\_id": 2,  "transactions": [  {  "shards": ["<base64-encoded shard>", "..."],  "proofs": [["<base64-encoded hash>", "..."]],  "merkle\_root": "<hex-encoded root>"  }  ]  } |

*Prevote Message Format -* Signals a node's support for a proposed Merkle root after verifying sufficient shards and proofs to reconstruct and validate the transaction batch.

|  |
| --- |
| {  "round": 0,  "sender\_id": 1,  "transaction\_index": 0,  "shard\_index": 0,  "shard\_data": "<base64>",  "proof": ["<base64>", "..."],  "merkle\_root": "<hex>"  } |

*Commit Message Format -* Confirms that a node has received a quorum of matching prevotes for a proposal and is finalizing the round by committing the Merkle root to the DAG.

|  |
| --- |
| {  "round": 0,  "sender\_id": 1,  "transaction\_index": 0,  "shard\_index": 0,  "shard\_data": "<base64>",  "proof": ["<base64>", "..."],  "merkle\_root": "<hex>"  } |

DAG Utilities

* Utils::dag\_utils: Offers DAG size checks, round synchronization enforcement, and parent availability verification.

Key Components

* CheckDagSize(dag)

|  |
| --- |
| Ensures |dag| ≤ MAX\_SIZE, where MAX\_SIZE is a tunable memory threshold. |

* VerifyParentsExist(round)

|  |
| --- |
| ∀ parent ∈ round.parents:  parent ∈ DAG.finalized |

* EnforceRoundSync(current, msg\_round)

|  |
| --- |
| if msg\_round > current + 1:  discard(msg) |

Merkle Proof Integration

* Finalized DAG entries include: MerkleRoot, ShardIndex, and Proof[].
* Verified using standard Merkle inclusion proof logic

Merkle Proof Verification Algorithm

|  |
| --- |
| Let h\_0 be the SHA256 of the shard. Given sibling hashes h₁, h₂, ..., h\_d;  MerkleRoot = SHA256(...SHA256(SHA256(h\_0 || h₁) || h₂)...) |

The node computes this recursively and checks

|  |
| --- |
| computed\_root == MerkleRoot\_from\_Proposer |

Message Processing Engine

* processors::priority\_queue: Implements a BinaryHeap-based queue to enforce message priority:

Ordering variant:

|  |
| --- |
| RoundFinalized > Commit > Prevote > Proposal |

Implemented via enum discriminants and heap-based min-heap inversion.

Dropping Logic:

|  |
| --- |
| if msg.round < current\_finalized:  discard(msg) |

* processors::rbc\_processor: Consumes prioritized messages, handles round logic, quorum tracking, and initiates rebroadcasts. Manages pruning of outdated messages and ensures round progression.

Algorithmic Flow

|  |
| --- |
| upon\_receive(msg):  match msg.type:  Proposal => handle\_propose(msg)  Prevote => handle\_prevote(msg)  Commit => handle\_commit(msg) |

Merkle Verification (Proof)

Each shard s\_j comes with a Merkle root R and proof path P.

|  |
| --- |
| verify\_merkle(s\_j, P, R) ⇒ SHA256\_tree(s\_j, P) == R |

This must hold before accepting the proposal into the DAG.

Quorum and Round Finalization

Progresses once:

|  |
| --- |
| |prevotes| ≥ 2f + 1 ∧ |commits| ≥ 2f + 1 |

On Commit Quorum:

|  |
| --- |
| DAG.insert(round\_id, MerkleRoot, shards) |

Message Tracking & Quorum Management

* proposal\_tracker: Tracks unique proposals; quorum is N - f.

Quorum Rule:

|  |
| --- |
| quorum\_met ⇐ |unique\_proposers| ≥ N - f |

Merkle Integrity Enforcement:

|  |
| --- |
| ∀ shard s\_j with proof P\_j:  SHA256\_tree(s\_j, P\_j) == MerkleRoot |

* quorum\_votes: Tracks prevote messages; quorum is 2f + 1.

Threshold Logic:

|  |
| --- |
| if |prevotes| ≥ 2f + 1:  send\_commit(MerkleRoot) |

Proof Soundness Guarantee:

|  |
| --- |
| ∀ s\_j ∈ shards:  SHA256\_tree(s\_j, P\_j) == MerkleRoot |

* commit\_tracker: Tracks commit messages; quorum is 2f + 1.

Commit Quorum:

|  |
| --- |
| if |commits| ≥ 2f + 1:  DAG.finalize(round\_id) |

Merkle Root Agreement:

|  |
| --- |
| ∀ commit\_i, commit\_j:  commit\_i.MerkleRoot == commit\_j.MerkleRoot |

* message\_count: Atomic counter for total messages processed, used to measure communication overhead.

Usage:

|  |
| --- |
| overhead = total\_messages / total\_finalized\_rounds |

Tracked Per Message:

* + Message Type (Proposal, Prevote, Commit)
  + Round ID
  + Whether Merkle proofs were verified

*1.3 Data Design & Structures*

BaseRequest

* Description: Shared header for all messages.
* Fields:
  + proposing\_node\_id: u32 – Unique ID of the node sending the message.
  + round\_id: u32 – Identifier for the protocol round.

Transaction

* Description: Encodes a full transaction batch with data shards and integrity proofs.
* Fields:
  + merkle\_root: String – Hex-encoded SHA-256 root of the transaction Merkle tree.
  + shards: Vec<String> – Base64-encoded Reed-Solomon data shards (typically padded to 250 bytes each).
  + proofs: Vec<Vec<String>> – Per-shard Merkle proofs (base64-encoded SHA-256 sibling hashes), one list per shard.
* Notes: Merkle tree is binary, left-right ordered, and used to validate individual shards against merkle\_root.

ProposeRequest

* Description: Sent by the round proposer to initiate consensus.
* Fields:
  + base: BaseRequest – Common request metadata.
  + transactions: Vec<Transaction> – Transactions proposed for this round.
  + parent\_unit\_hashes: Vec<String> – Hex-encoded SHA-256 hashes of parent DAG units.

PrevoteRequest

* Description: Broadcast by nodes after successful validation of a proposal.
* Fields:
  + valid\_proposals: Vec<ProposeRequest> – Proposals that passed Merkle proof and shard validation.
  + sender\_url: String – Network address (URL or IP:port) of the sending node.
* Notes: A node may include multiple proposals if multiple proposers are supported per round.

CommitRequest

* Description: Signals a node has received enough prevotes to finalize a round.
* Fields:
  + dag\_unit: DagUnit – Finalized consensus record for this round.
  + round: u32 – Round number being committed.
  + sender\_id: u32 – ID of the committing node.

DagUnit

* Description: A finalized, committed consensus unit stored in the DAG.
* Fields:
  + unit\_id: String – Hex-encoded unique identifier (typically a SHA-256 hash).
  + proposer\_node: u32 – ID of the node that proposed the unit.
  + round: u32 – Round in which this unit was committed.
  + transactions: Vec<Transaction> – Finalized transactions.
  + parents: Vec<String> – Hashes of parent DagUnits (hex).
  + merkle\_root: String – Root hash from the proposal, stored for audit.
  + finalization\_timestamp: u64 – UNIX timestamp (ms) of finalization.

DAGSyncRequest

* Description: Used when a node is missing consensus units from earlier rounds.
* Fields:
  + requesting\_node\_id: u32 – Node initiating the sync.
  + target\_round: u32 – Round number for which data is requested.
  + missing\_hashes: Vec<String> – Hashes of DAG units not found locally.

ReconstructedUnit

* Description: Temporary structure used during prevote stage for validating proposals.
* Fields:
  + transactions: Vec<Transaction> – Transactions reconstructed from shards.
  + source\_proposer\_id: u32 – ID of the original proposer.
  + reconstructed\_from: Vec<u32> – IDs of nodes whose shards were used.

Node

* Description: Represents a live Aleph node instance participating in the protocol.
* Fields:
  + node\_id: u32 – Unique identifier of the node.
  + total\_nodes: u32 – Total number of nodes in the network.
  + ip\_config: Vec<String> – IP addresses of all nodes.
  + dag: HashMap<u32, Vec<DagUnit>> – Mapping from round to finalized DAG units.
  + proposal\_tracker: Tracker – Tracks proposals received per round.
  + commit\_tracker: Tracker – Tracks commit quorum progress per round.
  + quorum\_votes: HashMap<u32, Vec<u32>> – Tracks vote counts per round.
  + transaction\_size: usize – Bytes per transaction (default: 250).
  + data\_shards: usize – Number of data shards for encoding.
  + total\_rounds: u32 – Total number of consensus rounds to execute.
  + event\_sender: Sender<Event> – Tokio channel for asynchronous message handling.
  + rbc\_processor: RBCProcessor – Logic component for Reliable Broadcast protocol.
  + message\_counter: AtomicUsize – Tracks messages sent/received for overhead metrics.
  + round\_manager: RoundManager – Orchestrates per-round progress and message handling.

TomlConfig

* Description: Parsed from a .toml config file generated by AWS CDK deployment tools.
* Fields:
  + network:
    - node\_id: u32 – This node’s ID.
    - address\_list: Vec<String> – IP or domain names of all nodes.
    - network\_size: u32 – Total number of nodes in the system.
  + consensus:
    - num\_transactions: u32 – Number of transactions per proposal.
    - num\_shards: usize – Shards per transaction for erasure coding.
    - max\_rounds: u32 – Maximum rounds the protocol will run.

*1.4 Protocol Algorithm (Merkle Tree-Based ch-RBC)*

Step 1: Node Initialization

* AWS CDK dynamically provisions EC2 instances for each node.
* Each instance receives a .toml configuration file, passed into the aleph\_rbc Rust binary at runtime.
* Upon launch:
  + The node parses the configuration, initializes logging, and connects to other peers using IP settings from the TOML.
  + Linux SAR commands are launched in the background to monitor and log CPU and memory utilization.
  + A global COMMUNICATION\_OVERHEAD counter is atomically incremented for every inbound or outbound message.

Step 2: Proposal Phase

* The designated proposer for the round generates a batch of transactions and divides each transaction into k data shards using Reed-Solomon erasure coding.  
  For each transaction:
  + A binary Merkle tree is constructed over the base64-encoded SHA-256 hash of each shard.
  + The root of this tree (merkle\_root) provides a succinct commitment to the data.
* The proposer then broadcasts a ProposeRequest containing:
  + The Merkle root.
  + The base64-encoded data shards.
  + Corresponding Merkle inclusion proofs for each shard.
* Literature Reference: Merkle trees [Merkle, 1987] provide a cryptographically secure method of summarizing large datasets. This design follows the encoding and validation logic outlined in the HoneyBadgerBFT paper (Miller et al., 2016), where each shard is verified using a Merkle proof to ensure correctness before participating in consensus.
* For the first round, a latency start timestamp is recorded.

Step 3: Proposal Validation

* Each node receiving a ProposeRequest performs the following checks:
  + Verifies that no duplicate proposal has been received from the same sender for the same round.
  + Confirms that shard sizes match the expected byte length (transaction\_size / data\_shards).
  + Validates each shard’s Merkle proof against the claimed merkle\_root, using the SHA-256 hash of the decoded shard.
  + Ensures that all parent DAG units listed in the proposal are locally available.
  + If any parent units are missing, the node sends a DAGSyncRequest and waits until all parents are synced.

Step 4: Prevote Phase

* If one or more valid proposals pass verification, the node constructs a PrevoteRequest containing accepted ProposeRequests.
* The node multicasts the PrevoteRequest to all other nodes.
* Upon receiving prevotes:
  + Each node tracks sender uniqueness using sender URLs to avoid duplicate counting.
  + Transactions are reconstructed using received shards and Reed-Solomon interpolation.
  + For each reconstructed transaction, the node re-hashes the shards and recomputes the Merkle root to ensure it matches the original merkle\_root.
* Literature Reference: The interpolation and Merkle root verification steps follow the Reliable Broadcast mechanism described in the Aleph protocol paper ([Gągol et al., 2019](https://arxiv.org/pdf/1901.08755.pdf)), which emphasizes the use of Merkle proofs for efficient shard-level verification across distributed replicas.

Step 5: Commit Phase

* When a node receives ≥ 2f + 1 valid PrevoteRequests (where f is the maximum number of faulty nodes tolerated), it creates and broadcasts a CommitRequest.
* Upon receiving a CommitRequest, a node:
  + Discards duplicates from the same sender for the same round.
  + Tracks the number of unique commit messages for that round.
* When a commit quorum is achieved (≥ 2f + 1):
  + The node finalizes the round by inserting a new DagUnit containing the committed transactions and parent hashes into the local DAG.
  + Emits a RoundFinalized event.
  + Persists the finalized DAG unit to disk.
  + Logs a timestamp used for TPS (throughput) computation.
* If the finalized round is equal to total\_rounds (as defined in the TOML config):
  + Log the end timestamp for latency evaluation.  
    Log the final communication overhead metric.
  + Upload all logs and DAG data to a centralized S3 bucket.
  + Shut down the node gracefully.

*1.5 Data Storage & Persistence*

* Finalized DAGs: Persisted to disk as json file after every round via utils::config\_utils::write\_finalized\_dag\_to\_file.
* Metrics:
  + Communication Overhead : Stored to general log file.
  + Throughput : Stored to general log file.
  + Memory Utilization : Sar commands stored to cpu\_usage and memory\_usage log files respectively.
  + Latency: Stored to general log file.
* After the final round
  + Upload logs and DAG state to AWS S3.

**Stage 2: Design & Implementation of RSA-Based Aleph**

*2.1 Architecture Overview*

* Language: Rust.
* Consensus Validation: RSA Accumulators + Reed-Solomon Erasure Coding.
* Communication Protocol: JSON over HTTP (port 30333).
* Deployment Environment: Distributed EC2 network (VPC).
* Logging: tracing + chrono for structured logs and timestamping.
* Storage: File-based JSON output for finalized DAG and a text file for logs that sync to a S3 bucket.

*2.2 System Components*

Core Protocol Engine

* src::aleph\_rbc: Executes the Propose, Prevote, and Commit phases; processes messages, maintains quorum state, and finalizes rounds using RSA accumulator validation.

Transaction Generator & RSA Accumulator Encoding

* utils::create\_transaction\_data: Generates transaction batches, splits them into erasure-coded shards, computes RSA accumulator roots, and generates inclusion proofs.
* utils::rsa\_accumulator\_utils: Provides utilities for accumulator creation, exponentiation-based proof generation, and verification.
* Literature Support: These proof structures follow the RSA accumulator principles introduced by (Benaloh and de Mare, 1993) and expanded in (Camenisch & Lysyanskaya, 2002), with applications to blockchains in (Reddy, 2021) and (Hussein & Al-Gailani, 2022).

*Creating Transaction Data Algorithm -* Splits transactions into shards, hashes them to

primes, builds an RSA accumulator, and generates inclusion proofs.

|  |
| --- |
| Function CreateTransactionData\_RSA(txs: Vec<Vec<u8>>, k: usize, n: usize, modulus: BigUint, generator: BigUint)  Input:  - txs: List of raw transactions  - k: Number of data shards  - n: Total number of shards  - modulus: RSA modulus (N)  - generator: Generator (g) for the accumulator group  Output:  - List of (shards: Vec<Vec<u8>>, accumulator\_root: BigUint, proofs: Vec<BigUint>)  Steps:  results = []  For each tx in txs:  Pad tx to fixed length (e.g., 250 bytes) → padded\_tx  Apply Reed-Solomon encoding → shards: Vec<Vec<u8>>  For each shard in shards:  Compute SHA-256 hash → hash\_bytes  Map hash\_bytes to a prime via hash\_to\_prime() → prime  Append to prime\_list  Compute accumulator\_root = generator ^ (product of all primes) mod modulus  For each j in 0..k:  product\_except\_j = ∏ (all primes except prime\_j)  proof\_j = generator ^ product\_except\_j mod modulus  Append (shards, accumulator\_root, [proof\_0, ..., proof\_k-1]) to results  Return results |

*Construct RSA Accumulator Root -* Computes the RSA accumulator root by exponentiating the generator with the product of all mapped primes.

|  |
| --- |
| Function ComputeAccumulatorRoot(primes: Vec<BigUint>, generator: BigUint, modulus: BigUint)  Input:  - primes: Vector of prime values mapped from hashed shards  - generator: RSA base (g)  - modulus: RSA modulus (N)  Output:  - accumulator\_root: BigUint  Steps:  exponent = product of all primes in primes[]  accumulator\_root = generator ^ exponent mod modulus  Return accumulator\_root |

*Generate RSA Inclusion -* Creates a proof for one shard by exponentiating the generator with the product of all primes except the one being proven.

|  |
| --- |
| Function GenerateRSAInclusionProof(primes: Vec<BigUint>, index: usize, generator: BigUint, modulus: BigUint)  Input:  - primes: List of mapped primes (one per shard)  - index: Index of the shard to generate proof for  - generator: g  - modulus: RSA modulus  Output:  - proof: BigUint  Steps:  proof\_exponent = product of all primes except primes[index]  proof = generator ^ proof\_exponent mod modulus  Return proof |

*Verify RSA Inclusion -* Validates an inclusion proof by checking if proof^prime ≡ accumulator\_root (mod N).

|  |
| --- |
| Function VerifyRSAInclusionProof(proof: BigUint, prime: BigUint, accumulator\_root: BigUint, modulus: BigUint)  Input:  - proof: RSA inclusion witness (g^∏\_{j≠i}p\_j mod N)  - prime: The prime value mapped from the shard hash  - accumulator\_root: Accumulator value (g^∏p\_i mod N)  - modulus: RSA modulus  Output:  - Boolean: True if proof is valid, else False  Steps:  reconstructed = proof ^ prime mod modulus  Return reconstructed == accumulator\_root |

*Hash to Prime*

|  |
| --- |
| Function HashToPrime(data: &[u8])  Input:  - data: Byte slice (e.g., a shard or its hash)  Output:  - prime: A BigUint that is provably prime  Steps:  1. hash = SHA-256(data)  2. candidate = BigUint::from\_bytes\_be(hash)  3. While candidate is not prime:  candidate += 1  4. Return candidate |

Inter-node Communication

* controllers::api\_routes: Handles HTTP message passing between nodes on port 30333.

*Proposal Message Format -* Contains transaction shards, their RSA inclusion proofs, and the accumulator root, allowing nodes to verify data authenticity and membership before prevoting.

|  |
| --- |
| {  "round": 0,  "proposer\_id": 2,  "transactions": [  {  "shards": ["<base64-encoded shard>", "..."],  "proofs": ["<base64-encoded RSA proof>", "..."],  "accumulator\_root": "<base64-encoded accumulator value>"  }  ]  } |

*Prevote Message Format -* Signals a node’s support for a proposed accumulator root after reconstructing transactions and verifying RSA inclusion proofs for each shard.

|  |
| --- |
| {  "round": 0,  "sender\_id": 1,  "transaction\_index": 0,  "shard\_index": 0,  "shard\_data": "<base64-encoded shard>",  "proof": "<base64-encoded RSA proof>",  "accumulator\_root": "<base64-encoded accumulator root>"  } |

*Commit Message Format -* Confirms a quorum of prevotes and finalizes a round by committing the accumulator root into the DAG for that transaction batch.

|  |
| --- |
| {  "round": 0,  "sender\_id": 1,  "transaction\_index": 0,  "shard\_index": 0,  "shard\_data": "<base64-encoded shard>",  "proof": "<base64-encoded RSA proof>",  "accumulator\_root": "<base64-encoded accumulator root>"  } |

DAG Utilities

Utils::dag\_utils: Offers DAG size checks, round synchronization enforcement, and parent availability verification.

Key Components

* CheckDagSize(dag)

|  |
| --- |
| Ensures |dag| ≤ MAX\_SIZE, where MAX\_SIZE is a tunable memory threshold. |

* VerifyParentsExist(round)

|  |
| --- |
| ∀ parent ∈ round.parents:  parent ∈ DAG.finalized |

* EnforceRoundSync(current, msg\_round)

|  |
| --- |
| if msg\_round > current + 1:  discard(msg) |

RSA Accumulator Proof Integration

* Finalized DAG entries include: AccumulatorRoot, ShardIndex, and

RsaInclusionProof.

RSA Accumulator Verification

|  |
| --- |
| Each shard s\_j is:   1. Hashed using SHA-256 2. Mapped to a prime p\_j via hash\_to\_prime 3. Verified using:   Verify: proof^p\_j mod N == accumulator\_root |

This RSA check must succeed before the proposal is accepted into the DAG.

Literature Reference: Based on RSA inclusion proof techniques from (Benaloh & de Mare, 1993), (Camenisch & Lysyanskaya, 2002), and applied in blockchain contexts per (Reddy, 2021).

Message Processing Engine

* processors::priority\_queue: Implements a BinaryHeap-based queue to enforce message priority:

Ordering variant:

|  |
| --- |
| RoundFinalized > Commit > Prevote > Proposal |

Implemented via enum discriminants and heap-based min-heap inversion.

Dropping Logic:

|  |
| --- |
| if msg.round < current\_finalized:  discard(msg) |

* processors::rbc\_processor: Consumes prioritized messages, handles round logic, quorum tracking, and initiates rebroadcasts. Manages pruning of outdated messages and ensures round progression.

Algorithmic Flow

|  |
| --- |
| upon\_receive(msg):  match msg.type:  Proposal => handle\_propose(msg)  Prevote => handle\_prevote(msg)  Commit => handle\_commit(msg) |

RSA Inclusion Verification

Each shard s\_j is verified by computing:

|  |
| --- |
| let hash = SHA256(s\_j)  let prime = hash\_to\_prime(hash)  assert(proof^prime mod N == accumulator\_root) |

This must hold before accepting the proposal into the DAG.

Quorum and Round Finalization

Progresses once:

|  |
| --- |
| |prevotes| ≥ 2f + 1 ∧ |commits| ≥ 2f + 1 |

On Commit Quorum:

|  |
| --- |
| DAG.insert(round\_id, AccumulatorRoot, shards) |

Message Tracking & Quorum Management

* proposal\_tracker: Tracks unique proposals; quorum is N - f.

Quorum Rule:

|  |
| --- |
| quorum\_met ⇐ |unique\_proposers| ≥ N - f |

RSA Integrity Enforcement:

|  |
| --- |
| ∀ shard s\_j with proof witness\_j:  proof^hash\_to\_prime(SHA256(s\_j)) ≡ accumulator\_root mod N |

* quorum\_votes: Tracks prevote messages; quorum is 2f + 1.

Threshold Logic:

|  |
| --- |
| if |prevotes| ≥ 2f + 1:  send\_commit(accumulator\_root) |

Proof Soundness Guarantee:

|  |
| --- |
| ∀ s\_j ∈ shards:  VerifyRSAInclusionProof(witness\_j, prime\_j, accumulator\_root) == true |

* commit\_tracker: Tracks commit messages; quorum is 2f + 1.

Commit Quorum:

|  |
| --- |
| if |commits| ≥ 2f + 1:  DAG.finalize(round\_id) |

Accumulator Root Agreement:

|  |
| --- |
| ∀ commit\_i, commit\_j:  commit\_i.accumulator\_root == commit\_j.accumulator\_root |

* message\_count: Atomic counter for total messages processed, used to measure communication overhead.

Usage:

|  |
| --- |
| overhead = total\_messages / total\_finalized\_rounds |

Tracked Per Message:

* + Message Type (Proposal, Prevote, Commit)
  + Round ID
  + Whether RSA proof were verified

*2.3 Data Design & Structures*

BaseRequest

* Description: Shared header for all messages.
* Fields:
  + proposing\_node\_id: u32 – Unique ID of the node sending the message.
  + round\_id: u32 – Identifier for the protocol round.

Transaction

* Description: Represents a full transaction batch encoded with erasure coding and validated via RSA accumulator inclusion proofs.
* Fields:
  + accumulator\_root: String – Base64-encoded RSA accumulator value computed over shard hashes.
  + shards: Vec<String> – Base64-encoded Reed-Solomon data shards (typically padded to 250 bytes each).
  + proofs: Vec<String> – One base64-encoded RSA inclusion proof per data shard.
* Notes: Each shard is hashed with SHA-256 and mapped to a prime via hash\_to\_prime. The inclusion proof is valid if proof^prime = accumulator\_root (mod N).

ProposeRequest

* Description: Sent by the round proposer to initiate consensus.
* Fields:
  + base: BaseRequest – Common request metadata.
  + transactions: Vec<Transaction> – Transactions proposed for this round.
  + parent\_unit\_hashes: Vec<String> – Hex-encoded SHA-256 hashes of parent DAG units.

PrevoteRequest

* Description: Broadcast by nodes after successful validation of a proposal.
* Fields:
  + valid\_proposals: Vec<ProposeRequest> – Proposals that passed Merkle proof and shard validation.
  + sender\_url: String – Network address (URL or IP:port) of the sending node.
* Notes: A node may include multiple proposals if multiple proposers are supported per round.

CommitRequest

* Description: Signals a node has received enough prevotes to finalize a round.
* Fields:
  + dag\_unit: DagUnit – Finalized consensus record for this round.
  + round: u32 – Round number being committed.
  + sender\_id: u32 – ID of the committing node.

DagUnit

* Description: A finalized, committed consensus unit stored in the DAG.
* Fields:
  + unit\_id: String – Hex-encoded unique identifier (typically a SHA-256 hash).
  + proposer\_node: u32 – ID of the node that proposed the unit.
  + round: u32 – Round in which this unit was committed.
  + transactions: Vec<Transaction> – Finalized transactions.
  + parents: Vec<String> – Hashes of parent DagUnits (hex).
  + accumulator\_root: String – RSA accumulator stored for audit.
  + finalization\_timestamp: u64 – UNIX timestamp (ms) of finalization.

DAGSyncRequest

* Description: Used when a node is missing consensus units from earlier rounds.
* Fields:
  + requesting\_node\_id: u32 – Node initiating the sync.
  + target\_round: u32 – Round number for which data is requested.
  + missing\_hashes: Vec<String> – Hashes of DAG units not found locally.

ReconstructedUnit

* Description: Temporary structure used during prevote stage for validating proposals.
* Fields:
  + transactions: Vec<Transaction> – Transactions reconstructed from shards.
  + source\_proposer\_id: u32 – ID of the original proposer.
  + reconstructed\_from: Vec<u32> – IDs of nodes whose shards were used.

Node

* Description: Represents a live Aleph node instance participating in the protocol.
* Fields:
  + node\_id: u32 – Unique identifier of the node.
  + total\_nodes: u32 – Total number of nodes in the network.
  + ip\_config: Vec<String> – IP addresses of all nodes.
  + dag: HashMap<u32, Vec<DagUnit>> – Mapping from round to finalized DAG units.
  + proposal\_tracker: Tracker – Tracks proposals received per round.
  + commit\_tracker: Tracker – Tracks commit quorum progress per round.
  + quorum\_votes: HashMap<u32, Vec<u32>> – Tracks vote counts per round.
  + transaction\_size: usize – Bytes per transaction (default: 250).
  + data\_shards: usize – Number of data shards for encoding.
  + total\_rounds: u32 – Total number of consensus rounds to execute.
  + event\_sender: Sender<Event> – Tokio channel for asynchronous message handling.
  + rbc\_processor: RBCProcessor – Logic component for Reliable Broadcast protocol.
  + message\_counter: AtomicUsize – Tracks messages sent/received for overhead metrics.
  + round\_manager: RoundManager – Orchestrates per-round progress and message handling.

TomlConfig

* Description: Parsed from a .toml config file generated by AWS CDK deployment tools.
* Fields:
  + network:
    - node\_id: u32 – This node’s ID.
    - address\_list: Vec<String> – IP or domain names of all nodes.
    - network\_size: u32 – Total number of nodes in the system.
  + consensus:
    - num\_transactions: u32 – Number of transactions per proposal.
    - num\_shards: usize – Shards per transaction for erasure coding.
    - max\_rounds: u32 – Maximum rounds the protocol will run.

*2.4 RSA Protocol Algorithm*

Step 1: Node Initialization

* AWS CDK provisions EC2 instances for each Aleph node.
* Each instance receives a .toml configuration file passed into the aleph\_rbc Rust binary at runtime.
* Upon startup:
  + The node parses the configuration, initializes logging, and establishes connections to peers using the provided network topology.
  + Background sar monitoring tools are launched to log CPU and memory usage.
  + An atomic COMMUNICATION\_OVERHEAD counter is incremented for every inbound and outbound message to track protocol-level communication cost.

Step 2: Proposal Phase

* The designated proposer for the round generates a batch of transactions and applies Reed-Solomon encoding to produce k data shards per transaction.
* For each transaction:
  + Each shard is hashed using SHA-256.
  + Each hash is mapped to a prime using a deterministic hash\_to\_prime function.
  + The proposer computes an RSA accumulator root A = g^∏p\_i mod N, where p\_i are the mapped primes.
  + For each shard, an inclusion proof (witness) w\_i = g^∏\_{j≠i}p\_j mod N is generated.
* The proposer broadcasts a ProposeRequest containing:
  + The RSA accumulator root.
  + The base64-encoded shards.
  + The base64-encoded inclusion proofs.
* Literature Reference: RSA accumulators (Benaloh & de Mare, 1993) allow compact, constant-sized commitments and support efficient membership proofs. Their application to verifiable distributed systems is further expanded in (Camenisch & Lysyanskaya, 2002) and (Reddy, 2021). This construction enables succinct data verification without needing Merkle trees.
* For the first round, the proposer logs the latency start timestamp.

Step 3: Proposal Validation

Each node receiving a ProposeRequest performs the following validations:

* Confirms that the proposal is not a duplicate for the current round and proposer.
* Verifies each shard:
  + Decode the shard from base64.
  + Compute its SHA-256 hash and map it to a prime.
  + Use the inclusion proof to check:  
     proof^prime ≡ accumulator\_root (mod N)
* Validates shard size:  
   len(shard) == transaction\_size / data\_shards
* Ensures that all parent DAG unit hashes listed in the proposal are locally available.
* If any parent units are missing, the node sends a DAGSyncRequest and waits for synchronization before proceeding.

Step 4: Prevote Phase

* If one or more proposals are successfully validated:
  + The node constructs a PrevoteRequest containing accepted ProposeRequests.
  + The PrevoteRequest is multicast to all other nodes.
* Upon receiving prevotes:
  + Each node tracks sender uniqueness using the sender\_url field to prevent duplicate counting.
  + It reconstructs the full transaction from received shards using Reed-Solomon decoding.
  + For each reconstructed shard, it recomputes the SHA-256 hash, maps it to a prime, and re-verifies the accumulator proof using the corresponding witness and accumulator root.
* Literature Reference: This prevote stage builds upon techniques in (Hussein & Al-Gailani, 2022) where RSA-based membership proofs are verified after partial data reconstruction, ensuring integrity in fault-tolerant distributed settings.

Step 5: Commit Phase

* Once a node collects at least 2f + 1 valid PrevoteRequests (where f is the fault tolerance threshold), it constructs and broadcasts a CommitRequest.
* Upon receiving a CommitRequest, the node:
  + Discards duplicate commit messages from the same sender for the same round.
  + Tracks the number of unique commit messages for quorum detection.
* When a commit quorum (≥ 2f + 1) is reached:
  + The node inserts a new DagUnit into its local DAG, including the committed transaction batch, parent references, and accumulator root.
  + Emits a RoundFinalized event to notify other system components of round completion.
  + Logs a timestamp for throughput (TPS) calculation.
* If the committed round matches total\_rounds defined in the TOML config:
  + Log the latency end timestamp.
  + Log the final message count for communication overhead analysis.
  + Upload finalized DAG and logs to AWS S3.
  + Gracefully shut down the node.

*2.5 Data Storage & Persistence*

* Finalized DAGs: Persisted to disk as json file after every round via utils::config\_utils::write\_finalized\_dag\_to\_file.
* Metrics:
  + Communication Overhead : Stored to general log file.
  + Throughput : Stored to general log file.
  + Memory Utilization : Sar commands stored to cpu\_usage and memory\_usage log files respectively.
  + Latency: Stored to general log file.
* After the final round
  + Upload logs and DAG state to AWS S3.

## **Stage 3: Deployment & Validation of Each Variant**

*3.1 AWS Infrastructure via CDK (Python)*

Stack Definition: The entire infrastructure is provisioned using the TestAleph AWS CDK stack, written in Python, which ensures reproducibility and systematic deployment of experiments.

Infrastructure Components:

* VPC with public subnets for inter-node communication.
* Security Groups to isolate node traffic and expose only required ports (30333 for RPC, 22 for SSH).
* IAM Roles provide EC2 instances access to S3 and CloudWatch.
* EC2 Instances for Aleph nodes, sized dynamically (e.g., t3.medium) based on experiment scale.
* IPManager Node (t2.micro): lightweight node responsible for coordination and readiness signaling.
* CloudWatch Log Group for optional real-time monitoring of logs and metrics.

Deployment Behavior:

* EC2 nodes are launched with user-data scripts.
* All nodes use the same aleph\_rbc binary but execute under different TOML configuration files.
* All logs and DAG files are eventually uploaded to S3 for validation.

*3.2 Node Startup Workflow*

The node bootstrap process is fully automated and instrumented for consistency:

1. Artifact Preparation:
   * aleph\_rbc is compiled for x86\_64-unknown-linux-musl via cargo build --release.
   * IpServer.py and the compiled binary are uploaded to S3 (aleph-research bucket).
   * An md5sum is printed and logged for post-deployment verification.
2. Execution Flow Upon AWS CDK Stack Launch:
   * Infrastructure is instantiated: VPC, subnets, roles, security groups, EC2 nodes, and IPManager.
   * Each node performs the following steps:
     + Installs essential tools (gcc, python3, jq, aws-cli, awslogs, etc.).
     + Starts sar to collect CPU and memory usage metrics every 5 seconds.
     + Downloads aleph\_rbc and IpServer.py from S3.
     + Registers itself with the IPManager instance.
     + Waits until all other nodes have registered.
     + Receives a generated aleph-node-config.toml with:
       - [network]: IP/port, peer list, and IPManager URL.
       - [consensus]: number of transactions, transaction size, number of shards, total rounds.
       - [logging]: log level, file output.
       - [node]: unique node ID.
     + Starts the aleph\_rbc binary and exposes a /health endpoint over port 30333.

*3.3 Workload Composition and Variant Validation Methodology*

This section provides the core testing methodology for validating each variant (Merkle tree-based vs RSA accumulator-based) of Aleph ch-RBC.

Transaction Workload Design

Each transaction is designed to stress-test shard encoding, proof verification, and reconstruction processes. The composition is as follows:

* Payload Size: 250 bytes per transaction (as per Aleph's original RBC design).
* Encoding: Reed-Solomon erasure coding with configurable parameters k (data shards) and n (total shards), where k/n = 0.5–0.75.
* Batch Size: Configurable (e.g., 512 to 8192 transactions per round).
* Rounds: Each experiment runs a fixed number of rounds (e.g., 20–50), as defined in TOML.
* Workload Pattern:
  + Each transaction is uniformly random data with padding.
  + Transactions are deterministic across nodes to isolate proof logic differences between variants.

*Validation Objectives Per Variant*

|  |  |  |  |
| --- | --- | --- | --- |
| Variant | Commitment Method | Verification Goal | Literature Basis |
| Merkle Tree | SHA-256 tree root | Rebuild and verify Merkle root from shard and proof | (Miller et al., 2016), (Gągol et al., 2019) |
| RSA Accumulator | Exponentiated root (g^∏p\_i mod N) | Verify proof^p\_i ≡ root mod N per shard | (Reddy, 2021) |

* Correctness Criteria: The protocol is validated by ensuring that all nodes:
  + Reach proposal threshold N-F in proposal phase.
  + Reach quorum thresholds (2f + 1) during prevote and commit phases.
  + Finalize DAG units for each round.
  + Log RoundFinalized events and persist finalized data to disk.
* Log-Based Validation: Nodes append structured logs to disk that indicate:
  + Transaction processing events.
  + Proposal acceptance and quorum achievement.
  + DAG finalization and log uploads.
* Consistency Checks:
  + Reconstructed units match original proposals.
  + Finalized DAG units are written to consistent files.
  + Message counts remain within expected bounds.

## **Stage 4: Evaluation & Analysis**

### *4.1 Implementation Validation Plan*

Following the successful deployment and execution of each variant, this phase focuses on post-execution validation and analysis to ensure correctness and completeness of the consensus protocol. While validation criteria were enforced during runtime (see Stage 3.3), this section outlines how correctness is confirmed via post-mortem log parsing, data inspection, and cross-node consistency checks.

Key Validation Strategies:

1. Log-Based Protocol Tracing

* Each node emits structured logs indicating:
  + Proposals received and accepted
  + Proofs successfully verified (Merkle or RSA)
  + RoundFinalized events
  + DAG insertion with accumulator or Merkle root
* Logs are parsed to verify:
  + Proposal threshold (N - f) was met.
  + Quorum thresholds (2f + 1) were achieved in both prevote and commit phases.
  + Each finalized round includes valid transaction roots and matching transactions.

2. Cross-Node DAG State Consistency

* DAG files from all nodes are downloaded from S3.
* For each round:
  + Compare unit\_id, proposer\_id, and accumulator\_root/merkle\_root across all nodes.
  + Ensure that all nodes finalized the same units for a given round.

3. Shard & Proof Verification Replay

* A standalone validation tool reprocesses a sample of finalized transactions:
  + For Merkle, recompute root from shard + proof and compare.
  + For RSA, re-run verify\_proof(proof, prime, accumulator\_root) for each shard.
* Confirms that no corrupted or unverifiable shard made it into the final DAG.

4. Failure Case Checks

* Look for:
  + Any rounds with incomplete quorum.
  + Any proposal or shard rejections.
  + Missing or delayed RoundFinalized events.

5. Communication Volume Accounting

* Total messages sent (from message\_counter per node) are aggregated.
* Confirms expected bounds for overhead relative to quorum rules and batch sizes.

Threats to Validity and Reliability:

* *Internal Validity*
  + Ensuring that variations in communication complexity result solely from RSA accumulators, not external network conditions.
  + Using a consistent transaction size and batch processing method across both implementations.
* *External Validity*
  + Evaluating the system under different node configurations (8 to 104 nodes) to simulate scalability.
  + Ensuring results can generalize to real-world blockchain systems by comparing with prior studies on HoneyBadgerBFT, BEAT, ABFT, and Aleph.
* *Instrument Validity*
  + The correctness of RSA accumulator integration will be verified through independent cryptographic validation tests.
  + Comparative analysis will be performed using well-established benchmarking methods used in prior blockchain consensus research.

### *4.2 Experiment Design for Comparison*

This section outlines how each experiment is designed, deployed, and analyzed to compare the Merkle-based and RSA accumulator-based variants of the Aleph ch-RBC protocol. Each experiment is configured to isolate the effect of the validation mechanism while holding all other factors constant including the number of nodes, transaction size, shard encoding, and runtime duration.

The experiments are designed to:

* Test performance and communication behavior under different network sizes and batch loads.
* Maintain reproducibility via AWS CDK-deployed infrastructure.
* Generate structured logs and artifacts for detailed post-execution analysis.

Each experiment follows a consistent flow:

* Setup: Define node count, batch size, and protocol variant (Merkle or RSA).
* Runtime Execution: Deploy EC2 nodes, run consensus for a fixed number of rounds.
* Data Collection: Capture throughput, latency, resource usage, and message volume.
* Artifact Upload: Persist logs to S3.
* Analysis: Post-process logs to extract performance metrics and generate visual summaries.
* Variables:
  + EC2 Instance Count: Specifies the number of EC2 nodes participating in the network. Each instance runs an Aleph node.
  + Number of Transactions per Round: Sets how many transactions each node proposes during each round, influencing system load and throughput.
  + Transaction Size: Defines the size of each transaction in bytes, impacting data volume and network usage.  
    Shard Size: Indicates how many data shards are created per transaction for erasure coding, contributing to data redundancy and validation.
  + Round Count: Determines the total number of consensus rounds the protocol executes, affecting the duration and depth of the experiment.
* Metrics Captured:
  + Throughput (TPS): Calculated per node by dividing the total number of committed transactions (TxPerRound × Rounds) by the elapsed time between the first and last RoundFinalized log entries. Logged timestamps mark the start and end of this period.
  + Latency: Measured as the total time taken to process all transactions, starting from the first transaction proposal and ending after the final round is complete via logged timestamps.
  + Communication Overhead: Tracked using message\_count, an atomic counter that records the total number of network messages (propose, prevote, commit) sent and received by each node. This metric helps evaluate protocol scalability.
  + Resource Usage: CPU and memory statistics are collected on each EC2 node using the sar system monitoring tool and exported to logs.
* Experimental Control:
  + Fresh EC2 instances launched for each variant.
  + Configuration file generated per node dynamically.
  + On the final round uploads logs to S3 bucket.
* Repeatability:
  + Each experiment can be rerun using AWS CDK script.
  + Logs and production code are archived under timestamped S3 folders.

### *4.3 Metrics Collection & Analysis*

* Repository Format: Logs will be loaded to s3://aleph-research/ in this format.

|  |
| --- |
| ├── experiments/  │ └── N{nodes}\_B{batch}\_R{rounds}  │ ├── node-0/  │ │ ├── run.log  │ │ ├── cpu\_usage.log  │ │ └── mem\_usage.log  │ ├── node-1/  │ └── … |

* Metric Origins:
  + Latency:
    - Logged via timestamp from info!("LATENCY START") in the node when the first transaction is sent.
    - Ends in handle\_commit when the final round completes: info!("LATENCY END").
  + Throughput (TPS):
    - Measured in handle\_commit after successful round finalization and DAG insertion: info!("Finalized round {} with {}/{} commits. TPS METRIC").
  + Communication Overhead:
    - Captured by incrementing message\_count every time a message is sent or received. Reported as info!("COMMUNICATION OVERHEAD: {:?}", message\_count).
  + Resource Usage (CPU & Memory):
    - Collected using sar via CDK startup scripts.
* Log Output:
  + Logs written to /aleph-node/logs.
  + Files include node\_status, cpu\_usage, mem\_usage.
* Upload Strategy:
  + On completion, logs are uploaded to S3 under: logs/RSA\_N{n}\_T{tx}\_R{r}.
* Parsing & Aggregation:
  + Logs are downloaded from S3 bucket.
  + Post-experiment scripts parse logs for Throughput, Latency, Communication Overhead, and Resource Utilization metrics.
  + Outputs findings to a human readable metrics report.
  + CSV summaries generated for plotting in Python or Excel.
* Visualization:
  + Using CSV summaries Throughput, Latency, Communication Overhead, and Resource Utilization metrics plotted over rounds for both Merkle and RSA variants.
  + Final charts made in Excel or Python graphing library.
* Storage Format:
  + Finalized DAGs persisted as structured as JSON files.
  + Finalized logs persisted as text files.
* Example run.log

|  |
| --- |
| [INFO] 2025-05-01T12:00:00Z LATENCY START: round 0  [INFO] 2025-05-01T12:00:12Z Finalized round 0 with 7/7 commits. TPS METRIC  [INFO] 2025-05-01T12:00:24Z Finalized round 15 with 7/7 commits. TPS METRIC  [INFO] 2025-05-01T12:01:22Z Finalized round 30 with 7/7 commits. TPS METRIC  [INFO] 2025-05-01T12:01:25Z LATENCY END: round 30  [INFO] 2025-05-01T12:01:30Z COMMUNICATION OVERHEAD: 2450 |

* Example mem\_usage.log

|  |
| --- |
| Time kbmemfree kbmemused %memused kbbuffers kbcached  12:00:05 PM 204800 812800 79.85 12000 280000  12:00:06 PM 200500 817100 80.30 12000 285000 |

* Example cpu\_usage.log

|  |
| --- |
| Time CPU user nice system iowait idle  12:00:05 PM all 2.37 0.00 96.11 0.00 1.52  12:00:10 PM all 3.04 0.00 95.08 0.00 1.88  12:00:15 PM all 4.10 0.00 94.02 0.00 1.88 |

**Proposed Sample**

To evaluate the scalability and performance of the Aleph ch-RBC protocol under varying consensus validation mechanisms, we simulate an asynchronous permissionless network using Amazon EC2 medium instances with 2 Virtual CPUs, mimicking the experimental instances of HBFT, BEAT, and ABFT. These works provide validated blueprints for EC2-based deployments under heavy throughput and high-latency. Following the BEAT deployment model, each transaction will be 250 bytes.

*Sample Ranges*

* Initial Network Size: Experiments begin with 8 nodes (small-scale) and scale up to 96 nodes to simulate increasingly complex quorum and communication behavior.
* Transaction Size: All transactions are fixed at 250 bytes, consistent with the experimental designs of HBFT, BEAT, and ABFT.
* Batch Sizes: Range from 128 to 10,000 transactions per round, allowing us to observe behavior under both low-throughput and high-throughput settings. This bounded range helps isolate communication and validation costs without overwhelming resource limits.
* Shard Configuration: For each experiment, the number of data shards is set to N - ⌊N/3⌋ and the number of parity shards is the remainder. This ensures resilience while maintaining consistent erasure coding pressure across the network.
* Total Rounds: Computed as max(1, B / N) per configuration to keep round duration practical and relative to system size.

This approach emulates performance bottlenecks and bandwidth stressors observed in prior permissionless BFT systems while validating Aleph's communication efficiency with RSA accumulators.

*Aleph ch-RBC Experiment Matrix (10 Scenarios)*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Exp # | Nodes (N) | Batch Size (B) | Data Shards | Parity Shards | Total Rounds | Purpose |
| 1 | 8 | 128 | 6 | 2 | 16 | Small-scale correctness baseline, fast quorum |
| 2 | 12 | 256 | 8 | 4 | 21 | Mid-scale low batch test for RSA proof validation |
| 3 | 16 | 1024 | 11 | 5 | 64 | Moderate load test for shard recovery + DAG tracking |
| 4 | 20 | 256 | 14 | 6 | 13 | High-N, low-TPS setup to isolate comm overhead |
| 5 | 24 | 2048 | 16 | 8 | 86 | Balanced config for end-to-end throughput |
| 6 | 32 | 512 | 22 | 10 | 16 | Commit + prevote delays with realistic shard count |
| 7 | 40 | 128 | 27 | 13 | 4 | High quorum tracking overhead under minimal load |
| 8 | 48 | 8192 | 32 | 16 | 171 | Max throughput stress test for RSA accumulator |
| 9 | 64 | 512 | 43 | 21 | 8 | Low batch test at scale for DAG sync validation |
| 10 | 96 | 10000 | 64 | 32 | 104 | Peak scale test for end-to-end latency + TPS tracking |

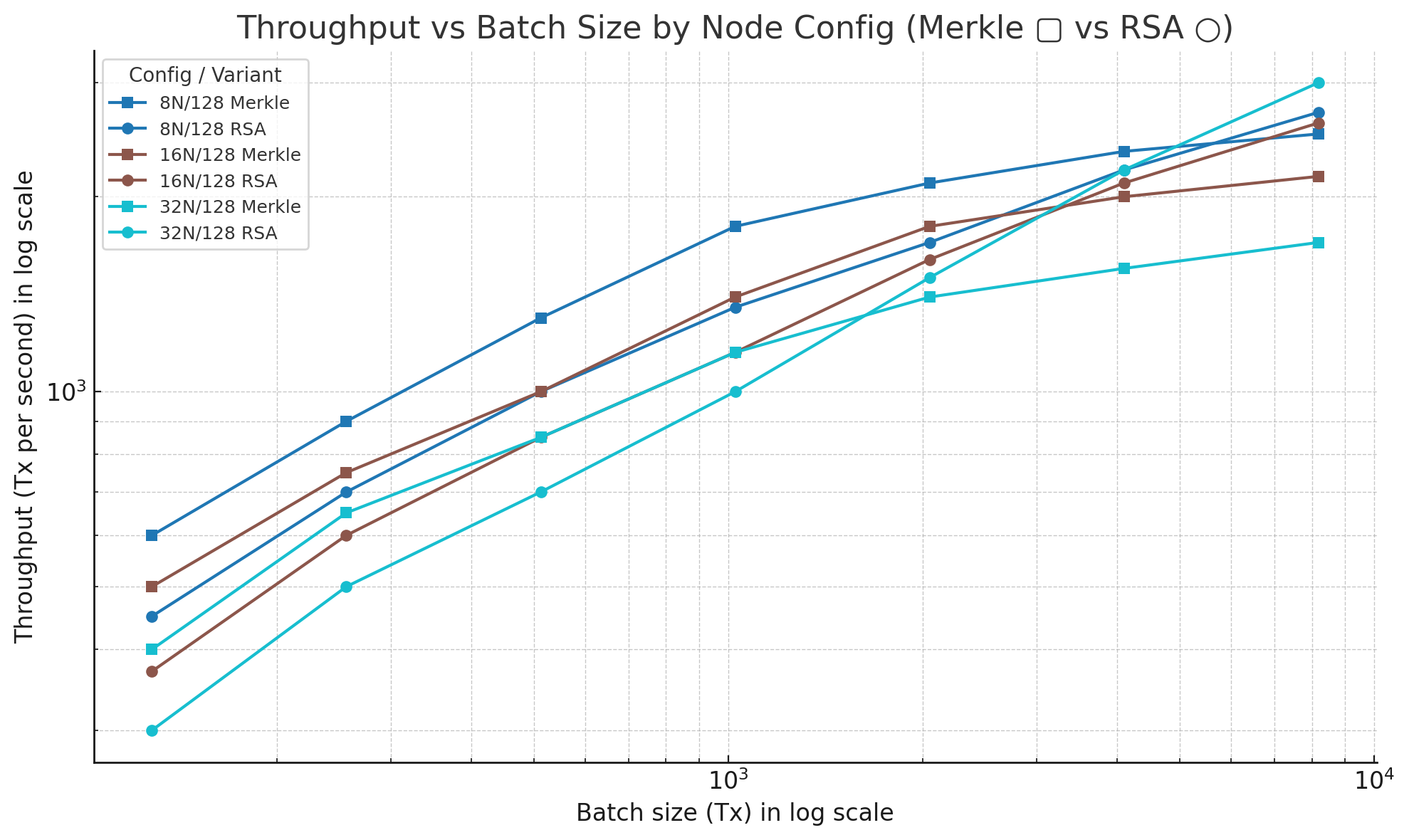
### *Visual Presentation of Results*

In the final paper, figures inspired by HBFT, BEAT, and ABFT will illustrate the protocol’s behavior under different load and network conditions.

Displaying TPS vs Batch Size

This figure plots transaction throughput (TPS) on the Y-axis against batch size on the X-axis using a log-log scale. Each color represents a specific deployment configuration (e.g., 8, 16, 32, or 96 nodes), while the marker shape distinguishes the consensus variant squares for Merkle and circles for RSA.

The goal is to compare how throughput scales across different batch sizes and network sizes, highlighting where RSA accumulators begin to outperform Merkle trees in terms of communication efficiency and scalability.



This graph will be displayed three times to highlight small, medium and large deployments for clarity and reduction of clutter.

TPS vs Batch Size Graph 1: Small Deployments

* Exp 1: 8 nodes / 128 batch — baseline correctness test.
* Exp 2: 12 nodes / 256 batch — RSA proof validation focus.
* Exp 3: 16 nodes / 1024 batch — moderate load for DAG recovery.

TPS vs Batch Size Graph 2: Medium Deployments

* Exp 4: 20 nodes / 256 batch — isolates communication overhead.
* Exp 5: 24 nodes / 2048 batch — balanced throughput evaluation.
* Exp 6: 32 nodes / 512 batch — commit and shard realism testing.

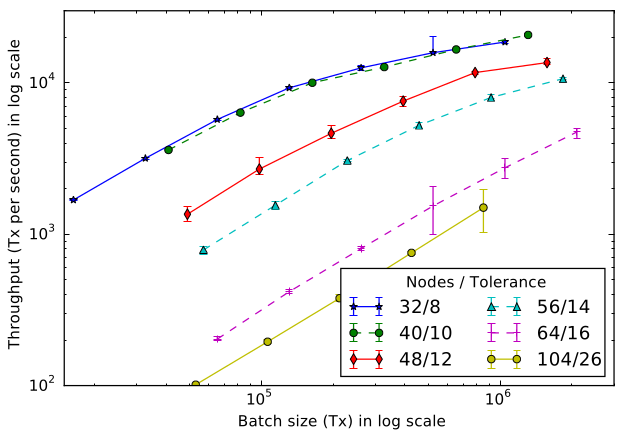
TPS vs Batch Size Graph 3: Large Deployments

* Exp 7: 40 nodes / 128 batch — quorum tracking stress at low TPS.  
  Exp 8: 48 nodes / 8192 batch — max throughput stress for RSA.
* Exp 9: 64 nodes / 512 batch — large-scale DAG sync validation.
* Exp 10: 96 nodes / 10000 batch — full-scale latency and TPS test.

Each variant (RSA, Merkle) is plotted per deployment tier to show where performance diverges, with RSA expected to scale more effectively at higher batch sizes and larger networks. This structured comparison ensures each protocol is fairly evaluated under matched workloads and progressively increasing system complexity.

The visualization was inspired by Figure 6 in the HoneyBadgerBFT paper, which analyzes TPS scaling across batch sizes and network scales.

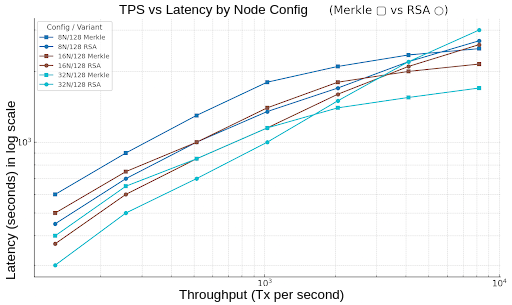
#### *HBFT Figure 6 : TPS vs Batch Size*



Displaying TPS vs Latency

This figure plots transaction latency (in seconds, log scale) on the Y-axis against throughput (TPS) on the X-axis. Each color corresponds to a distinct deployment configuration (e.g., 8, 16, or 32 nodes with fixed batch size), while the marker shape differentiates between consensus variants — squares for Merkle trees and circles for RSA accumulators.

The objective is to assess how latency is impacted by increasing TPS across different network scales, and to identify the tipping point where RSA accumulators begin to offer reduced latency under high-throughput loads compared to Merkle-based validation.



To reduce visual clutter and enable focused interpretation, the visualization is divided into three deployment tiers:

TPS vs Latency Graph 1: Small Deployments

* Exp 1: 8 nodes / 128 batch — baseline correctness and latency.
* Exp 2: 12 nodes / 256 batch — small RSA validation latency.
* Exp 3: 16 nodes / 1024 batch — moderate workload and DAG propagation.

TPS vs Latency Graph 2: Medium Deployments

* Exp 4: 20 nodes / 256 batch — isolates latency due to communication overhead.
* Exp 5: 24 nodes / 2048 batch — latency under high shard counts.
* Exp 6: 32 nodes / 512 batch — realistic conditions for commit + shard recovery.

TPS vs Latency Graph 3: Large Deployments

* Exp 7: 40 nodes / 128 batch — quorum stress at low throughput.
* Exp 8: 48 nodes / 8192 batch — RSA latency under peak throughput.
* Exp 9: 64 nodes / 512 batch — DAG sync latency at scale.
* Exp 10: 96 nodes / 10,000 batch — end-to-end latency bottleneck evaluation.

Each configuration shows how latency scales relative to achieved TPS, and where the RSA variant starts to outperform Merkle in terms of round finalization time. This structure provides a fair side-by-side evaluation of each consensus variant as the network size and throughput demands increase.

The layout and purpose of this figure are directly inspired by Figure 7 from the HoneyBadgerBFT paper, which presented a similar analysis of latency vs TPS trade offs across consensus configurations.

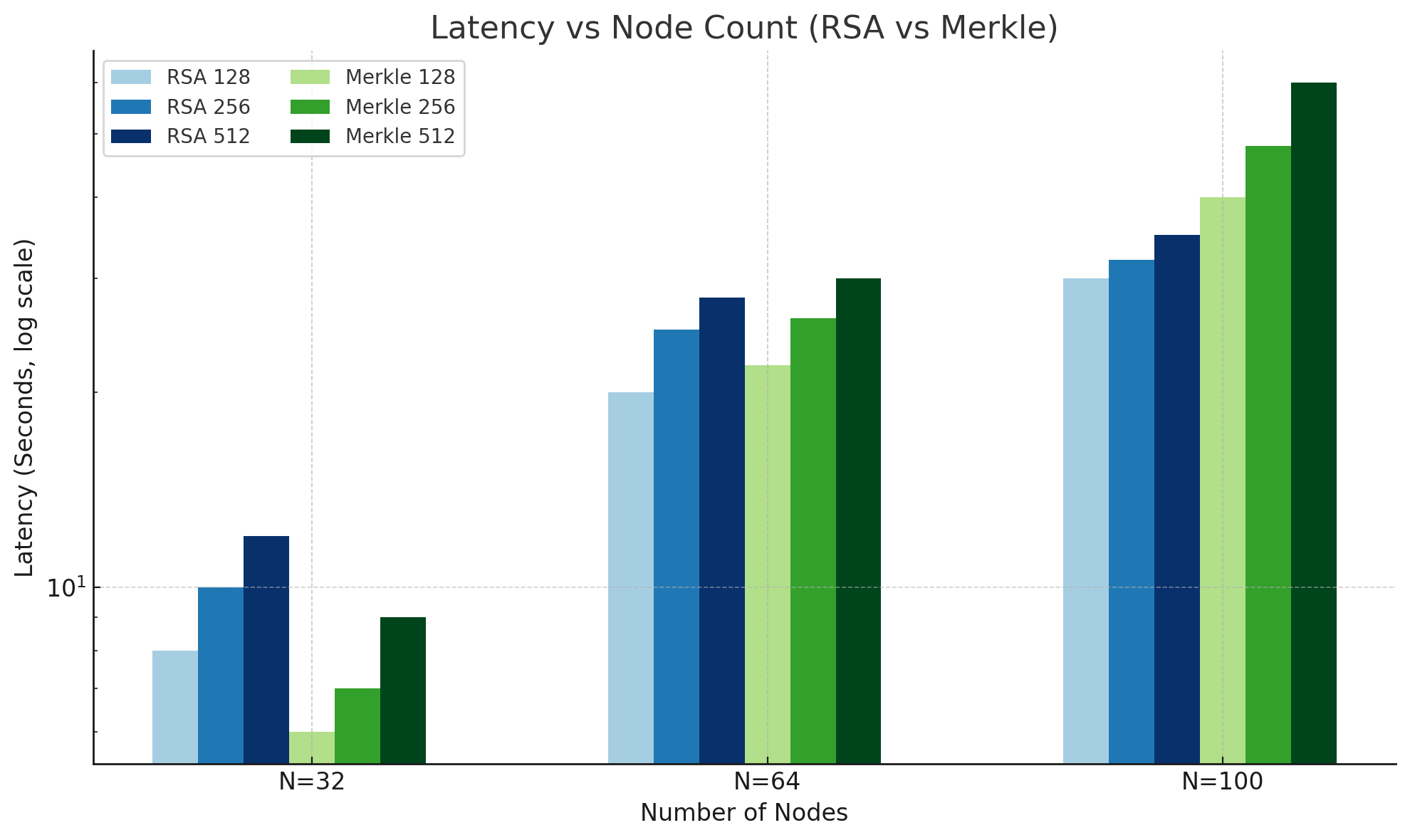
#### *HBFT Figure 7 : TPS vs Latency*

#### 

Displaying Latency vs Node Count

This bar graph plots latency (seconds) on a logarithmic scale against the number of nodes (N = 32, 64, 100), showcasing how both Merkle and RSA implementations perform under increasing network sizes. Each group of bars represents three configurations based on transaction batch sizes: 128, 256, and 512 transactions.

* Merkle variants are represented in three shades of green, showing better performance at smaller scales due to faster proof verification.
* RSA variants are in shades of blue, showing initially higher latency but scaling more efficiently as the number of nodes grows—eventually outperforming Merkle in large deployments due to reduced communication overhead.



This visualization is modeled after Figure 1 from the ABFT paper, which analyzes how consensus latency grows with the number of nodes. Our adapted version provides a side-by-side comparison of RSA vs Merkle-based ch-RBC, clearly illustrating the scalability crossover point.

#### *ABFT Figure 1 : Nodes vs Latency*

#### 

### **Resource Requirements Summary**

* Cloud Infrastructure: AWS EC2 t3.medium, S3, IAM roles, VPCs, and security groups.
* Deployment: AWS CDK (Python), automated setup scripts for binary upload and logging.  
  Software Stack: Rust (protocol), Python (analysis and test setup), SAR, and bash.
* Visualization: Jupyter Notebooks, Seaborn, Matplotlib, Excel.

**Summary**

This chapter detailed the research methodology for evaluating the impact of RSA accumulators on Aleph’s ch-RBC protocol. A comparative experimental approach is used, implementing both Merkle tree-based and RSA-based versions of ch-RBC and testing them in AWS-based decentralized network simulations.

Key elements of the methodology include:

* Implementing and validating the RSA accumulator-based ch-RBC in Rust.
* Conducting large-scale simulations with varying network sizes and transaction loads.
* Collecting and analyzing key performance metrics such as communication overhead, throughput, latency, and resource usage.
* Using structured logging, AWS monitoring, and statistical analysis to evaluate results.

By following this systematic and reproducible approach, the research will provide empirical benchmarks demonstrating how RSA accumulators improve the scalability and efficiency of asynchronous permissionless consensus protocols.

**References**

Amazon Web Services, Inc. (n.d.). *AWS Cloud Development Kit (AWS CDK)*. Retrieved August

26, 2024, from<https://aws.amazon.com/cdk/>

Baird, L., & Luykx, A. (2020). The Hashgraph protocol: Efficient asynchronous BFT for

high-throughput distributed ledgers. In *International Conference on Omni-layer*

*Intelligent Systems* (pp. 1-7).<https://doi.org/10.1109/COINS49042.2020.9191430>

Belling, A., Soleimanian, A., & Ursu, B. (2024). *Vortex: A list polynomial commitment and its*

*application to arguments of knowledge*. Cryptology ePrint Archive, Paper 2024/185.

Retrieved from<https://eprint.iacr.org/2024/185>

Cardinal Cryptography. (n.d.). *AlephBFT* [GitHub repository]. GitHub. Retrieved October 14,

2024, from<https://github.com/Cardinal-Cryptography/AlephBFT/tree/main>

Djari, A., Anceaume, E., & Tucci-Piergiovanni, S. (2022). An extensive agent-based simulation

study of sycomore++, a DAG-based permissionless ledger. In *Proceedings of the 37th*

*ACM/SIGAPP Symposium on Applied Computing* (pp. 334–336). Association for

Computing Machinery.<https://doi.org/10.1145/3477314.3507245>

Duan, S., Reiter, M., & Zhang, H. (2018). Beat: Asynchronous BFT made practical. In *ACM*

*SIGSAC Conference on Computer and Communications Security* (pp. 2028–2041).

<https://doi.org/10.1145/3243734.3243812>

Gągol, A., Leśniak, D., Straszak, D., & Świętek, M. (2019). Aleph: Efficient atomic broadcast in

asynchronous networks with Byzantine nodes. In *Proceedings of the 1st ACM*

*Conference on Advances in Financial Technologies (AFT '19)* (pp. 214–228).

Association for Computing Machinery.<https://doi.org/10.1145/3318041.3355467>

Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., & Zhang, Z. (2022). Dumbo-NG: Fast asynchronous

BFT consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM*

*SIGSAC Conference on Computer and Communications Security (CCS '22)* (pp.

1187–1201). Association for Computing Machinery.<https://doi.org/10.1145/3548606.3559379>

Gencer, A. E., Basu, S., Eyal, I., van Renesse, R., & Sirer, E. G. (2018). Decentralization in

Bitcoin and Ethereum networks. In *Proceedings of the 22nd International Conference on*

*Financial Cryptography and Data Security* (pp. 439-457). Springer.

<https://doi.org/10.48550/arXiv.1801.03998>

Guo, B., Lu, Z., Tang, Q., Xu, J., & Zhang, Z. (2020). Dumbo: Faster asynchronous BFT

protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and*

*Communications Security (CCS '20)* (pp. 803–818). Association for Computing

Machinery.<https://doi.org/10.1145/3372297.3417262>

Guo, B., Lu, Y., Lu, Z., Tang, Q., Xu, J., & Zhang, Z. (2022). Speeding Dumbo: Pushing

asynchronous BFT closer to practice.<https://doi.org/10.14722/ndss.2022.24385>

Han, J., Liu, X., & Li, K. (2023). Swarm: A high-performance asynchronous BFT protocol

adapted to high network delay. In *Proceedings of the 2023 IEEE 29th International*

*Conference on Parallel and Distributed Systems (ICPADS)* (pp. 1801-1808). Ocean

Flower Island, China.<https://doi.org/10.1109/ICPADS60453.2023.00249>

Hajian Berenjestanaki, M., Barzegar, H. R., El Ioini, N., & Pahl, C. (2024). Blockchain-based

e-voting systems: A technology review. *Electronics, 13*(1), 17.

<https://doi.org/10.3390/electronics13010017>

Hussein, K. M., & Al-Gailani, M. F. (2022). An efficient bandwidth based on the cryptographic

technique of the RSA accumulator in blockchain networks. In *2022 Fifth College of*

*Science International Conference of Recent Trends in Information Technology (CSCTIT)*

(pp. 164-168). Baghdad, Iraq.<https://doi.org/10.1109/CSCTIT56299.2022.10145614>

Jupyter. (n.d.). Jupyter notebooks. Retrieved August 26, 2024, from https://jupyter.org/

Kharangate, A. (2023). Asynchronous Merkle trees. arXiv.

<https://ar5iv.labs.arxiv.org/html/2311.17441>

Knudsen, H., Li, J., Notland, J., Haro, P., & Ræder, T. (2021). High-performance asynchronous

Byzantine fault tolerance consensus protocol. In *IEEE International Conference on*

*Blockchain* (pp. 476-483).<https://doi.org/10.1109/Blockchain53845.2021.00073>

Lauinger, J., Ernstberger, J., Regnath, E., Hamad, M., & Steinhorst, S. (2021). A-PoA:

Anonymous proof of authorization for decentralized identity management. In *2021 IEEE*

*International Conference on Blockchain and Cryptocurrency* (ICBC) (pp. 1-9). Sydney,

Australia. doi:10.1109/ICBC51069.2021.9461082

Manzoor, R., Sahay, B. S., & Singh, S. K. (2022). Blockchain technology in supply chain

management: An organizational theoretic overview and research agenda. *Annals of*

*Operations Research*.<https://doi.org/10.1007/s10479-022-05069-5>

Miller, A., Xia, Y., Croman, K., Shi, E., & Song, D. (2016). The honey badger of BFT protocols.

In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications*

*Security* (pp. 31–42).<https://doi.org/10.1145/2976749.2978399>

PostgreSQL. (n.d.). *PostgreSQL: The world's most advanced open source relational database*.

Retrieved August 26, 2024, from<https://www.postgresql.org/>

Principato, M., Babel, M., Guggenberger, T., Kropp, J., & Mertel, S. (2023). *Towards solving the*

*blockchain trilemma: An exploration of zero-knowledge proofs*. In *ICIS 2023*

*Proceedings* (pp. 1-15). Association for Information Systems.

<https://aisel.aisnet.org/icis2023/blockchain/blockchain/5>

Reddy, B. S. (2021). securePrune: Secure block pruning in UTXO based blockchains using

accumulators. In *2021 International Conference on COMmunication Systems &*

*NETworkS (COMSNETS)* (pp. 174-178). Bangalore, India.

<https://doi.org/10.1109/COMSNETS51098.2021.9352892>

The Rust Project Developers. (n.d.). *The Rust programming language*. Retrieved August 26,

2024, from<https://www.rust-lang.org/>

Singh, C., Thakkar, R., & Warraich, J. (2022). Blockchain in supply chain management.

*European Journal of Engineering and Technology Research, 7*(5), 60-69.

<https://doi.org/10.24018/ejeng.2022.7.5.2888>

The Tracing Project. (n.d.). *tracing: Documentation*. Retrieved August 26, 2024, from

<https://docs.rs/tracing/latest/tracing/>

Tokio Contributors. (n.d.). *Tokio*. Retrieved August 26, 2024, from<https://tokio.rs/>

Woznica, A., & Kedziora, M. (2022). Performance and scalability evaluation of a permissioned

blockchain based on Hyperledger Fabric, Sawtooth and Iroha. *Comput. Sci. Inf. Syst.,*

*19*, 659-678.

Zhou, Q., Huang, H., Zheng, Z., & Bian, J. (2020). Solutions to scalability of blockchain: A

survey. *IEEE Access*. Advance online publication.

<https://doi.org/10.1109/ACCESS.2020.2967218>