FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Multiparty Set Reconciliation

Pedro Vidal Marcelino

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Carlos Baquero-Moreno

July 31, 2026

Multiparty Set Reconciliation

Pedro Vidal Marcelino

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Francisco Maia
Referee: Prof. Ana Alonso
Referee: Prof. Carlos Baquero-Moreno

July 31, 2026

Abstract

The set reconciliation problem, that consists in bringing two or more replicas of a set into agreement is present in all databases, distributed ledgers and any large scale synchronization systems. While the two party case is well studied, the multiparty one exposes a large design space of protocols whose communication cost varies drastically with topology, replica count and the level of similarity. A very slow space to search on by hand.

This thesis investigates the use of an autonomous agent loop as a research tool to navigate that specific design space. This means using an agent to iteratively propose, implement and evaluate reconciliation protocols against a single measurable objective until convergence is achieved. In this case, the objective was the total bytes transmitted, including state and metadata, taken as a geometric mean over an evaluation matrix with eighteen cells and three seeds.

The search yielded `MultiReplica`, a topology dispatched protocol that combines all neighbor Bloom filter exchange on star and tree topologies with pairwise distance doubling on chord and that reduces communication cost by roughly 30 % relative to the strongest hand designed baseline while scaling to replica counts at which the baselines exhaust memory. The contribution is twofold. On one side the discovered protocol itself and a reproducible demonstration that an autonomous agent loop can serve as a credible, auditable method for distributed systems protocol research.

Keywords: Set reconciliation • Multiparty set reconciliation • Multi-replica reconciliation • Set difference computation • State synchronization • Distributed systems • Replicated data • Eventual consistency • Replica convergence • Anti-entropy protocols • Gossip protocols • Network topology • Peer-to-peer systems • Distributed databases • Distributed ledgers • Bloom filters • Invertible Bloom lookup tables • Rateless invertible Bloom lookup tables • Rateless Bloom filters • Set sketches • Probabilistic data structures • Coded sketches • Hash-based set representation • Communication efficiency • Bandwidth optimization • Communication complexity • Byte-cost minimization • Convergence rounds • Autonomous agent-driven protocol design • LLM-based program search • Automated algorithm discovery • AI-assisted systems research • Agentic optimization loop • Protocol synthesis • Empirical protocol evaluation • Reproducible experimental methodology

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals(https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

Although this dissertation is a technical study of multiparty set reconciliation, its contributions connect to this framework in two ways. First, the reconciliation protocols discovered here reduce the communication resources required to keep distributed systems consistent, which is intrinsically an efficiency objective. Second, the autonomous agent loop introduced as a research instrument is itself a contribution to how distributed-systems protocols are designed, searched, and validated. Third, and final this thesis has some impact on climate action as transmitting fewer bytes lowers the energy demand of the network and compute infrastructure that carries the data, and hence its carbon footprint. This effect is a downstream consequence of the efficiency gains and is not quantified in this work. Note that no energy or emissions saving is claimed. The goals to which a direct, measurable contribution can be attributed are listed below.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	The discovered protocols reconcile identical state with substantially fewer transmitted bytes, improving the efficiency of the communication layer underlying replicated and distributed systems.	Total bytes transmitted to reach convergence (the fitness function). A relative reduction in comparison with the the best baseline protocol (of the order of 30 % at $N = 16$).
	9.5	Enhancing scientific research and innovation: an autonomous agent loop is demonstrated as a reproducible research instrument for discovering and validating reconciliation protocols.	Number of distinct configurations searched and validated (64 logged attempts); a fully reproducible search trajectory and released artifacts.
12	12.2	Efficient use of resources: minimizing the bytes exchanged per reconciliation directly reduces the communication resources consumed to keep distributed replicas consistent.	Geometric-mean communication cost across the 18-cell evaluation matrix. A relative reduction in comparison with the the best baseline protocol (of the order of 30 % at $N = 16$).

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BCH	Bose–Chaudhuri–Hocquenghem
BF	Bloom Filter
CPISync	Characteristic Polynomial Interpolation-based Synchronization
CPU	Central Processing Unit
CRDT	Conflict-Free Replicated Data Type
CSV	Comma-Separated Values
CV	Coefficient of Variation
ECC	Error Correcting Code
FPR	False Positive Rate
FST	Full State Transfer
IBF	Invertible Bloom Filter
IBLT	Invertible Bloom Lookup Table
JSON	JavaScript Object Notation
LLM	Large Language Model
OOM	Out Of Memory
OS	Operating System
PBS	Parity Bitmap Sketch
PRNG	Pseudo-Random Number Generator
QUIC	Quick UDP Internet Connections
RAM	Random Access Memory
RBF	Rateless Bloom Filter
RIBLT	Rateless Invertible Bloom Lookup Table
SDG	Sustainable Development Goal
TCP	Transmission Control Protocol
TOML	Tom's Obvious Minimal Language

Declaration of honour

I declare that this dissertation is my original work and has not previously been submitted or assessed in any other course or curricular unit, whether at this or any other institution.

References to other authors (statements, ideas, thoughts), as well as the use of published works fully complies with the rules of attribution and are appropriately cited in the text and bibliography, in accordance with reference standards. I am aware that plagiarism and self-plagiarism constitute academic misconduct.

I further declare that I have not used Artificial Intelligence tools to produce any part(s) of this dissertation or, when such tools were used, their use and results are clearly indicated and were carefully reviewed for the detection of errors, inaccuracies, unfounded attributions, or other forms of content and argument bias, as well as for the determination of authorship.

I further declare that I have included studies carried out with multiple co-authors, having specified, with transparency and rigour, the contribution of each co-author, provided information on the possible existence of any other dissertation in which the same article has been included, reproduced the work in full with the permission of the journal in which it was published, and obtained the explicit written consent of all co-authors for the inclusion of the article.

Pedro Vidal Marcelino

Contents

Chapter 1

Introduction

In modern distributed systems, high availability and fault tolerance are a must have in order to enable low-latency access, making data replication essential. However with replication comes along the challenge of maintaining consistency among all copies. As each copy starts updating their replicated states, they begin to diverge creating the need for synchronization. This ends up affecting every class of systems, including distributed databases, peer-to-peer networks and replicated data types such as Conflict-Free Replicated Data Types (CRDTs)

This complicated task of synchronization can be defined as a set reconciliation problem. In other words, given two or more sets - distinct states of shared data - the goal is to exchange the least possible missing information to allow all parts to reach a common state. With this abstraction, it is possible to encapsulate a wide range of distributed systems, allowing this algorithms to be used effectively across all kind of platforms.

1.1 Context

Traditional naive synchronization methods, such as transmitting full state across nodes or relying on centralized coordination, despite simple are often bandwidth heavy or failure prone, making them effectively infeasible under latency constraints or susceptible to network failures.

More modern set reconciliation techniques however, offer a and bandwidth efficient alternative. As recent studies show, by combining probabilistic structures, such as Bloom Filters (BFs) with others, Invertible Bloom Lookup Tables (IBLTs)—replicas can efficiently communicate the missing elements exchanging only the necessary information with the slight overhead of encoding and decoding the data. Some recent research has advanced these methods even further: *Practical Rateless Set Reconciliation* [15] introduced a rateless coding scheme effectively removing the need for prior size negotiation for decoding IBLTs between replicas. Similarly, *ConflictSync* [6] having applied these to state-CRDTs, demonstrates the benefits of efficient set reconciliation bandwidth-efficient state synchronization can be applied even in complex peer-to-peer replicated data structures.

Despite the substantial progress made in recent years in the field of efficient set reconciliation, its practical applications are still limited. Most of the existing literature, such as *Practical Rateless Set Reconciliation* [15] and *ConflictSync* [6], while insightful, mostly focuses on a two-replica scenario, failing to mimic real-world behavior, as most systems involve not only a larger number of nodes with different states but also diverse network conditions and topologies. Extending these algorithms to multi-party environments depends on all these variables and represents a crucial step forward in understanding how efficiency can scale in all types of realistic systems. This paper therefore, is a logical step forward in this field of investigation and is of great value from an academic perspective.

One of the objectives of this research paper is its concrete practical application in the real world. Despite their efficiency the integration of these types of algorithms, still remains low. In current peer-to-peer large scale networks like Bitcoin and Ethereum, transaction propagation still relies primarily on gossip or flooding-based protocols, where nodes repeatedly announce transactions to all peers, producing large amounts of redundant communication and bandwidth waste. Despite recent optimizations, such as Ethereum's *NewPooledTransactionHashes* mechanism or Bitcoin's proposed Erlay protocol, there is still some space for improvement specially in these types of environment where speed and consistency matters the most. Applying advanced set reconciliation methods to these processes could increase efficiency drastically. Take mempool synchronization in cryptocurrency networks for instance, different network nodes must continuously exchange lists of unconfirmed transactions before adding them to the ledger. This whole process could be made far more efficient if rather than broadcasting transaction IDs, they shared compact BFs or rateless IBLTs.

1.2 Problem

The main question this thesis addresses is how to apply and generalize the principles behind efficient pairwise reconciliation - to multi party setups.

While the current reconciliation approaches - presented in *Practical Rateless Set Reconciliation* [15] and *ConflictSync* [6] - reach high efficiency in pairwise scenarios, naively applying them to every pair of nodes in a distributed system would result in quadratic communication growth with avoidable redundant data exchange.

1.3 Objectives

Building upon the principles introduced in *Practical Rateless Set Reconciliation* [15] and *ConflictSync* [6], the main goal of this thesis is to employ an agent driven search over reconciliation protocols configurations to discover efficient multi-party set reconciliation techniques, and to study how can topology and state similarity among multiple nodes impact bandwidth usage.

- Study existing reconciliation algorithms and identify their limitations in different multi-party scenarios.

- Build a controlled, reproducible simulator that measures the bandwidth cost of reconciliation across network topologies, replica counts, and state-divergence levels.
- Use an agent-driven search over protocol configurations within this simulator to discover topology-aware strategies that minimize redundant data transfer.
- Evaluate the discovered strategies against established baselines and gather insightful metrics.
- Return conclusions on the practicality and scalability of the solutions, in real-world scenarios (i.e.: CRDTs, blockchain).

1.4 Dissertation structure

In addition to this introduction, the dissertation is organized as follows.

Chapter 2 reviews the foundations the work builds on: the set reconciliation problem and the contexts that give rise to it, together with the probabilistic structures these techniques rely on, including BFs and IBLTs, as well as the flooding based approaches they aim to improve upon.

Chapter 3 surveys the state of the art in efficient set reconciliation, focusing on the recent rateless and CRDT advances that motivate this work, and identifies the gap it addresses, such as, their confinement to the two-party case and the quadratic cost of applying them naively across many nodes.

Chapter 4 presents the approach taken for achieving results in this thesis, describing the multi-party reconciliation model, the network topologies and protocols under study, and how the problem is cast as a search over protocol configurations.

Chapter 5 details the evaluation framework (harness), the simulator, the bandwidth-cost metrics it records, the fitness objective, and the agent-driven search used to explore the configuration space.

Chapter 6 reports the experimental results, which include the final protocol the search discovered and its bandwidth efficiency against established baselines across the tested topologies, scales, and divergence levels. It closes with a discussion of the threats to validity this study faces.

Chapter 7 concludes the dissertation, summarizing its contributions and outlining directions for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter describes the methodology used to gather the academic publications relevant to its objects presented in the Introduction Chapter. It aims at describing the entire process in order to allow for a reproducible set of documents for identifying the current state of the art in this research field. As for the chosen methodology, it was used a mixture of both exploratory and systematic.

2.2 Collection and screening methodology

2.2.1 Screening and selection process

Firstly, papers were gathered from IEEE XPLORE, ACM Digital Library, Arxiv and google scholar using defined queries registered below, with the respective results being assigned to a separate Zotero collection based on the query responsible for retrieving the publication after the Screening process. The second phase consisted in the analysis of the Titles and abstracts of each potential publication, to eliminate clearly irrelevant works. From the selected publications, those where relevance was questionable were target of the Eligibility phase, where a full-text assessment was performed to assess its suitability to the topic of efficient multiparty set synchronization. The final set of publications was then included in this thesis for citation and analysis in the State of the Art chapter.

2.2.2 Search strategy

Beside some exploratory search starting from the initial papers, made to retrieve work highly relatable to the theme of thesis [5] and [15], the literature search was made across IEEE Xplore, Arxiv, ACM Digital Library and google scholar. The following four search queries were used in the four search engines, each with a different objective. The queries are as follows

1. **Set reconciliation foundations** The aim of this query is to retrieve papers related to the basis of the set reconciliation problem, and to set up the main assumptions the rest of thesis will be built upon.

   ("set reconciliation" OR "set synchronization" OR "set difference reconciliation") AND ("symmetric difference" OR "set difference metric") AND ("communication complexity" OR "lower bound" OR "information theoretic" OR "optimal communication")

2. **Coding theoretic approaches**

   This query is used to retrieve the coding-theoretical approaches that transform the set reconciliation problem into an error correction problem. We combine the following terms, to make sure both Error Correcting Code (ECC) based and polynomial based techniques are found.

   ("set reconciliation" OR "set difference") AND ("error correcting code" OR "syndrome" OR "BCH" OR "secure sketch" OR "fuzzy extractor" OR "characteristic polynomial" OR "polynomial interpolation") AND ("decoding complexity" OR complexity OR "communication complexity" OR "trade-off" OR "optimal communication")

3. **Probabilistic data structures**

   This query is used with the intent of finding reconciliation protocols , based on probabilistic data structures, such as BF variants and IBLTs.

   ("set reconciliation" OR "set difference") AND ("Bloom filter" OR IBF OR "Invertible Bloom Filter" OR IBLT OR "Invertible Bloom Lookup Table" OR "Difference Digest") AND ("peeling" OR "decode" OR "decoding" OR "subtraction" OR "pure cell")

4. **Rateless and multi-replica reconciliation**

   This query's goal is to find the literature whose work is the closest to this thesis goal, by combining both the rateless set reconciliation keywords with topology aware terminology.

   ("rateless" OR "adaptive" OR "rate-compatible" OR "multi-party" OR "multi-replica") AND ("set reconciliation" OR "set difference" OR "state reconciliation") AND ("topology" OR "overlay network" OR "gossip" OR "mesh" OR "redundancy" OR "linear sketch" OR "network coding")

2.2.3 Inclusion and exclusion criteria

After the initial filtering phase where duplicates and non full-text publications were left out, another round of exclusion criteria was applied to remove the papers irrelevant to the research topic.

For each publication to be included in the final collection had to reference a study addressing either synchronization or set reconciliation algorithms. Publications were also considered if they were related to bandwidth efficiency or communication optimization over high latency networks or similar topics. We also included papers focused on either probabilistic data structures, ECC

portocols or rateless algorithms in the context of set reconciliation mechanisms. Finally, we also included papers related to the set reconciliation problem over multi replica environments.

Some exclusion criteria were also applied when filtering the retrieved papers. Firstly, publications were considered to lack relevance if they focused only on general data synchronization without a clear formulation of the set reconciliation problem, if they only referred to studies focusing on membership testing without difference recovery, as is the case with some probabilistic data structures related studies, and lastly if the paper did not mention distributed systems. Moreover, we also excluded papers with methodological limitations such as protocols requiring Full State Transfer (FST) to complete the reconciliation process, or those lacking sufficient detail to assess the complexity of the presented protocol.

In order to narrow results down to those that specifically mention helpful topics for this research, publications such as surveys with no experimental evaluation or improvements on algorithms were discarded. Publications about data synchronization over consensus protocols or those solely related to blockchain security, were also excluded from the final collection, due to its lack of contribution to the bandwidth efficiency topic.

Table 2.1: Search queries used in the literature review

Query ID	**Theme**	**Purpose**
Q1	Foundations and lower bounds	Identify formal definitions of set reconciliation and information-theoretic communication limits.
Q2	Coding-theoretic approaches	Retrieve ECC-based reconciliation methods and analyze communication–computation trade-offs.
Q3	Probabilistic data structures	Capture BF, Invertible Bloom Filter (IBF)/IBLT-based reconciliation and peeling-based decoding techniques.
Q4	Rateless and multi-replica reconciliation	Retrieve adaptive, rateless, and topology-aware reconciliation techniques relevant to multi-replica settings.

2.3 Results

The final result of the selection process resulted in a set of scientific publications relevant to the topic of set reconciliation and bandwidth friendly communication.

Since many queries returned duplicate papers, each paper was sorted into groups according to its main theme, rather than the query, or queries, that retrieved it. Four groups were made, papers about the foundational limits and optimal cost for set reconciliation in general, secondly probabilistic approaches to the problem, the results were also separated based on whether they are about coding theoretical reconciliation or protocols based on error correcting codes and lastly, more importantly, if they focused on rateless solutions over multi party environments. Some papers

were put into two categories based on their importance to both areas. This separation of concerns allowed for a much more accurate assessment of the relevance of each paper for the State of the Art chapter, and can be found in the following subsections.

We are confident this methodology allowed us to retrieve a significant set of publications to establish the state of the art when making this thesis. All the publications can be found in the bibliography.

2.3.1 Set reconciliation foundations and lower bound

This group of papers is dedicated to work formally describing the set reconciliation problem and analyzing its limits across difference scenarios, by deriving lower bounds on both communication complexity and computational cost. this clarifies which performance metrics are viable to be used to evaluate a protocol and those that are inherent to the problem itself.

The main objective of including these papers in the literature review is to clearly define a well founded baseline for accurately choosing the best algorithms and protocols in the rest of the thesis, exposing trade-offs, associated with system assumptions in real-world distributed systems.

By setting up these theoretical foundations, we ensure later comparisons between error correcting, probabilistic and rateless approaches are made with the same optimal criteria rather than empirical performance.

2.3.2 Probabilistic data structures for set reconciliation

Great part of the state of the art in set reconciliation protocols is based on probabilistic data structures, more specifically BFs, these and other variants such as IBFs and IBLTs have been explored and studied together in many approaches, from systems with prior context and knowledge about the set difference to systems with no previous information where an estimation is required or a rateless approach must be used.

Including this group of papers in this literature review is crucial for finding the trade-offs protocols faced when trying to reduce the computational cost induced by theoretically correct protocols. These works evaluate the impact of false positives and decoding probability on the efficiency of each protocol, while laying out the ground for future developments such as rateless approaches and topology aware reconciliation approaches based on BFs and IBLT data structures.

2.3.3 Coding theoretic approaches to set reconciliation

Besides probabilistic data structures, using sketches based on ECCs to retrieve the symmetric difference between two different replicas is a popular approach in many state of the art studies. This groups papers focused on this approach.

Coding theoretic protocols have the particular benefit of low communication cost, as they represent sets in sketches over large universes, removing the need for transmitting element identifiers and other metadata. However this communication efficiency comes at the cost of the computational cost inherent to their decoding process, which limits their usability in high divergence scenarios.

The inclusion of this group in the literature view, is due to two distinct reasons. First, their near-optimality in communication cost can be used as a benchmark for other protocols. At the same time, their limitations, can be used as a motivation for the exploration of other protocols.

2.3.4 Rateless and multi-replica set reconciliation

A great improvement in set reconciliation protocols' performance, both in computational and communication cost over the past few years has been due to the introduction rateless or adaptive approaches that do not require previous context before reconciliation and therefore no previous protocol configuration.

Furthermore, there has been significant work related to applying, the until then, pairwise protocols to multi replica environments with networks of different topologies that directly impact protocols performance. In scenarios with more than two replicas, the reconciliation problem's objective shifts from resolving a single pairwise discrepancy to eliminating global disagreement across an entire network. Many factors influence this process, from the protocol's efficiency to the network topology, making it crucial to be a thematic of papers included in the literature review to ensure this thesis is up to date with the last updates in this area.

This group of papers will contain the works that relate the most to this thesis' work making it of upmost importance to be expanded on the state of the art chapter.

2.4 Summary

This chapter presented the structured process used for collecting and screening the relevant literature to correctly create the state-of-the-art and to develop a good solution to the problems presented in the Introduction chapter. The querying process over well defined and referenced databases, alongside with some exploratory search from the initial set of papers, ensures it is fully reproducible with similar results.

Chapter 3

State of the art

3.1 Foundations of Set Reconciliation

The set reconciliation problem is very common in distributed systems mainly when it comes to synchronizing replicated data held by multiple parties. Formally, given two or more party each holding a set of elements, the problem can be defined as determining the differences between the sets in order to enable all parties to achieve a common state while minimizing communication overhead. [7]

For this paper, the difference between sets will be defined as follow. Given S_A and S_B that denote the sets held by two replicas A and B, the difference between the replicas will be calculated by the symmetric difference $\Delta = (S_A \setminus S_B) \cup (S_B \setminus S_A)$[7][4]. Due to real-world network issues and bandwidth constraints, where replicas may consist of large datasets that only differ by small changes originated from temporary network partitions or propagation delay, transmitting the entire datasets to reconcile states becomes heavy on the network making this approach obsolete [7]. Efficient reconciliation protocols, therefore are required to achieve communication overhead proportional to $|\Delta|$, rather than to the involved sets size $|S_A|$ or $|S_B|$.

Up until the release of Difference Digest in 2011 [4], set reconciliation techniques relied on replicas having access to the prior changes context. Subsequent work shifted focus toward reconciliation without relying on a shared state between replicas, removing the need for sharing logs or version vectors. This change made set reconciliation protocols must more robust to node churn, reducing the barrier for new nodes to be added to a system, whether due to intermittent connectivity or partial network failures, both very common issues in decentralized systems. [4]

Most set reconciliation solutions fall under two fundamentally different categories, either they are protocols based on ECC [2], which essentially reduce the reconciliation process to decoding a sparse error vector between two encoded representations of similar sets, or they are rely on probabilistic data structures to reduce the computation strain caused by the first group, at the cost of requiring extra communication rounds or additional metadata transmission [6].

3.1.1 Probabilistic data structures in reconciliation

One of the great advancements in this area was the use of probabilistic data structures for a more compact representation of large sets, that also contributed to the reduction of the computation overhead in reconciliation algorithms. Early implementations, used BFs [1][12], which laid the foundation for later developments such as rateless BFs [15] and other frameworks [6] which were later improved upon with the use of IBLTs [8] among other improvements.

In 1970, Bloom introduced BFs a space efficient probabilistic data structure that allowed for approximate membership queries while utilizing only a fraction of the space regular approaches require [1]. The trade off is the probability of false positives. In a BF elements are represented by a fixed-size bit array of length m and k independent hash functions. Insertions are processed by setting multiple bits corresponding to the hash function outputs, and queries check whether all associated bits are set. This approach allows no false negatives but allows false positives, with the error probability determined by parameters m the filter size, number of hash functions k, and number of inserted elements.

Following this, BFs can be used in synchronization schemes where one replica sends a BF summarizing its set to another replica [1]. The receiving replica can then test its own elements against the filter to identify candidates that may be missing from the sender's set. However, in order to compensate for the uncertainty caused by the false positives, additional communication rounds are required making them inefficient. In addition, BFs do not support element deletion, making them not suitable for exact set reconciliation[6].

IBLTs , extend BFs to overcome these limitations. Similarly to BFs, IBLTs distribute elements across a table using hashing functions, storing alongside each cell additional information: a count, an idSum and a hashSum. Each element is mapped to k cells using hash k independent hash functions, but differently from BFs, when an element is inserted its identifier is XORed into the cells' idSum, the element itself is hashed and XORed into the hashSum field and the count is incremented by one. Removal does the same operations, only decrementing the count instead of incrementing it. This extra storage overhead allows IBLTs to support insertion, deletion, and the recovery of individual elements through an iterative decoding process called peeling.[8]

This peeling process depends on the concept of a pure cell. A cell is said to be pure when its count is exactly one. In such a cell, the element identifier and element Hash can be recovered directly. Once an element is recovered from a pure cell, it is possible to remove it form all other cells to which is being mapped. Such operation may lead to creating other pure cells enabling an iterative decoding process that continues until either all elements have been recovered or no more pure cells remain in case of a failed decoding.[8]

3.2 Efficient synchronization of state-based CRDTs

State based CRDTs is a data structure used to reconcile loosely consistent data that allows replicas to record updates locally and later merge those changes asynchronously. Traditional versions of

these protocols require exchanging entire states to reconcile divergent replicas, which incurs a large communication and computation overhead as the data size and number of replicas increases, leading to very poor scalability.[3]

Delta CRDTs were introduced as an answer to this problem. The idea is to reduce the reconciliation cost by transmitting only incremental changes, known as deltas (δ) instead. This improvement however is not enough, as study show, existing delta propagation algorithms fail to deliver the expected improvements over naive approaches sometimes even incurring a higher overhead. The authors identify the two main causes for this:

- **Back Propagation:** replicas send data back to neighbors they originally got data from increasing redundant transmissions.
- **Reconciling old information:** There is no mechanism for assessing if received deltas contain information propagated by other nodes, which leads to cycles of redundant transmission cycles in non-tree topologies scenarios.

The authors address this issue with a synchronization framework based in lattice theory [8]. Each CRDT state is modeled as a join-semilattice which in turn can be uniquely decomposed into a set of join-irreducible elements. These elements represent the smallest indivisible components of the state, or in other words, they are the minimal information required to represent the evolutions of the state. This decomposition then permits the creation of an optimal delta function $\Delta(a,b)$, which computes the minimal state partition that when joined with state a, yields state b, whose function is to ensure synchronization messages only contain strictly inflationary state, as described by the authors [3]. This delta function is then used to solve the Redundant State issue, by applying it upon receiving a delta group, and only merging the result into the replica's state, mitigating the issue of merging redundant states, which proves to be particularly important in gossip or topologies where cycles are unavoidable.

The authors also present a simple work around for the back propagation problem by associating each delta entry with their original replica, avoiding delta retransmissions to their original replica.

The study's experimental evaluation which included tests on multiple topologies and applications, including a Retwis based social networking workload, showed significant improvements, reaching bandwidth reductions of up to 94% and Central Processing Unit (CPU) overhead reduction by a factor of up to 7.9in high replica divergence scenarios when compared with classical delta CRDT synchronization, by eliminating redundant message processing [3]. The results were also positive in memory usage as only non-redundant state portions were being maintained. One other advantage of this approach was the low metadata overhead when compared to vector based synchronization protocols [8].

This work provides a good example of topology aware synchronization, and common computational and communication bottlenecks these types of protocols may find.

3.3 Difference Digest: IBF with estimator

One of the main problems with set reconciliation protocols is the difficulty associated with computing the symmetric difference between sets without any prior context or coordination. Difference Digest addresses this issue where keeping logs or a stable shared history is impractical, providing good efficiency in scenarios where d is small relative to the total set size [4].

Traditional mechanisms either rely on probabilistic data structures with a large overhead due to the storage of metadata such as BFs or are dependent on ECC approaches that reduce the communication overhead to $O(d)$ at the cost of cubic decoding complexity $O(d^3)$, which becomes unfeasible in large sets scenarios. The idea behind *Difference Digest* is to combine the best of both approaches.

At the core of the Difference Digest protocol is an enhanced version of traditional BFs, the IBF. This version is very similar to the already mentioned IBLT, where each cell stores an XOR sum of all element identifiers mapped to it, an XOR sum of the hashes of all the elements mapped to it and a count of how many elements are mapped to that specific cell. By keeping this extra metadata, IBFs are able to remove their own coded symbols from a IBF sent from another replica through the decoding process known as "peeling" [8] [4], (see Section 3.1.1).

Selecting an appropriate table size, is the biggest challenge for this approach since it depends on the unknown size of the symmetric difference between the sets d. To mitigate this issue the authors introduce the Strata Estimator, a hierarchical structure composed of multiple IBFs with exponentially decreasing membership probability. Firstly the universe of keys is stratified into $L \approx log(u)$, in practice this is achieved by separating elements based on the number of trailing zeros in the element's hash value. A small fixed-size IBF is then created and populated based on that criteria. Forming L strata in a hierarchical manner where the highest strata has an IBF with very few elements mapped whereas the lowest is very densely populated. The estimation process works as follows:

Replica A builds its stratum and sends it to Replica B. Upon receiving it , replica B calculates the difference between its set and each stratum starting from the highest level, the easiest to decode due to its IBF's low element density, in search for the lowest level where the peeling process is still successful without needing extra coded symbols. Once that point is reached, the estimation final result is calculated by scaling the amount of retrieved elements so far by 2^{i+1} where i is the levels id used to form the stratum in the beginning.

The estimator provides an accurate estimate of d, even when d is very small. For larger differences, the authors propose an hybrid estimator that combines strata sampling with Min-wise hashing, ensuring accurate estimation across a wide range of d, which in turn enables the IBF to be parameterized correctly and the set reconciliation process to be completed in a single round of communication[4].

This protocol was tested in a linux service [4], *KeyDiff*, whose objective was to perform set synchronization over Transmission Control Protocol (TCP). The results it provided show that Difference Digest achieves communication complexity proportional to d. In most of the cases, it

sized the IBLT to approximately twice the estimated difference and achieved a decoding success rate of 99% with a single round of communication.

3.4 Rateless reconciliation and Bloom filter prefiltering

One of the downsides of using traditional BFs is the need to define the False Positive Rate (FPR), dependent on m, the number of bits in the bit array, k the number of hash functions and n, the number of elements in needs to support. This rapidly becomes inefficient as it requires an accurate estimate of the size of the symmetric difference (d) between both sets.

Rateless Bloom Filters (RBFs) are an improvement on the traditional BF design, created to adapt to an unknown symmetric difference between two sets. This approach inspired by Mullin's work [12] on multi-filter Bloom joins consists in having the sender continuously stream a sequence of small slices with optimal density, each generated using a single hash function. Upon receipt of each slice, the receiver assesses whether the stream should be terminated based on whether the benefit of identifying new false negatives is outweighed by the communication cost associated with transmitting more slices rather than just reconciling the rest of the elements directly. Effictevly mitigating the need for setting up parameters beforehand while still taking advantage of the BF bandwidth efficiency in high scenarios where d is large.

As it was stated before, BFs and therefore RBFs, are not capable of ensuring full synchronization due to the possibility of false positives. Therefore, in order to achieve exact reconciliation, the RBF is used only as a preliminary stage in a hybrid protocol completed by Rateless Invertible Bloom Lookup Tables (RIBLTs). These are used specifically for not requiring a prior estimate of d similarly to RBFs. In this case since the "peeling" decoding process is iterative, the sender streams their coded symbols until the receiver has enough information to decode the symmetric difference, removing the need for retries for over optimistically parameters, or avoiding communication of excessive coded symbols, until complete reconciliation is achieved.

The hybrid RBF-RIBLT approach presents several improvements over using either BFs or IBLTs rateless or not in isolation [5]. Traditional BFs may need to send 7x more metadata if the FPR is poorly chosen, whereas RBF, despite not requiring any type of prior context can adapt to the similarity level dynamically.

One of the major issues with solely using RIBLTs for set reconciliation is the constant overhead caused by all the fields each cell contains[6]. By using a RBF to filter most common elements first, metadata communication is reduced by up to 92% for large symmetric differences when compared with pure RIBLTs [5]. The communication cost reduction is also visible staying from 20% to 30% lower when compared with pure set reconciliation algorithms as similarity between the two replicas increases, making it the most efficient approach for similarities below 85% [5].

3.5 PinSketch

Rateless BFs and IBLTs based protocols are not the only bandwidth efficient solutions to the set reconciliation problem in distributed systems that have achieved communication complexity proportional to the size of the symmetric difference of both replicas. Despite being computationally heavy and being limited by a certain difference size threshold, PinSketch has shown that secure sketch approaches can be optimal for set reconciliation.

PinSketch is a secure sketch construction designed specifically to calculate and nullify set difference across two replicas in exponentially large universes with flexible set sizes as long as each element has the same fixed size. Marking its improvement on previous work "fuzzy vault" by Juels-Sudan [9] which required fixed set sizes. Its architecture is based on the application of Bose–Chaudhuri–Hocquenghem (BCH) error-correcting codes over characteristic vectors.

In a large universe, assigning each each element with a bit would be computationally unfeasible. PinSketch's work around this issue is computing the syndrome of the set instead, assuming it as its representation allowing it to be manipulated in sublinear time relative to the universe size.

The construction consists of two primary procedures sketching and recovery. Assume Alice with replica w and Bob with replica w' with $|w\Delta w'| < t$ t being the threshold for PinSketch decoding process to be successful. In the sketching step, a sequence of power is calculated in the finite field $GF(2^m)$, as follows $(s_i = \sum_{x \in w} x^i)$. The resulting sketch is composed of these syndrome values: $(s1, s3, \ldots, s_{2t-1]})$. The recovery step is where the Bob receives the sketch from Alice given Bob's noisy set w' and the sketch of w, Bob calculates their own syndrome and extracts the syndrome of the difference (σi) Using efficient decoding algorithms such as $Berlekamp's$ or $Euclid's$, the system identifies the symmetric difference $w\Delta w'$ in polynomial time[2].

PinSketch is recognized as an essentially optimal construction regarding resource efficiency as its storage requirements and entropy loss are limited to tlog(n+1), which matches the theoretical lower bounds for the set difference metric in large universes [2].

Another improvement Pin Sketch presented over other reconciliation protocols such as Characteristic Polynomial Interpolation-based Synchronization (CPISync) [14] is its computational advantage. Where early methods required $O(t^3)$ time to solve systems of linear equations [2] for what PinSketch resorted to the Euclidean algorithm for decoding, reducing its time complexity to $O(t^2)$, making it highly suitable for real word applications such as PGP key server updates [2].

Despite its improvements on previous reconciliation methods, Rateless BFs element size flexibility and set difference wider range make it a better fit for most systems, reason why it is the focus of this thesis.

3.6 Parity Bitmap Sketch (PBS)

3.6.1 Motivation

Parity Bitmap Sketch (PBS) is another state of the art set reconciliation scheme with the same goal of efficiently finding the symmetric difference between two divergent replicas. This solution proposed

to solve a trade off in set reconciliation algorithms up until them, where communication efficiency was not achievable without gravely affecting the computational complexity of the algorithm in PBS in scenarios with large set differences and unconstrained universe size [7].

Prior to PBS the existing solutions to set reconciliation fell into two distinct categories. One one hand, there were probabilistic data structures IBLTs, as described above these achieved linear computational complexity $O(d)$ relative to the set size difference d but with high communication overhead due to the structure's fields and the occasional decoding failures for a bad estimation of the set difference and consequently transmission of insufficient coded symbols to the second replica. On the other hand, there were coding theoretic approached such as Pin Sketch that offered communication costs close to the theoretical lower bound, with the downside of requiring $O(d^2)$ decoding computation complexity, rendering them utterly impractical in scenarios where the replica divergence is large. The motivation for the development of PBS was exactly to achieve a reconciliation algorithm that presented with a low communication overhead while still maintaining a linear computational complexity effectively getting the best of all solutions, while also making itself robust against decoding failures [7].

3.6.2 Design

The PBS core design consists in partitioning the sets into g smaller groups of size dependent on the size of d using hash functions and then recovering them using BCH ECCs. This strategy arises from the increase in computational complexity in ECCs as the set difference size increases. By keeping the size of the sets BCH works on small, its decoding complexity can be reduced to $O(1)$, reducing the overall decoding process complexity to $g * O(1) = O(d)$ since g is proportional to d instead of the usual $O(d^2)$ in solutions based on ECCs [7].

The first step is to make an estimation of the size of the set difference using a lightweight estimation technique a,Tug-of-War as an example. This estimation enables the protocol parameters to be configured dynamically, working around the lack of prior knowledge about the value of d, commonly unknown in distributed systems environments and allowing the average number of differences per group δ to be set accurately [7].

After d is accurately defined, the set is split into g groups where $g = d/\delta$, using a hash function. In order to try and split the elements that belong to the symmetric difference, Each group is further subdivided into n smaller subsets using a second hash function. n is a parameter that must be tuned for maximum efficiency. For each group, each replica builds a parity bitmap, in other words, a binary vector in which each bit indicates whether each subset contains an odd or even number of elements. BCH is then used to encode the bitmap into a sketch that is to be sent to the other replica. This way, we avoid sending the entire bitmap while still ensuring the set difference can be retrieved by the other replica.

Upon reception of a sketch, the receiver's job is to compare its local bitmap and identify every subset with different parity, signaling elements belonging to the set difference. Once identified the, the XOR sum of the subset's elements is computed and transmitted back to the original sender. At which point it can recover and reconstruct any missing elements by subtracting its contributions.

3.6.3 Decoding robustness

A major improvement on previous work PBS presents is it's blocked approach to reconciliation. Since each BCH coded block is only associated with one independent partition, the decoding success or failure does not affect others, making the retrieval of most symmetric difference's elements successful in the first rounds. Results show more than 95% of said elements are retrieved in the first round, with additional rounds incrementally resolving remaining holes. Furthermore, by using an analytical framework to tune the protocol's parameters such as the bitmap size n and the error correction capacity of the ECCs such as BCH, PBS is shown to have high success rates within a small number of rounds, commonly three, even for large replica divergence scenarios[7] .

3.7 Multi-party set reconciliation and topology-Aware coordination

In modern distributed systems often rely on vast amount of data being managed across disperse locations. Most of the times, as high availability and fault tolerance are the top priorities, these types of systems are forced to maintain multiple loosely consistent replicas of their data.

The fundamental framework used to address replica divergence issues is set reconciliation, not in its simplest form with only two participants where two parties Alice with set S_A and Bob with set S_B aim at reaching a state where each converges their set to the union $S_A \cup S_B$, but in multi party scenarios where each party holds a different replica S_i and has a goal of reaching $\cup_i S_i$ [11]. Moreover, the amount of replicas data is scattered around many users each with often more than one local replica of it across many devices[4], that may or may not be frequently connected to the network[6], increasing the range of state divergence across the board . This alongside the proportional growth of users with access to shared information, only complicates the synchronization process. Making it crucial that the communication cost of the protocols used for such task is optimal and dependent on d rather than the size of the dataset themselves [7].

The most intuitive and naive approach to tackle this problem which has been the base of some implementations, is to apply the most efficient pairwise protocols such as PinSketch [2] or RBFs [15] between all replicas until complete reconciliation is achieved. While this provides a simple and functional baseline model for a solution, it introduces significant inefficiencies that make it unfeasible in large scale. For instance, in a three replica scenario involving nodes A, B, and C, node A might first reconcile with B, followed by B reconciling with C. This comes with quadratic message complexity, where in a network of n parties, completing the reconciliation problem would take up $n(n-1)$ point to point messages for each replica to compute their own differences against every other replica [11]. It also fails to fully utilize the linear properties presented by modern pairwise solutions. When we treat each interaction as an isolated event, we are stopping the system from aggregating multiple sketches into a single global summary of the set difference. Another major issue with this approach is the lack of acknowledgment of the network topology effect on the communication cost without proper changes to the pairwise algorithms [11]. In many scenarios, synchronization messages have to pass through intermediate nodes or relays. Following

the pairwise strategy using polynomial algorithms, each relay is forced to fully decode a sketch, identify the differences and then generate a new sketch for the next hop, encompassing a $O(d^2)$ for polynomial-based methods [2], effectively introducing latency when compared to strategies where relays are only required to perform mathematical summations.

In order to prevent this bottleneck from happening, we can use probabilistic structures that possess the summation property ideal in this case of scenario where adding together the sketches of many sets outputs the final total difference among them. First we need to change the IBLT implementation to go from binary data structures leveraging in order for them to be able to identify the set difference effectively with more than 2 replicas [11].

Right now, the chosen operation for assessing ownership in these kinds of sketches is XOR. Despite its efficacy in tracking the parity of an element's presence it fails to distinguish different contributions when more than 2 replicas are in play. As an example, in traditional implementations, a element present in 2 out of 3 sets would be incorrectly registered as not part of the set difference. Mitzenmacher and Pagh show that this the upgrade on this algorithm is possible by shifting from a binary operation to a structure that treats hash keys as vectors over a finite field F_p where p represents a prime number so that $p \geq n$, n being the number of parties to be handled. This occurs through an injective mapping transforming the insertion and removal process into modular sums and subtractions. The great advantage is the linearity property of the sketch, by operating over vectors, any linear sketch combination, results in a global sketch that represents the same linear combination of the original sets, effectively preserving information about multiplicity that would have been lost using XOR operations. The linearity of sketches over F_p enables two properties [11]. Let each replica have a sketch v_i, the sum $L = \sum i = 1n\alpha i v_i$ is the sketch of the aggregated information of all participants which provides two properties:

- **Global canceling:** Elements present in every set eg.: $i \in (S_1 \cap S_2 \ldots S_n)$ will be automatically eliminated in the final sketch given the coefficients are correctly chosen (ex: $\sum \alpha i \equiv 0 \pmod{p}$. This result can be achieved without coordination among all replicas. For instance, any replica A_j that obtains the unweighed sum of all sketches $L = \sum_i v_i$, can subtract its own local sketch, by applying a calculated factor, for example computing by $L + (p-n)v_j \pmod{p}$
- **Multiplicity recovery:** Contrary to the XOR approach, together with changing the count value to work under module p, we can identify exactly the number of replicas the element is in, making the decodification operation easier even in.

Hand in hand with the change to the hash_sum field the count field must change as well, to support the number of replicas. Each cell's count field now works under module p. A cell is considered pure if it contains contributions from only a single distinct key, regardless of the count being larger than one. Let a denote the multiplicity of the key x encoded in the cell. We can retrieve the key by calculating the modular inverse $a^{-1} \pmod{p}$ and applying it to the stored key and hash sums field. After checking its consistency against the cell's field the peeling process can proceed by,

similarly to the traditional method, removing the hash from every cell it is mapped to potentially revealing additional pure cells along the way [11]. By resorting to this method, we are taking advantage of two main factors.

1. **Decoding in linear time:** Despite its complex algebraic base, the peeling process is still efficient, always operating in linear time proportional to d, $O(d)$ or $O(dlog(d)$, showing a large improvement over the $O(d^2)$ decoding time normal in characteristic polynomial based methods such as *PinSketch*.

2. **Communication cost:** By adapting probabilistic data structures to produce sketches capable of recovering multiplicity while maintaining linearity and summation properties, we can avoid the quadratic growth associated with pairwise reconciliation protocols, while also improving efficiency on relays.

Chapter 4

Approach

4.1 Overview

This thesis studies the set reconciliation problem in multi replica environments and aims at answering the questions of how an efficient strategy for that setting can be found. Instead of designing by hand every protocol, it transforms the problem into a search over configurations of reconciliation protocols and uses a loop driven by an agent to explore that space, letting an efficient scheme emerge from measured performance rather than from a prior design. The starting point is the state of the art in pairwise reconciliation, the hybrid approach with RBF and RIBLT has shown near optimal communication cost in scenarios with only two replicas [5], but its behavior beyond tow replicas is largely unexplored. These pairwise techniques serve as baselines against which the final discovered configuration can be compared.

Both the search and its evaluation are based in a controlled experimental framework, or harness, that simulates the set reconciliation process under a range of topologies, workloads and degrees of state divergence. This evaluation framework is implemented in Rust due to its low level performance and memory safety that make it well suited to implementing highly efficient code with little computational overhead in addition to the protections from its type system.

Every simulation is run on a single machine mimicking real network communication, this is, every message's byte size is accounted for as if it was serialized and by giving a measure of the communication cost independent from the transport layer. This makes the bandwidth objective that drives the search a faithful encoding of its efficiency on a real distributed deployment.

4.2 System Model

For each measured run, N replicas, each holding a set S_i will be simulated, the goal is for each replica to reconcile its state with every other replica, or in other words, to converge their state into the union of all the other sets, rendering the symmetric difference between each of them null. A run is only considered successful if and only if the following condition is met.

$$\exists t < \infty : \quad \forall i \in \{1,\ldots,N\}, \quad S_i(t) = \bigcup_{j=1}^{N} S_j(0)$$

Each simulation will work under the assumption that there are no nodes with byzantine behavior as fault tolerance is outside of the scope of this thesis. Furthermore we will assume each replica won't change its state once the reconciliation process begins. This assumption keeps the system realistic as it is achievable in production scenarios by creating a snapshot of the current state once the process begins. Across every run, the network topology will guarantee every replica is accessible.

Both the communication and computation cost will be measured in a systematic way. Communication cost will be measured as the total number of bytes transmitted, this encompasses both protocol metadata and serialized elements. The computation cost is measured as the time spent in encoding and decoding operations by each replica.

4.3 Baseline protocols

This section describes the reconciliation protocols that will be used as baselines, against which the discovered multi-party strategy (6.2.1) is evaluated. They span from a naive upper bound on communication cost to the pairwise state of the art. All these protocols are run against the same matrix as the discovered configuration so that the comparison is made under identical conditions

4.3.1 Hybrid rateless Bloom filter with rateless IBLT

This protocol is included as it was the protocol that led to this thesis. Initially, the goal was to find a multiparty variant of the algorithm that kept its advantages from pairwise reconciliation. It represents the state of the art for pairwise reconciliation algorithms for sets with elements of varied size, without neither prior context nor any kind of parameter tuning. It takes advantage of a probabilistic first phase that uses a RBF to find likely common elements. It then utilizes a RIBLT to deterministically finish the reconciliation process once the cost of reconciling confirmed negatives is outweighed by the cost of reconciling those same elements.

4.3.2 RIBLT

The purely RIBLT protocol will also be implemented and monitored as a baseline to assess the contribution of the RBF prefiltering in the first approach. This protocol will only use a RIBLT incrementally transmitting coded symbols from one replica to the other until the latter's peeling process succeeds, ensuring exact reconciliation.

4.3.3 Static Bloom Filter with IBLT

Despite not being up to date with the state of the art, this protocol will still be tested in order to compare the cost in protocols that require prior knowledge of d and those that don't (4.3.1).

The protocol must first estimate d in order to accurately parameterize both the BF and the IBLT. Similarly to the hybrid rateless approach, the BF is used to find common elements beforehand, improving the IBLT's efficiency in retrieving the rest of the symmetric difference elements.

4.3.4 Full state transfer

This naive approach, despite simple to implement and inefficient when compared to others, will be simulated as an upper bound on communication cost and therefore provide a reference point against which the rest of the protocols can be evaluated. Furthermore it will be used by the agent loop's search (5.12) as a starting point.

4.4 Multi party coordination strategies

This section focus on assessing the impact on communication efficiency, coordination has when performing set reconciliation in different multi replica scenarios with various topologies and network latencies. Coordination is the dominant factor on communication cost at scale, since a naive scheme resends the same elements along every path. Two strategies are considered, including the all neighbor baseline and the topology-dispatched strategy discovered by the search.

4.4.1 All neighbor dissemination

This is the naive approach that will be tested as the baseline model where each replica will independently reconcile with every one of its neighbors. As mentioned in the state of the art it is expected to be communication heavy with a lot of redundant transmissions, as the same element reaches a node from multiple paths. For this reason its use is to define the upper bound on coordination overhead. The baseline protocols (5.10) are run under this scheme.

4.4.2 Topology aware dissemination

Beyond the baseline approach, the search may vary how dissemination is coordinated and may do so independently for each topology. The objective of this thesis is to understand how good of a fit is an autonomous loop for research in the distributed systems area, and by allowing these large decisions to be made by the loop itself rather than by hand, the coordination strategy becomes a direct test of the depth the agent can reach. The core the agent works within is a round based sketch and deliver loop, from there it can choose the sketch, to which neighbor it sends and which messages to absorb, for instance, all of them or only those at exponentially increasing distance. These together with the sketch's parameters namely the BF FPR and whether a round additionally carries a RIBLT to resolve the residual differences of large sets. This wide search space left to the agent, is what makes the found configuration insightful on how valid this approach is for search. The specific final configuration and its effect on communication cost are reported in chapter 6.

Table 4.1: Experimental matrix. The first group lists the axes varied across runs; the second lists parameters held fixed throughout.

Group	Parameter	Values
Varied	Topology	Star, Tree, Chord
	Jaccard similarity (J)	0.25, 0.5, 0.75
	Number of replicas (N)	16, 64
Fixed	Replica set size ($\|S_i\|$)	10 000 elements
	Element universe	500 000
	Workload distribution	Zipf, exponent 1.0
	Payload / digest size	32 B payload, 64-bit digest
	Divergence pattern	Uniform
	Runs per cell (seeds)	3 (42, 43, 44)
	Round cap	100

4.5 Experimental matrix

The evaluation is to run every protocol over a fixed matrix of conditions, summarized in Table 4.1. Each will be tested against three varying axis, including the network topology, similarity between replica states and the number of replicas, while every other parameter is held constant so the three factors are the only variables that may impact the final result. Make note that the parameters internal to the protocols are deliberately absent from this matrix as these are one of the conditions the search optimizes and the configuration arrives at, as reported in 6 .

4.5.1 Varied axes

The protocols will be evaluated in an environment with $N \in \{16, 64\}$, which represent mid and large scale point whose objective is to expose how each protocol's properties handle different scales.

On top of that, three topologies are used to mimic a wide range of possible structural characteristics that may be found real distributed deployments. These are Star, whose goal is to represent a central bottleneck, Tree, that represents a decentralized coordination with logarithmic depth and Chord that represents an even more decentralized structure with cycles that was chosen over a gossip network since the latter's considers node churn and volatility that are outside of the scope of this thesis.

Finally, state divergence is controlled through the Jaccard similarity index $J \in \{0.25, 0.5, 0.75\}$ between replica sets, that ranges from substantially divergent ($J = 0.25$) to largely agreeing ($J = 0.75$) states, since the amount a protocol must reconcile is exactly what its efficiency should be measured against.

4.5.2 Fixed workload

All runs are to use the same workload model so that the varied axes are the only source of difference. Each replica draws a set of 10 000 elements from a universe of 500 000 under a Zipf distribution (exponent 1.0), giving a skewed, realistic popularity profile. Elements carry a payload with 32 bytes, a 64 bit digest and divergence is spread uniformly across the set rather than clustered. Holding the workload fixed means the set size,payload, digest and skew are not themselves studied here.

4.5.3 Execution

Each cell of this matrix is run three times under different seeds to characterize variation from run to run. As discussed in threats to validity (Section 6.5.2), a defect meant these three runs share a single workload instance, making the spread seen in results be a result of hashing nondeterminism rather than independent workload resampling. A round cap of 100 was put in place and runs to fail to converge in that period or are by themselves too slow to finish, are reported as non convergent and their data is retained as an informative outcome when comparing results and moving forward with the search loop.

4.6 Metrics

Each simulation will be evaluated based on the following metrics.

4.6.1 Communication cost

The total number of bytes transmitted during the reconciliation process. It includes all the coordination messages, BF slices and IBLT coded symbols which provides a topology independent metric that accurately translates to the protocol efficiency.

4.6.2 Computation cost

Besides communication cost, the computation's is another important cost to be measured. It is read as the wall-clock time spent in protocol encoding and decoding operations at each replica. Encoding time captures the cost of generating reconciliation data structures, while decoding time captures the cost of interpreting received data and reconstructing missing elements. These measurements allow evaluation of how reconciliation overhead scales with the number of replicas and the chosen coordination strategy.

4.6.3 Convergence behavior

In multi party scenarios, convergence behavior can be defined as the number of reconciliation rounds required until correctness is achieved. For pairwise strategies, a round consists on a complete execution of a reconciliation algorithm across all selected edges. When it comes to protocols that

do not require nodes to reconcile theirs sets two at a time, a round become becomes one full aggregation and decoding cycle, after which each replica has their local state updated accordingly.

In both cases, reconciliation is an iterative process, which facilitates the recording of the metric. At the start of which simulation, each replica i will be initialized with a set $S_i(0)$. The simulator then proceeds with reconciliation rounds until convergence is achieved at which point the measure is returned.

4.6.4 Correctness

This metric is evaluated by verifying all replicas converge to identical sets containing exactly the union of their initial elements, and provides a boolean value to whether the reconciliation was successful or not.

Chapter 5

Evaluation Framework

5.1 Design goals

The simulator was built from scratch instead of relying on a network emulator or a real deployment for three reasons: reproducibility, fair bandwidth accounting and guaranteed control over every protocol simulation.

5.1.1 Reproducibility

The reproducibility of the evaluation framework can be assessed by how close the results of each experiment are, given a fixed workload seed, topology, protocol and simulation seed across time. By creating my own simulator this can be a focus of the architecture, in contrast with testbed measurements where the Operating System (OS) scheduler can induce variance in the results that are not a direct result of the protocols' efficiency and therefore noise.

5.1.2 Fair bandwidth accounting

The goal of this thesis is to find the most bandwidth efficient protocol, starting from FST and from the Hybrid RBF and RIBLT protocol. Therefore, every byte sent or received by each replica in a protocol must be accounted for. This creates the need for a special data structure through which, all communication must pass. The simulator therefore centralizes byte accounting in a single network object that owns the counters and is the only place they are incremented. The mechanism is described in 5.4.

5.1.3 Protocol isolation

Building on the previous subsection (5.1.2), it is important that no protocol can have access to the full topology, as every replica would be able to skip the metered network and access other's inbox directly. In order to avoid this, each protocol is an implementation of the Protocol trait, in a way, each replica only has access to its own local state and inbox. Although this measure is justified on the correctness it ensures, its value is only magnified by its ability of preventing agent generated

protocols from inadvertently accessing the global state and producing invalid metrics and skewing the results.

Every architectural decision in the remainder of this chapter has the fulfillment of one of these three goals. The Architectural section (see 5.3), describes the modules responsible for enforcing them.

5.2 Non-goals

The question this thesis aims to answer is which reconciliation protocol moves the fewest bytes between replicas to reach a converged state. Anything that does not affect the answer to that question is deliberately out of scope, and modeling more effects would not make the comparison more honest but would instead add variance that obscures the signal the simulator is built to improve. This section describes and justifies the design decisions made.

5.2.1 Network latency and packet loss

The simulator does not aim at reducing network latency or packet loss nor it keeps track of it. These are properties of the deployment and not of the protocol. Due to this, it is assumed each protocol runs above a reliable transport layer, such as TCP or Quick UDP Internet Connections (QUIC), and each protocol only observes a reliable channel. This allows the autonomous search loop described in section 5.12 to focus on optimizing solely for reconciliation efficiency by not considering fault tolerance as a protocol property, leaving outside of scope retransmissions, acknowledgements and transport headers. Packet loss thus affects only the absolute byte total of a deployment but not the relative ranking by application bytes, which is the value this thesis optimizes.

5.2.2 Node failures and Byzantine behavior

Fault tolerance is a separate research question with its own metrics. Including it would require the parameters of the evaluation matrix to be wider.

5.2.3 Encode and decode time

The engine does record the encode and decode time per replica, which is the time it takes for the sketches to be built and decoded respectively. Even though these durations are stored and exported they are excluded as an objective. Since they are measured with a live monotonic clock (`Instant::now`) they are dependent on the machine and the OS scheduler and therefore not reproducible in the sense of Subsection 5.1.1. Folding them into the fitness function would reintroduce the measurement noise the simulator exists to remove, so they are kept as diagnostic fields only and never influence the search or the reported ranking.

5.2.4 Heterogeneous hardware

The simulator runs synchronously in a single process. This removes the scheduling jitter that would otherwise appear in the `encode_time` and `decode_time` measurements. Since these are non-goals so is testing on hardware with different capabilities.

Each exclusion is a deliberate scoping choice. The threats to validity that may arise from these choices are revisited in 6.5

5.3 Architecture overview

Inside the library module there are three input modules (`worload.rs`, `topology.rs` and `replica.rs`), are responsible for constructing the sets for each replica and their respective elements as well as the communication network graph. These feed the `engine.rs` simulation loop, responsible for driving every replica through each protocol's `send_phase` and `recv_phase` steps, defined by the `Protocol` trait. All communication between replicas passes through `network.rs`, the billing layer, which is the only place the byte counters are incremented, these counters are then used by the engine to record per-round results into `metrics.rs`. Furthermore, the library is setup so reusable components live in `algorithms/`, so adding a new protocol is a question of orchestration rather than reimplementation. Figure5.1, shows the modules and the relationships between them. The following subsections describe each module in the order a single simulation run uses them.

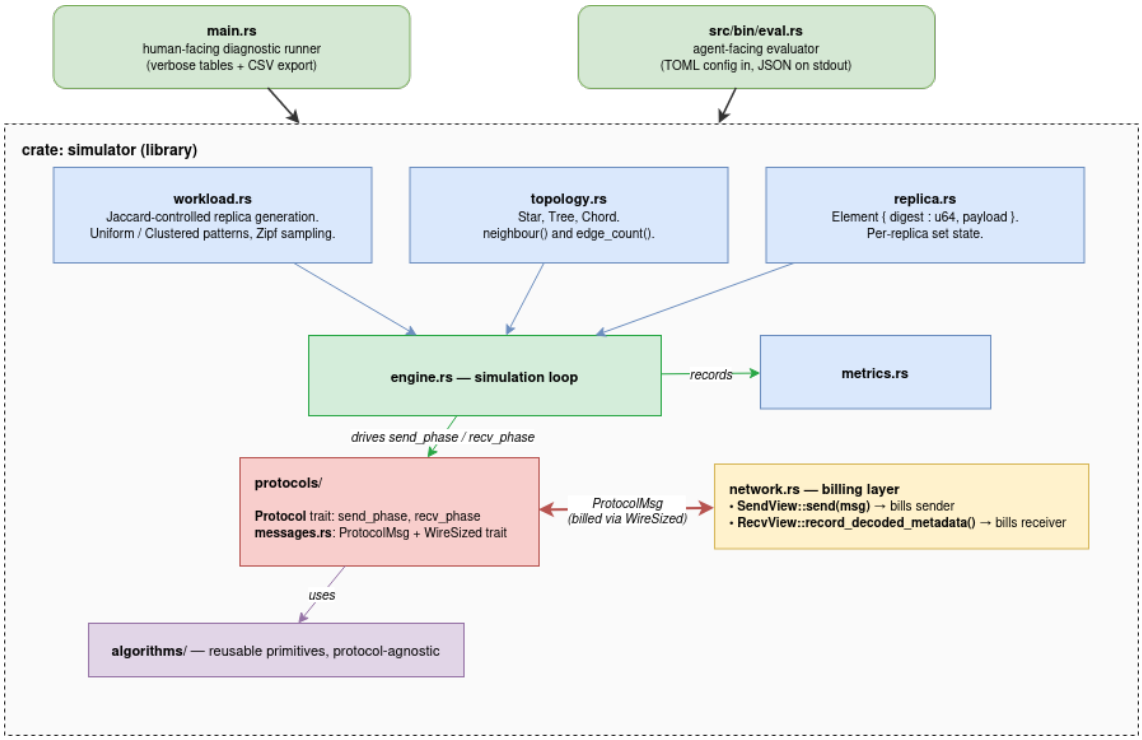

Figure 5.1: Simulator architecture

5.3.1 Binaries

The simulator is a single library crate exposed through two binaries: `src/main.rs` a human facing diagnostic runner that prints verbose tables and appends to `results.csv`, and `src/bin/eval.rs`, an evaluator dedicated to the agent, that reads a Tom's Obvious Minimal Language (TOML) configuration and emits a structured JavaScript Object Notation (JSON) summary on stdout. This split exists only to serve two different types of readers, as both binaries are dependent on the same library modules and produce the same `SimulationResult` data structure.

5.3.2 Library modules

5.3.2.1 The workload module

`workload.rs` is responsible for producing the per-replica initial sets the engine will later reconcile. This is done through the `Workload::generate(&config)` that receives the desired configuration as a argument. The configuration itself is held by the struct `WorkloadConfig` and consists of the number of replica, the per-replica set size, the divergence pattern (uniform or clustered), the universe size and Zipf exponent that govern element sampling, and lastly, a seed. This only constructor, outputs the `replica_sets`, which is a `HashSet<Element>` that represents the initial set of each replica and it also computes the `target_union` which is the set every replica must eventually hold for a convergence to be considered successful.

This module guarantees that generatoin is a pure function of its `WorkloadConfig`, which means that for the same config, the same sets will be returned regardless of the machine or environment it is run on. This is the property that makes a simulation run reproducible, as defined by the reproducibility goal in 5.1.1. The reasoning behind deep design choices of the likes of why Jaccard Index and why zipf sampling are deferref to 5.7.

5.3.2.2 The topology module

`topology.rs` defines the communication graph the reconciliation protocols will use. It exposes a `TopologyKind` enum, which only supports `Star`, `Tree` and `Chord`, and a `Topology` struct built through a construct that takes the kind of the topology and the number of nodes. This constructor then dispatches to one of the three specific constructors, responsible for creating an unidrected graph with a set number of nodes with no self loops. The three shapes and the reasoning behind them are discussed in 5.6.

Once built, a `Topology` is immutable. This module can only be interacted with through a set of read-only accessors that allow protocols to find out which nodes they can communicate with, find out the node and edge count and in the case of the tree variant, if a given replica is root, find their parent and children. It is important to note, the topology only exposes node identifiers and not their contents, making it so a protocol holding a reference to a `Topology` instance cannot use it as a backdoor to acquire information about its peers' state, contributing to asserting the protocol isolation design goal, as detailed in subsection 5.1.3.

5.3.2.3 The replica module

`replica.rs` defines the data a single node holds in every step of a simulation. The `Element` struct is the atomic unit reconciled across the simulated network, it is composed of a `u64` digest and a `Vec<u8>` payload with a `wire_size` method that reports the on-wire cost (eight bytes for the digest plus the payload length) that the billing layer uses to charge a protocol when an element is transmitted. The `Replica` struct wraps a node identifier, a `HashSet<Element>` carrying its current set, a `ReplicaPhase` responsible for tracking its status in an ongoing simulation and a `ReplicaStats` record that accumulates every metric the engine may later read, which include state bytes sent, metadata bytes sent, encode and decode time and the number of elements added during the run. Handing a protocol a full `Replica` reference would compromise the protocol isolation stated in 5.1.3. Therefore a separation is enforced by the type system. The engine owns every `Replica`, but the methods on the `Protocol` trait never receive one. Instead, before each step in a simulation, the engine constructs a fresh `ReplicaView<'a>` carrying only the node's `id` and a shared reference to its elements `HashSet`. Engine-facing fields like `stats`, which would let a protocol read or alter its own billing record, and `phase` are absent from the view, so no cheating surface is available for a protocol. This same mechanism also prevents lateral access, since `ReplicaView` borrows from a single `Replica`, a protocol holding one view has no path to any other replica's set, only its own.

5.3.2.4 The engine module

`engine.rs` owns the simulation loop. It executes simulation runs based on a static `Simulation-Config` with its own workload, topology, protocol, round cap, and seeds. This configuration is set upon the calling of the method `Simulation::new(config)` that creates the `Simulation` struct all the protocols will use for the reconciliation process. A Simulation has two driver methods: `run()` and `step()` that execute rounds until the simulation terminates or advance exactly one round and returns a `RoundOutcome` of `Continue`, `Converged` or `RoundCapReached` respectively. Once the simulation terminates, the engine produces a `SimulationResult` carrying the round count, the topology and protocol identifiers, a convergence flag, and the aggregated `MetricsSnapshot` 5.3.2.7.

A single round proceeds in a fixed sequence. The engine first resets the network's per-round counters, then iterates over every replica and calls the methods required to be implemented by the `Protocol` trait: `send_phase` and `recv_phase` in order. During the sending phase, the engine builds a `ReplicaView`, a reference to the `Topology`, an `Outbox` bound to that replica's identifier , and the replica's own carry from the previous round. Each carry is moved out of the engine into exactly one `send_phase` call and never shared across replicas. Once every replica has sent, the engine drains all inboxes and calls `Protocol::recv_phase`, collecting each replica's returned `next_set`, `LocalMetrics`, and new carry. Metadata bytes are billed only after the receive phase completes, due to the fact that rateless sketches and BFs reach their final on-wire size only once the receiver has decoded against them. The engine then records per-replica byte

counts and encode/decode times into each `Replica.stats`, stores the returned carries back into per-replica slots, and updates each replica's set.

This same loop also enforces the monotonic union invariant 5.5.2, with an assert that verifies that every `next_set` returned by `recv_phase` is a superset of the previous set, so a protocol cannot fake convergence by deleting elements. At the end of each round, converge is checked separately when the engine compares each replica's set against the workload's already calculated `target_union` and returns an appropriate `RoundOutcome`: `Converged` if every replica holds the full onion, or `RoundCapReached` if the RoundCap is reached before the Protocol has the chance to finish. The engine never reads or writes a replica's set through any path other than the one described here, which is what makes a single `SimulationConfig` always output a single `SimulationResult`.

5.3.2.5 The protocols module

`protocols/` defines the trait every reconciliation strategy must implement and provides the implementations the evaluation matrix runs against. It exposes the `Protocol` trait, the `ProtocolKind` enum that identifies the implementation to be used at configuration time, and the results and metrics the engine consumes after every round. The trait exposes three methods. First, `kind()` returns the `ProtocolKind` the engine uses to label results. Second, `send_phase` which the engine calls once per replica per round to emit outbound messages, and lastly, `recv_phase`, which the engine calls following the previous phase to consume the inbox and produce a `ProtocolStepResult`.

A protocol instance is immutable across the entire simulation, both phase methods take an immutable reference to self rather than a mutable one. Any state that must persist between rounds is returned from `recv_phase` as a `PendingElements` carry and handed back to the same replica's next `send_phase` by the engine.

The `ProtocolStepResult` is the final output of a protocol round, it contains the `next_set`, which is the replica's new local set, a `LocalMetrics`, that keeps track of the encode and decode durations the protocol used, and an optional carry. The opaque payload emitted through the `Outbox` is a type of the `ProtocolMsg` enum and is defined in the billing layer 5.3.2.6.

The simulator ships five implementations of protocols. `full_state_transfer` is the upper bound for the rest of the protocols. Every replica ships its entire set to its neighbors.`riblt` reconciles pairs of replicas using RIBLTs, peeling the symmetric difference incrementally. `StaticBfIblt` combines both static BFs' membership queries with an IBLT-backed exchange of the elements once the filter's FPR starts degrading. `hybrid_rbf_riblt` replaces the static filter with its rateless counterpart, allowing the receiver to decide how many cells are needed. Lastly, `multi_replica`, is the only file the autonomous researcher is allowed to edit 5.12.3. The four baselines are frozen so that any change in fitness across iterations is attributable to the agent's protocol rather than to a moving comparison surface.

5.3.2.6 The billing module

`network.rs` owns every byte counter in the simulator and is the only place these counters can be changed. It consists of a `WireSized` trait that every transmittable payload implements, the generic network through which all objects implementing the `WireSized` flow, and the `Outbox` that allows protocols to send messages.

`WireSized`, ensures every Message has its own wire-cost accounted for. This cost is split into two, `state_bytes` and `metadata_bytes`. State bytes carry the actual reconciled elements and are billed by `Network::send` at send time, whereas metadata bytes that carry the sketches, filters, coefficients, and headers the protocols use to identify the symmetric difference are billed by `Netowrk::bil_metadata_post_recv`, called by the engine after `recv_phase` returns.

The `ProtocolMsg` type carried over the network is deliberately opaque. The inner enum is private and has no public constructor, in order to make it impossible for protocols to build a fresh `ProtocolMsg` from outside the module, mostly to prevent the replacement of messages for similar ones with less or zero cost, which would otherwise discard already billed wire bytes. Access from outside the module goes through sealed wrappers, one for each already implemented algorithm, each of which only exposes the operations the receiver legitimately needs (decoding, membership testing, extension, ...) and whose constructors are module-private as stated above.`WireSized` for `ProtocolMsg` dispatches to per-variant helpers that compute the actual byte cost of each sketch or filter from its state after the recv phase, so the bill always matches what would have crossed a real network.

5.3.2.7 The metrics and export modules

`metrics.rs` aggregates per-replica counters into a summary at the end of each run. This module exposes tow structs. First, `NodeMetrics` carries the totals for a single replica, including state bytes and metadata bytes sent, encode and decode time as well as elements added during the run. Second and last, `MetricsSnapshot` carries one `NodeMetrics` per replica together with the aggregate totals of the entire simulation as well as the round count. None of these methods are mutable, all they do is read the `stats` field off every `Replica` and sum the per-node values into the total, `MetricsSnapshot::from_replicas(replicas, rounds)` is the only way a snapshot can be built. The engine is responsible for calling it once at the end of a `run()` right after the last round has been billed. There is no other path through which a metric is changed outside the billing pipeline described in 5.3.2.6:

`export.rs` turns a `SimulationResult` into a flat `RunSummaryRow` suitable for emitting a Comma-Separated Values (CSV). The row holds the workload parameters that produced those run, which include the protocol, topology, replica count, set and payload size, digest width, Jaccard similarity index, divergence pattern, seed, round cap and so on. Run results expose the final round count, a convergence flag, the target union size, total state and metadata bytes, total encode and decode time and finally the total elements added. `RunSummaryRow::from_run()` builds the row from a `SimulationConfig` and a `SimulationResult`. The human-facing binary uses

this path to accumulate a per-run history across different invocations, whereas `src/bin/eval.rs` emits the same summary as JSON on stdout for the agent to better access the data and does not touch `results.csv` (5.3.1).

5.3.2.8 The algorithms module

`algorithms/` contains the reconciliation primitives the protocols can reuse. Unlike every module described so far, it is deliberately protocol-agnostic, it does not reference any other type defined in the simulator. A primitive is a data structure plus the operations a protocol would invoke on it during reconciliation. These primitives were adapted from the work of Pedro Gomes [6], whose implementations were reused as a validated starting point and modified to fit this simulator's message interface and byte accounting convention.

Five primitives are provided. `bloom` implements a classical BF for approximate membership testing. `rateless_bloom` extends the previous implementation into a rateless variant that allows the receiver to increase the length of the filter until a certain criterion is met, in order for the receiver to not have to commit to a FPR upfront. `riblt` implements a RIBLT that encodes a set as a stream of coded symbols a receiver can peel against one of its own to recover the symmetric difference incrementally.`multiparty_sketch` generalizes the RIBLT construction to accept contributions from more that two replicas at a time by replacing the binary operation XOR with finite-field arithmetic modular arithmetic. Even though is not used by any baseline protocol, it is provided as a starting point for protocols that may want to explore more than pairwise exchanges. Finally, the module also implements `bayesian_estimators` used by the rateless protocols to decide when to stop extending a sketch.

This separation is what makes adding a new protocol an orchestration problem rather than a reimplementation one, as a new protocol is only required to pick primitives off this library instead.

5.3.3 Data flow of one simulation run

A simulation run goes through the modules described in the previous subsection in a fixed order. First, the entry point, `src/main.rs` provides an interactive use, `src/bin/eval.rs` is for the autonomous agent. The engine parses its configuration and builds the `SimulationConfig` that sets the protocol, topology, workload parameters, the seeds and the round cap. The configuration is then handed to `Simulation::new`, that builds the per-replica sets via `Workload::generate` 5.3.2.1, constructs the communication graph (5.3.2.2), pins each set to a `Replica` 5.3.2.3 and finally instantiates the `Network` from the given topology 5.3.2.6. From this point on, the configuration is set and no other module will read it.

The engine then enters the simulation loop with `Simulation::run`. Each round begins with `Network::reset` for zeroing the per-round counters while still leaving the cumulative `Replica.stats` untouched. The engine iterates over every replica and calls `Protocol::send_phase` with a freshly built `ReplicaView`, a reference to the `Topology`, an `Outbox` bound to that replica's identifier, and the replica's own carry from the previous round.

Messages emitted through the outbox are billed for their state bytes at send time and dropped into the recipient's inbox. Once every replica has sent, the engine drains all inboxes and calls `Protocol::recv_phase`, collecting each replica's `ProtocolStepResult` that encapsulates the new local set, the encode and decode time and an optional carry. The bytes sent are then billed by `Network::bill_metadata_post_recv` against the state of every inbox message after being read by the recipient so any growth in rateless sketches is captured accurately 5.3.2.6. After each round's receiving phase, the engine records each replica's byte counters and timings into `stats`, asserts the monotonic-union invariant, updates the replica's set and stores the returned carry back into the slot reserved for that replica's next `send_phase`.

At the end of each round the engine checks whether every replica's set equals the workload's `target_union`. If this is the case, the simulation loop terminates with `RoundOutcome::Converged`. There is another possible scenario where the simulation loop terminates and that is when the round count reaches the established `round_cap`, at which point the engine returns with a `RoundOutcome::RoundCapReached`, and the run is marked as unconverged.

Once a run has finished, the engine calls `MetricsSnapshot::from_replicas` 5.3.2.7 to aggregate every `Replica.stats` into a per-node and total view inside a `SimulationResult` struct that can be used by the exporting module and holds the round count, the convergence flag, the topology and protocol identifiers besides the metrics snapshot. Depending on the binary used the way this output is printed can vary. The diagnostic binary prints a human-readable table and appends a `RunSummaryRow` to `results.csv` via `export.rs`, whereas the agent-facing binary serialises the same `SimulationResult` to JSON on stdout and writes nothing to disk.

Most of this pipeline is fixed except for the `Protocol` instance the engine drives. All input parameters, such as the workload, topology, seeds, billing layer, converge check and results aggregation are identical across every run. This is what makes a fitness function change in between two runs be attributed to the protocol rather to changes in the underlying simulation framework, and what makes the autonomous research agent loop 5.12 a defensible methodology.

5.4 Billing model

The bandwidth comparison this thesis is based on is only meaningful if every protocol is charged equally for every byte it would have moved on a real network. The billing model is the layer responsible for accounting for all the bytes each protocol sends, without mistakes, in order to ensure the main function of the simulator, specially for automatically generated protocols.

This section's goal is to describe the invariant this model was built on, that no data can cross between replicas and that no protocol can write to a byte counter directly. Subsection 5.4.1 defines what counts as a billable byte and explains the reasoning behind the fitness function. Subsection 5.4.2 describes and justifies the design of the possible ways the byte counters can be incremented, as well as the structural mechanisms including the opaque `PrtocolMsg`, sealed wrappers, senders' `Outbox` and neighbor assertion that were used to close off every other alternative.

Finally, subsection 5.4.3 addresses the exception where per-replica state needs to be kept in between rounds, and the reason it does not violate the invariant.

5.4.1 Wire bytes

A wire byte is any byte that would travel across the network in a real deployment of the protocol. The simulator keeps track of two different types of bytes: state and metadata. State bytes correspond to the serialized payload of set elements actually being transferred between replicas. In the simulator implementation this is calculated as the sum of all of a replica's `Element::wire_size` carried in a message plus the associated opaque payload of `payload_size` bytes. Metadata bytes, on the other hand, represent everything else that may be transmitted on the wire, including BFs, bit-arrays, RIBLT coded symbols and their checksums. In short, they are any information passed through the network the protocols use to find what the symmetric difference is.

The fitness function (5.9) is computed over the total bytes a run transmits. The split exists because of a fundamental difference between reconciliation in pairwise and multiparty scenarios. In a two-replica environment during a reconciliation run, given a fixed workload, every protocol must eventually transmit the same elements, which is the symmetric difference. State bytes therefore reduce to a workload dependent constant that would inflate every protocol's metrics by the same amount. Metadata bytes in contrast, vary with the strategy a protocol uses to discover that difference. Keeping track of state bytes is also useful for the `Full State Transfer` protocol that due to not sending any metadata bytes make the state byte counter the only valid signal for it. In this scenario, another reason for keeping separate counters is the timing of the accounting of the bytes. A sketch's metadata cost for rateless implementations is uncertain until the receiver signals the difference has already been decoded. Subsection 5.4.2 describes how metadata is charged on the receiver side when its true size becomes observable, rather than at send time when it would still be an estimate.

In an environment where multiple replicas have to reconcile, this assumption no longer holds. Now, an element missing from several replicas must traverse multiple nodes to reach all of them, this means the amount of times it is transmitted depends on the dissemination strategy and topology demand. The amount of state bytes a protocol moves stops being a constant, making them a genuine axis of efficiency that separates protocols and a fitness computed over metadata only would reward a kind of redundant payload propagation, without ever raising the fitness function value. By scoring over the total bytes, both the cost of discovering the difference and the cost of disseminating it across the topology are accounted for. An objective lower bound for the symmetric difference payload volume as each replica must receive each element that it lacks at least once, but in the multiparty case it depends on the topology and therefore is far less trivial to compute.

5.4.2 A closed billing surface

Defining what a wire byte is (5.4.1) is only useful if a protocol cannot bypass the counters and record whatever value it wants. This subsection describes the mechanism that achieves this. It

includes a single trait that exposes the message size, two and only two controlled write paths, and a small set of structural restrictions on the simulator Application Programming Interface (API) that leave no gaps for a replica to move data between them without being charged.

As discussed before every message is required to have a wire size that represents the cost in bytes it would take to send that message over a real network. The compiler enforces this by requiring every message type to implement the `WireSized` trait, that exposes a message's cost as a method for each type of bytes. Throughout the entire simulation the trait is only invoked by the `Network` module and protocols not only never read it but also have no way to write to it. This has as a direct effect that the size a protocol claims its messages have is irrelevant, only the values the network layer has is considered the true.

Another significant measure to avoid wrong accounting of bytes is the fact that the byte counters in `ReplicaStats` are private to the simulator crate and the only two functions that write to them are `Network::send` and `Network::bill_metadata_post_recv(replica)`. The first function bills the `msg.state_bytes()` to the sender at the moment the message is enqueued in the receiver's inbox. State bytes are known at send time and therefore charging them here is exact. The second function is called by the engine, once per round, only after the receiving phase has returned. It iterates over the messages the replica consumed during that phase and bills each message's `metadata_bytes()` to its sender. This is the mechanism that handles the billing of sketches whose size is only fixed once decoding finishes, avoid having to account for an estimate instead.

Despite these two measures closing down some ways a protocol could cheat its final fitness function result, a trait and minimal write paths are not enough on their own, as a protocol could still fabricate a zero-cost message type, send a message to a non-neighbor replica or handing state to another replica through some shared object or reference that would effectively bypass the billing layer. Three structural restrictions were implemented to prevent this practices: Opaque ProtocolMsg, Outbox bound to senders and narrowed views.

A Protocol with the ability of creating its own ProtocolMsg is a protocol that can define its own message's size. This can become a problem when an Artificial Intelligence (AI) agent has as a sole goal to reduce the transmitted metadata. To avoid this, all message types carried by the network are a public wrapper around a private inner enum. This setup prevents Protocols from constructing `ProtocolMsg` directly and forces them to go through already existent sealed wrappers like, `RibltMsg`, `BloomMsg` and `RatelessBloomMsg`, each of which already has a tested `WireSized` implementation.

In order to fix the potential issue of having replica's sending messages to other nodes not in their neighbors list, an `Outbox` is bound to every sender via `Outbox::for_replica` and holds both a mutable reference to the network and the sender id. This way, a protocol cannot impersonate another replica because the `from` field is set once, by the engine, before the protocol ever sees the outbox. `Network::send` additionally asserts that the `(from, to)` pair is an edge in the current `Topology`, so a protocol cannot reach a non-neighbor by accident.

In order to prevent wrongful changes to a replica's stats and/or phase, a protocol send phase

never receives a full reference to a `Replica`, instead it receives a `ReplicaView` created by the engine before every send phase that only exposes the replica's id and an immutable reference to its set, leaving `ReplicaStats` and `ReplicaPhase` out of scope. These changes leave therefore no in-memory object shared between replicas that the protocol could use as a side channel.

All together these restrictions reduce the exposed surface across data can move between replicas to a single function call (`Outbox::send`, with a single billing point on the sender side (`Network::send` and a single billing point on the receiver side (`Network::bill_metadata_post_recv`). Summing up, the `WireSized` trait fixes the units while the API closes every other route.

5.4.3 Engine-routed carry state

The mechanism described in 5.4.2 closes every path through which data can move between replicas without being billed. However, it also closes every path through which a protocol could carry state from one round to the next. The `ReplicaView` a protocol receives is borrowed for the duration of a single phase, and the `Outbox` is dropped at the end of the send phase, leaving no object owned by the protocol that survives across rounds. Even though this default for the billing invariant is valuable, some reconciliation protocols are stateful. Take a rateless decoder, for example, it needs to keep a decoding table between rounds. The simulator must therefore expose a way for replicas to hold their state without reopening the side channels the previous subsection closed.

The mechanism is a carry value that the engine, crucially not the protocol, holds between rounds. `recv_phase` returns a `ProtocolStepResult`, whose `carry` field has type `Option<PendingElements>`, where `PendingElements` is a `HashMap<usize, Vec<Element>>` keyed by neighbor replica id. Its functionality is deliberately narrow. A carry is nothing more than a record of elements the replica intends to forward to specific neighbors in a subsequent round. The engine takes ownership of the returned carry, stores it in a per-replica slot (`carry_states[i]`), and passes it back into the same replica's `send_phase` on the next round via the carry argument.

It is important to note that the carry never crosses replicas and that it only contains information that will be later billed during the following send phase. The engine stores each replica's carry in its own slot and only hands it back to that same replica's next `send_phase`. In the meantime, no protocol can ever observe another replica's carry. The `PendingElements` a carry contains consist only of `Element` values keyed by neighbor id. The following `send_phase` can only act on these values by placing them on the wire through the valid interfaces, at which point `Network::send` bills their state bytes as it would for any other transmission.

The current design replaces an earlier mechanism in which cross-round hints were stored in a shared object visible to every replica. That arrangement made hint propagation indistinguishable from message passing on an unbilled path. By replacing the shared store with a per-replica slot routed by the engine preserves the multi-round capability that real protocols need while leaving the invariant that the only way data moves between replicas is through the network layer intact.

5.5 Protocol isolation

A bandwidth measurement is not only invalidated by an under-reported byte; it is also invalidated by an over-informed protocol. If the code being measured can read information that a real deployment could not have observed — the contents of a peer's set, the union the replicas are converging towards, the parameters of the workload that generated their initial sets — then the byte count it produces is a measurement of a different problem than the one a real network solves. In the autonomous-agent setting this concern is sharper than in a hand-written comparison: the protocol is being rewritten by a process whose objective is to make a number go down, and any in-scope piece of state that correlates with that number is a shortcut the agent will eventually find.

Section 5.4 closed the channels through which a protocol could move data between replicas without being charged. This section closes the channels through which a protocol could *read* information it would not have in deployment. The invariant is the dual of the one stated there: *a protocol's decisions in a given round are a function only of its own replica's set, the topology adjacency it sees, the messages it has received through the network, and the carry the engine handed it from the previous round — nothing else*. Subsection 5.5.1 reframes the `Protocol` trait as the contract that fixes those inputs and forbids protocol-private memory across rounds. Subsection 5.3.2.3 enumerates the information a real node would have access to and confirms that none of it is in scope: peer sets, the target union, workload parameters, peer phases and statistics, and the contents of any topology node other than the protocol's own. Subsection 5.5.2 addresses the one isolation failure the type system cannot prevent — a protocol that fakes convergence by discarding elements rather than reconciling them — and describes the runtime assertion that catches it.

5.5.1 The `Protocol` trait

The `Protocol` trait exposes two methods with the following signatures.

```
fn send_phase(
    &self,
    local: ReplicaView<'_>,
    topology: &Topology,
    outbox: &mut Outbox<'_>,
    carry: Option<PendingElements>,
);

fn recv_phase(
    &self,
    local: ReplicaView<'_>,
    topology: &Topology,
    inbox: Vec<(usize, ProtocolMsg)>,
) -> ProtocolStepResult;
```

This trait can be looked at as a list of inputs a reconciliation protocol is allowed to consult, and it has three properties that fix what a protocol can know.

First, the receiver is never a mutable reference. An implementation of a Protocol is not a state machine it cannot store decoding tables, nor message histories. Anything that a protocol must remember must pass through the carry described in 5.4.3, whose shape is set by the engine and whose contents are restricted to pending elements. Second, the inputs are exhaustive. A protocol receives a `ReplicaView` of its own replica, an immutable reference to the `Topology`, the inbox of messages it has been delivered or an `Outbox` to send them through, in the send and receive phase respectively. Nothing else is in the protocols' scope and this includes the workload, simulation configuration as well as metrics snapshots. The trait is the entire surface a protocol has to get information from the reconciliation process. Third, similarly to the process of receiving information, the output is also restricted in the same way. `recv_phase` returns a `ProtocolStepResult` carrying the next set, the `LocalMetrics` for encoding and decoding time and the carry. There is no out-parameter through which a protocol can write to peer state and no shared object it can mutate. The send side is even narrower with `send_phase` having as only option to send messages its own `Outbox` assigned by the engine and billed by the network as described in 5.4.2. A protocol's effect on the rest of simulation is the messages it places on the network, the next set it returns and the carry its own future can use.

The `ProtocolKind` discriminant returned by the trait's `kind` method is a separate concern. It is only read by the engine and by the export layer to label results in `results.csv`.

5.5.2 The monotonic-union invariant

The structural restrictions of the protocol trait (5.5.1) leave one shortcut the type system cannot prevent. In this conditions a protocol could still return a `next_set` that drops elements the replica holds. This would produce an empty set that would then equal every other empty set and trigger a convergence without having reconciled anything.

Even though in this simulator, convergence is defined as every replica holding the target union defined beforehand by the workload, the engine still forbids this with an assertion. After each `recv_phase` the engine compares the returned `next_set` against the replica's set from the previous round and ensures the new set is a superset of the old one. This is the monotonic union invariant, that directly correlate the convergence check with a notion of progress, as a replica that is reported converged has achieved the target union by additions and never by discarding elements.

5.6 Topology model

The bandwidth cost of a protocol depends on the workload it is put up against and on the graph over which replicas communicate. Some strategies may excel with a central coordinator when others may prefer a tree like graph. To prevent a protocol from overfitting a determinate shape, the evaluation runs every protocol against three distinct topologies specifically chosen to disagree on three structural parameters that directly influence the reconciliation process: diameter, maximum

degree and regularity. The three topologies are Start, balanced binary tree and Chord with offsets of powers of two. They are all defined in 5.6.1.

5.6.1 Definitions

As mentioned before, every topology is constructed by `Topology::build` in `src/simulator/topology.rs` from the kind and the node count n provided by the `SimulationConfig`. For simplicity, all graphs are undirected, simple and connected for $n \geq 1$. Figure 5.2 illustrates the three constructions for $n = 8$.

5.6.1.1 Star

In this topology there is a central hub designated as the hub. The edge set is $\{\{0,i\} \mid 1 \leq i < n\}$. The graph has $n-1$ edges and diameter $n \geq 3$. The hub has degree $n-1$ therefore presenting a skewed distribution of the degree.

5.6.1.2 Balanced binary tree

For each node $i \geq 1$ the parent is $\lfloor (i-1)/2 \rfloor$, producing the standard binary tree rooted at node 0. Similarly to the Star case, the graph has $n-1$ edges however the diameter can be calculated with the following formula $2\lceil \log_2 n \rceil$. In this topology the degree of every node has a degree upper bound independent of n, the root has degree at most 2 whereas internal nodes have degree 3 (parent and two children) and leaves have degree 1.

5.6.1.3 Chord

For every node i and every k with $2^k < n$, an undirected edge is added between i and $(i+2^k) \mod n$. This makes it so each node has at most $2\lfloor \log_2 n \rfloor$ neighbors, giving $\Theta(n \log n)$ edges in total and diameter $\Theta(\log n)$.

The three graphs therefore disagree on every axis the simulator can measure, creating different circumstances where sometimes there is a centralized configuration and others there are more communication paths with different distances.

5.6.2 Why these three topologies

This subsection argues for the choosing of these three distinct topologies and what distinct failure mode for a reconciliation protocol they can induce.

5.6.2.1 Star

The main property this topology shows is the fact that every message between two non-hub replicas must traverse node 0, and that the hub's degree grows linearly with n. A protocol that scales its per-edge metadata with neighbor count, or that implicitly assumes the hub can aggregate on behalf

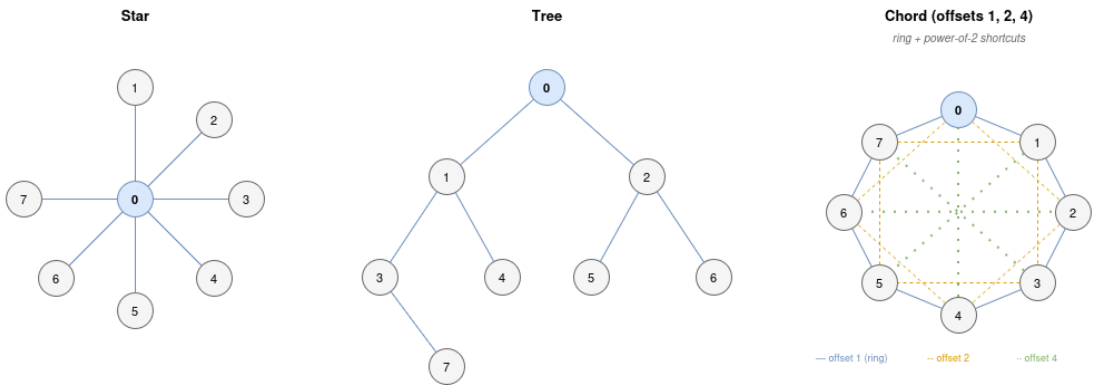

Figure 5.2: Tree, Star and Chord topologies with 8 nodes

of the rest, will pay a cost here that is invisible on any regular graph. On the other hand, a protocol that leverages the hub as a coordinator, by having it broadcasting and collecting once, will benefit greatly. Making it look very promising unless evaluated against other topologies where no such vertex exists.

5.6.2.2 Balanced tree

The tree has no vertex with an extraordinarily high degree but it does have a unique path between any two replicas that distinguishes it from the other two topologies. This path can be as long as $2\lceil \log_2 n \rceil$, which means, information cannot travel from one leaf to another in fewer rounds than the diameter, regardless of message size or complexity. The tree therefore exposes protocols whose per-round metadata is small but whose round count grows with depth.

5.6.2.3 Chord

In a chord topology , every node has the same neighborhood structure, this means, there is no hub to exploit and no privileged root to route through. Its $\Theta(\log n)$ diameter rules out the tree's depth penalty, and its degree no larger than $\log n$ rules out the star's hub cost. A protocol that wins on Chord cannot be doing so by exploiting either particularity making the gain attributable to the reconciliation algorithm itself.

A further reason to include Chord is that, unlike the Star and the Balanced Tree, it is not acyclic. The ring of successor edges together with the power of two links create more than one path between most pairs of replicas, so the graph contains cycles. This is the only one of the three topologies where such redundancy of routes exists. For instance, in the Star every path crosses the hub, and in the tree case, there is an unique path between any two replicas (Subsection 5.6.2.2). This property matters because cycles allow for assessing the quality of the protocol's handling of redundant dissemination. Where a unique path guarantees that each element reaches a replica from a single direction, multiple paths let the same payload arrive from several neighbors, and a protocol that does not track what it has already sent or received will re-transmit elements around the cycles and inflate its state bytes Subsection 5.9.1).

5.6.2.4 Coverage of the structural parameter space

Taken together the three graphs cover a wide range of the structural parameters the simulator can vary for a fixed n. These provide a skewed, bounded and uniform degree distribution, a constant and logarithmic diameter and some versions of symmetry. A protocol that improves the fitness number on all three has improved the underlying reconciliation cost whereas one that improves on a single one has found a structural shortcut, which despite insightful is not the objective of this thesis. The evaluation matrix (5.8) is designed to make that visible. This property of the results of the simulations made is particularly important for the autonomous agent, which again, has as its only purpose to reduce the value of the fitness number and without this topology spread it could very well converge on a protocol that exploits one graph's configuration at the cost of the others'.

5.7 Workload generation

As described in 5.3.2.1, the workload generator in `src/simulator/workload.rs` produces the per-replica initial sets that feed every run, this includes drawing elements from a universe according to the configured distribution and arranging the resulting sets so that the pairwise Jaccard similarity matches a target value. From this set of information, only the replica's set themselves are ever seen by the protocols, the target similarity is only used by the workload generator and again by the engine for the convergence check (5.9.3).

The rest of this chapter describes the tools used to ensure the three properties that make this generator deterministic and compliant with the reproducibility constraint defined in 5.1.1. First, the generator is parameterized by a Jaccard similarity rather than by an absolute number of differing elements, so a single configuration scales sensibly with the total set sizes (5.7.1. Second, all elements are drawn from a Zipf-distributed universe rather than sampled uniformly. The motivation is not that skew penalizes sketch internals. The motivation is that an uniform universe produces an unrealistic symmetric difference, since in real reconciliation workloads, like mempools and replicated stores, popular elements are disproportionately shared and rare elements are very unique, so both the size and the composition of the set difference are correlated with the skew. Since every reconciliation cost in this evaluation scales with the symmetric difference, the input distribution has to be heavy headed to keep the workload representative (5.7.3). Third, the generator supports two different divergence patterns, uniform and clustered, that differ on whether the similarity structure between replicas is flat or has as a hub-like structure (5.7.4), both are needed for the same reason three topologies are (5.6) and because they help mimic a real deployment environment.

5.7.1 Jaccard parameterization

The generator is parameterized by the target pairwise Jaccard similarity $J \in [0,1]$ rather than by an absolute count of differing elements. For two sets A, B of equal size n sharing c elements, $|A \cup B| = 2n - c$ and

$$J = \frac{|A \cap B|}{|A \cup B|} = \frac{c}{2n - c},$$

which inverts to

$$c = \frac{2nJ}{1+J}.$$

This is the formula implemented by `jaccard_to_common_size` in `workload.rs`. The module has a unit test `jaccard_to_common_size_round_trips` that checks that it recovers J to within 10^{-3} across the operating range. Each replica then receives a set with $n-c$ unique elements plus the c common elements which makes the pairwise Jaccard match J within 0.05 in practice.

Jaccard is the natural axis for this evaluation, but it is worth flagging that it is not the conventional one. Most reconciliation literature like CPISync [10], the IBLT protocols of Eppstein et al.[4] and Goodrich and Mitzenmacher [8], rateless IBLT [15], Graphene [13], Minisketch parameterizes evaluation by the symmetric difference size $d = |A \triangle B|$, because each protocol's bandwidth bound is stated directly in d. This simulator uses J instead for two reasons. First, J moves independently from the scale, in other words, a workload configured at $J = 0.9$ has the same similarity structure at $n = 10^3$ and at $n = 10^5$, whereas d must be restated for every set size. Second, the multi-party setting studied here has no single symmetric difference as the per pair, d_{ij} varies, and a per-pair Jaccard target is a cleaner way to specify a workload than a matrix of d_{ij} values. The two metrics are interchangeable for equally sized sets through the formula $d = 2n(1-J)/(1+J)$, so results can be re-expressed in the literature's convention when comparing against a specific single-pair benchmark.

5.7.2 Universe and element construction

The universe is a fixed pool of elements with size equal to `universe_size`. These are built once per run by `build_universe`. Each element is constructed deterministically from its integer index $i \in \{0, \ldots, \texttt{universe\_size} - 1\}$ and carries two fields: a digest, produced by hashing i and masking the result to the low `digest_bits` (`masked_digest` in `workload.rs`) and a payload which essentially is a buffer of size equal to `payload_size` filled from a Pseudo-Random Number Generator (PRNG) seeded by i under a fix domain separator (`deterministic_payload`). Both fields are pure functions of i ,so element i is identical across runs, seeds and replicas. The workload seed controls only which elements are sampled into each replica's set and does not change any element's content. This setup is what allows the construction in 5.7.1 to give the same element on different replicas. The two parameters control the two cost functions every protocol sees. `payload_size` sets the cost in state bytes of transmitting that element while `digest_bits` sets the collision rate in the hash space some sketches rely on. At `digest_bits` $= 64$ the digest is effectively unique. Narrowing this parameter makes the simulator admit the collisions. Since the digest is a deterministic function of i , the collision structure is reproducible across seeds, which makes two indices that collide in one run collide in every run at the same `digest_bits`. The payload size is fixed at 32 bytes for every element in every run and it is not varied across the experiment matrix. Payload size is a parameter bound to the application the protocol is being used on and not a property of the reconciliation protocol itself. As mentioned above it enters only in the

cost as state bytes and is billed by every protocol identically. Therefore holding it constant isolates the protocol comparison from a dimension that scales all protocols' state cost by the same factor and is therefore an invariant to the final ranking. 32 B was chosen as a representative workload of small records.

5.7.3 Zipf sampling

Sampling from the universe is weighted rather than uniform. `build_zipf_sampler` assigns weight $1/r^s$ to the element of rank $r \in 1\ldots,$`universe_size`, where $s =$ `zipf_exponent` is a configured constant in the range $0.8\ldots1.4$. From there a `WeightedIndex` then draws indices in proportion to those weights. It is important to note both calls in the workload generator to a the zipf sampler use the same sampler.

An uniform sampling would be a simpler solution and is what most of the sketching literature implicitly uses, as most experiments are parameterized by the symmetric difference size d and the underlying elements are either drawn as uniform random integers [4] or treated as opaque hashes whose distribution is left unspecified [8][15][13]. The justification behind choosing a zipf sampling is directly related to the fact that the symmetric difference in real reconciliation workloads popular elements tend to be shared between replicas at a disproportionally larger rates than the unique ones. Both the size and the composition of the per pair symmetric difference are correlated with this skew. Since every reconciliation cost in this evaluation scales with the symmetric difference, the input distribution has to be representative of the scenarios the protocols are to be deployed in.

5.7.4 Divergence patterns

The generator supports two divergence patterns, selected by the `DivergencePattern` field of `WorkloadConfig`, that differ in how the pairwise similarity is distributed across replicas.

5.7.4.1 Uniform

In the uniform pattern (`generate_uniform`) a single common set of size $c = 2nJ/(1+J)$ is sampled once from the universe. Each of the r replicas then takes the common set and adds $n-c$ locally unique elements drawn from the universe with the common set excluded, as described in the beginning of the section. This is the pattern used whenever the evaluation is sweeping along the J axis and wants the similarity structure between replicas to be held constant, as the pairwise Jaccard similarity is approximately J for any given pair of replicas.

5.7.4.2 Clustered

The clustered pattern (`generate_clustered`) instead produces a different similarity matrix where replicas inside the same cluster are more similar to each other than replicas in different clusters. It is parameterized by three values: `cluster_count`, `jaccard_inter` and `jaccard_intra`.

The construction of the sets begins with setting a global common set of size $c_{\text{inter}} = 2nJ_{\text{inter}}/(1+J_{\text{inter}})$ which is shared by every replicas regardless of cluster. On the second step, a cluster shared set of size $c_{\text{intra}} - c_{\textbf{inter}}$ is sampled on top of the global common, where $c_{\text{intra}} = 2nJ_{\text{intra}}/(1+J_{\text{intra}})$ is the total target overlap between two replicas in the same cluster. Replicas are assigned to clusters using a round robin approach by index. Finally, every replica adds $n - c_{intra}$ locally unique elements on top of its cluster base.

Following this design a pair of replicas in the same cluster effectively shares c_{intra} elements which is the equivalent to the global common plus the cluster shared layer while a pair in different clusters shares only c_{inter} elements.

An extra constraint is asserted by `validate_config`: $J_{\text{intra}} \geq J_{\text{inter}}$. This is what makes pairs in the same cluster be at least as similar as cross-cluster pairs that is what should happen by definition. If the inequality was reversed the construction would still run but the resulting similarity matrix would no longer mimic real clusters.

5.8 Evaluation matrix

Having defined all existent topologies and seeds and how the jaccard indexes and total set sizes can influence the final result, it is important to define an evaluation matrix that allows any protocol to be tested against a fixed set of configurations. The matrix is the unit of comparison for a protocol's performance as it summarizes and aggregates the final value of the fitness function across all cells.

The choice of which parameters to sweep and which values to use is a delicate one. Too few axes and a protocol can overfit to a single configuration. Too many and the amount of runs and therefore the time it takes for the agent to test its ideas skyrockets. The next subsections specify which parameters are swept and which are held constant throughout every execution (5.8.1) and how a cell is defined and summarized (5.8.2).

5.8.1 Sweep dimensions

The orchestrator in `bin/eval.rs` iterates over four lists declared in the TOML evaluation configuration including `topologies`, `jaccard_similarities`, `num_replicas` and `seeds`. The matrix the agent used to compare protocols is as follows.

- topology $\in \{\text{Star}, \text{Tree}, \text{Chord}\}$,
- $J \in \{0.25, 0.5, 0.75\}$,
- $n \in \{16, 64\}$,
- three simulation seeds,

Every combination of these four produces one simulation run, which makes the total amount of runs equal to the product of their lengths, which means there were a total of $3 \times 3 \times 2 \times 3 = 54$ runs per evaluation. The topology sweep is justified in 5.6, the jaccard similarities in 5.7.1 and

the replica count is included to access how each protocol's efficiency scales with n. Every other parameters is held constant across the matrix including the entirety of the workload configuration except for the number of nodes.

5.8.2 Cell definition

A cell is a triple $(\text{topology}, J, n)$, one of the points in the matrix once the seed is collapsed. `build_cells` in `bin/eval.rs` is responsible for grouping this triple and emits a `CellSummary` per cell containing four aggregates. First `mean_bytes` represents the arithmetic mean of `total_bytes_sent` across the cell's seeds, where `total_bytes_sent` is the sum of state and metadata bytes recorded by the network in 5.4. Secondly, it shows `std_bytes`, which represents the standard deviation of the runs for the same number of replicas n. It also keeps track of the average amount of rounds. The operation used is the arithmetic mean and is shown as `mean_rounds`: Finally, it presents a flag,`all_converged`, only set to true in case all seeds converge within the round cap.

Importantly, the aggregation only happens across seeds. The other three axes the matrix sweeps, the topology, jaccard similarities and number of nodes show structural properties of the protocol being tested and hiding them would compromise the objective of the matrix 5.6, 5.7.1. Aggregation by seeds, on the other hand, has as sole purpose to average out internal randomness in the protocols running in a fixed workload.

The standard deviation (`std_bytes`) is used to discard small differences between protocol candidates. A change in the fitness number (5.9) smaller than the standard deviation of the cells it aggregates over is not treated as a real improvement by the autonomous agent.

5.9 Fitness function

The evaluation matrix (5.8) produces a structured result containing the byte count metrics for every run. The autonomous agent that rewrites the protocol (5.12) has access to the full breakdown of each run and uses the per-cell information to distinguish improvement from noise. However to ensure a fair comparison between different algorithms and to better assess whether or not a protocol should be kept or discarded.

The fitness function does exactly that, reduce every component to a single value that represents the cost of each protocol where a lower value means more efficiency.

The entire evaluation harness is dependent on this one value making it one of the most important design decisions, as it is the sole optimization target for the agent. A protocol is accepted only if it reduces the scalar while still keeping the convergence flag up (5.9.3). This means the agent will pursue every option available to lower that value and therefore must be carefully chosen. The following subsections justify the formula (5.9.1), the choice of a geometric mean over a a sum or an arithmetic mean (5.9.2) and finally the convergence guard (5.9.3) that closes down the easiest path of reducing a lower amount of bytes simply by not reconciling every replica.

5.9.1 Definition

The fitness of a protocol is the geometric mean of the per-cell mean byte counts, as described in the following formula.

$$\text{fitness} \;=\; \left(\prod_{c\in\mathscr{C}} \overline{b}_c\right)^{1/|\mathscr{C}|} \;=\; \exp\!\left(\frac{1}{|\mathscr{C}|}\sum_{c\in\mathscr{C}} \ln \overline{b}_c\right),$$

where $\mathscr{C}$ is the set of $|\mathscr{C}| = 3\times 3\times 2 = 18$ cells of the matrix (5.8) and $\overline{b}_c$ is the cell's `mean_bytes` (5.8.2).

The geometric mean was computed in the log space form $\exp\!\left(\frac{1}{|\mathscr{C}|}\sum_c \ln \overline{b}_c\right)$ rather than by calculating the product $\prod_c \overline{b}_c$ directly, because it would quickly overflow with the results this matrix is due to output assuming each $\overline{b}_c$ is on the order of 10^6 to 10^9 bytes in the agent's configuration. Furthermore, to ensure the function is strictly positive two asserts were put in place, one to ensure every cell's value is larger than 0 ($\overline{b}_c > 0$) and the other to ensure the matrix is not empty. By failing loudly on failure, it is guaranteed no wrongfully generated cell can contribute 0 to the product and therefore collapse the entire fitness function value to 0.

Each $\overline{b}_c$ is the mean of `total_bytes_sent` across the cell's seeds. This includes state bytes as well as metadata ones (5.4). As mentioned in subsection 5.1.2 every replica must receive the payload of each element it is missing at least once, creating a constant lower bound for every cell, that does does not dilute any protocol's efficiency, but instead flags those that overshoot that floor and therefore should be considered inefficient.

5.9.2 Why geometric mean

The matrix aggregates the results by geometric mean rather than by a sum or arithmetic mean of the byte counts due to the difference in the magnitude of the results. While for the cases of Tree or Chord at $n = 64$ protocols can move several gigabytes, at smaller values of they will only move a few megabytes. If the chosen solution was to use sum, the cells with the highest cells would dominate the result and therefore make the agent optimize for them instead of optimizing for every possible scenario, completely undermining the evaluation matrix efforts to avoid overfitting (5.8).

Moreover, the geometric mean removes the dependence on the absolute magnitude. Because $\ln(r\cdot b) - \ln(b) = \ln r$ for any baseline b, meaning a relative change in any cell affects the score by the same amount regardless of its size. Take a baseline with two cells with a large cell at 1,000,000 bytes and a small cell at 1,000 bytes, and two candidate edits, each a 10% reduction of one cell.

Using a sum, edit X scores 901,000 and edit Y scores 1,000,900, making the agent choose edit X by nearly 100,000. Using the geometric mean, the final fitness function for both is the same as they both represent the same relative gain $\sqrt{900{,}000\cdot 1{,}000} = \sqrt{1{,}000{,}000\cdot 900}\% = 30{,}000 =$ 10%. This effectively reduces the artificial advantage protocols who that favored larger cells had by comparing each improvement with its baseline.

This function also excels at preventing the agent from accepting protocols that provide small improvements in all cells except for a select few if those few suffer catastrophic regressions, as a relative blow-up in any single cell would deeply impact the total score.

Finally this function was chosen as well due to its semantics. Using the geometric mean, a $X\%$ cheaper protocol in every cell will have display a $X\%$ lower overall fitness function. Making it the optimal choice to keep the optimization target in line with the breadth the matrix is meant to measure.

	large cell	small cell
baseline	1,000,000	1,000
edit X (-10% large)	900,000	1,000
edit Y (-10% small)	1,000,000	900

5.9.3 Convergence

The fitness number measures the bytes transmitted by all replicas until convergence is achieved. Since the goal is to lower this number, a protocol that does not transmit anything over the network, despite not learning any new information and failing at the task it was created for, it would effectively score better than any other.

Convergence is therefore checked independently of the byte count and treated as a requirement and not as a contribution to the final fitness function value. A run is only considered as converged when every replica's set is equal to the target union (5.9.3). The engine is responsible for confirming convergence once a run has finished and stores that information in a flag passed downstream. Each cell carries an `all_converged` flag that is true only if all of its seeds converged. The matrix then has a variable of its own (`summary.all_converged`) that is only true if every cell has converged, which then the autonomous agent in its loop is required to check before a candidate is evaluated. In cases where `all_converged` is not true the protocol is immediately discarded and the changes reverted before being documented in the hints.md file.

Together with the strictly positive guard of 5.9.1, the convergence precondition closes the way a protocol could lower its fitness without doing real reconciliation work: transmitting less information than the reconciliation requires, by stopping short of the target union or in the limit by sending no information at all.

5.10 Baseline

The autonomous agent optimizes a single scalar (5.9) but a byte count by itself is not meaningful until a good comparison metric taken with the same evaluation harness is provided. The baseline is the ceiling every protocol must remain under given a certain workload and provides that reference. This experience uses a single baseline and keeps the sketches and filter algorithms described in section 5.3.2.8 as a shared toolkit the agent can use to create new protocols.

The chosen baseline was the FST protocol. It is both trivially correct as every replica sends its entire information to every other replica and reconciles with each one pairwise, while also representing the upper bound in transmitted bytes to the redundant transmission of information.

When it comes to the way it fits in this evaluation matrix it is also the simplest implementation of the `Protocol` trait. During `send_phase` each replica only collects its local set and sends its payload to every neighbor, while on `recv_phase` it inserts each received element into its local set. It only transmits state bytes, meaning no sketch nor metadata is produced (5.4).

These characteristics make it the natural starting state. `multi_replica.rs`, the only file the autonomous agent is allowed to edit (5.12), begins as a copy of it so the agent's opening fitness is exactly the baseline ceiling at the start. Every edit the agent keeps is then a reduction against that line, and the gap between the final protocol and this baseline is exactly the outcome this experiment aims at producing.

5.11 Reproducibility

Reproducibility is the property this entire experiment depends on. Every protocol the autonomous agent tests is kept or discarded by comparing only the fitness number against the previous (5.9), and that comparison can only be considered meaningful if the number is a function of the protocol rather than of the run. Therefore, the simulator was built for this scenario. A run's setup is completely determined by its configuration file (`agent.toml`). This file fixes all the sources of randomness in the execution guaranteeing the same per-cell byte count for the same configuration. This ensures the agent, therefore, reads a difference between two related fitness numbers as a difference between protocols and not as noise from the harness. It further allows for the complete replay of any run with the same results from, given it starts from the same configuration file effectively fulfilling the design goal of 5.1.1.

5.11.1 Single input source

The evaluation binary (5.3.1) has no hard-coded parameters. Every value used to define a run, including protocol, topologies, replica counts, divergence levels, seeds, round cap and the full workload description is read from a single TOML file, the source of truth composed of two distinct parts. The top block names the protocol under test and the swept axes (`topologies`, `seeds`, `jaccard_similarities`, `num_replicas`, `round_cap`). Whereas the bottom one handles the workload configuration by fixing the set generation with parameters like, `set_size`, `payload_size`, `digest_bits`, `pattern`, `universe_size`, `zipf_exponent`, and the workload `seed`.

Nothing outside this file influences a measured byte count. From it the binary expands the Cartesian product `topologies` × `seeds` × `jaccard_similarities` × `num_replicas` and runs one simulation per combination, which are exactly the $3 \times 3 \times 2 = 18$ matrix cells of 5.8, each repeated across three different seeds, for a final total of 54 simulations per test.

Since each swept axis is represented as a list in the file rather than hard-coded constants in the source, the same binary reproduces the full matrix simply by editing the configuration, which makes the agent never need to recompile the harness to change what it measures.

5.11.2 Build and invocation

The evaluation consists of a single binary (5.3.1) invoked with one argument, the configuration file.

```
cargo run --release --bin eval -- --config agent.toml
```

The binary reads the file, expands the matrix described in the previous subsection (5.11.1), runs all 54 simulations, and writes the entire result as JSON to the standard output. Release mode is not optional since the $n = 64$ cells of the baseline are quadratic in replica count and run for tens of seconds each even when compiled with optimizations. Re-measuring the baseline involves the same process with the same binary and same workload changing only the protocol field, because the command is the whole measurement and there is no second step.

Because the substitution between the FST protocol and any other touches only the protocol and leaves every workload parameter and seed in place, the comparison isolates the protocol as the sole variable, defending the property the baseline (5.10) depends on to be a fair reference.

5.11.3 Correctness gate

A fitness number is only admissible if the protocol that produced it is correct, so every measurement only goes forward after the test suite is run. The agent's loop runs the two commands together `cargo test && cargo run -release -bin eval`, so the shell aborts in case of a failing test that would spoil the signal of the agent's loop, yielding no JSON instead. To better enforce this constraint, the agent's instructions require the suite to pass after every edit (5.12)

This matters since the quantity being optimized, the bytes transmitted during the entire run, as explained in the section 5.4, can easily be minimized by a protocol that fails to converge. The test suite allows for an early detection of problems without having to run the entire evaluation matrix, all 54 runs, effectively adding another layer of protection against failing protocols on top of the monotonic union assertion (5.5.2) and the closed billing surface (5.4). Convergence itself is checked independently by the engine after each and every run, closing both routes to a cheap but wrong result.

5.11.4 The per-run record

The reproducible unit is the configuration file, but a single aggregate fitness number hides the 54 runs beneath it, so each run is recorded individually and tagged with the inputs that produced it. The binary facing the agent emits, alongside the per-cell aggregates, the total fitness scalar and the all converged flag, a `runs` array in which every entry carries its topology, Jaccard level, replica count, and seed together with its outcome. This allows to match any per-cell value through the cell it belongs to (5.9) to the individual simulations that produced it.

The diagnostic binary (5.3.1) records the same runs to CSV for archival, extraordinarily adding the workload parameters that the agent binary leaves implicit including the set and payload size, digest width, and divergence pattern.

These rows fixing the parameters being tested is what makes the simulations reproducible (5.11.1) but the records of the results is what makes them traceable.

5.12 Autonomous agent loop

5.12.1 Motivation

The target of the optimization is the protocol, which in its essence is a part of code that implements the `Protocol` trait (5.1.3) and not a vector of numeric parameters. Every candidate's variation is a structural one, which filter to use, how to size it or what to carry between them, which is a space that can be described as discrete, high-dimensional with no continuity or gradient for a classical optimizer to follow, as these commonly require a fixed parameterization of the solution. The solution is the program, and the interesting changes are edits to its control flow rather than tunings to its constants. Even though manual protocol design can make these changes, it is slow and biased toward the designer's prior intuitions. Furthermore, as a methodology, it is hard to audit and assess, requiring a lot of overhead to explain every decision. This thesis question is whether an autonomous agent can instead discover efficient protocols on its own, given a solid harness to guide it. An Large Language Model (LLM) is suitable for this as it operates in the same representation a human designer would, by reading the codebase, forming an hypothesis, writing valid code and evaluating it based on a common metric across every run (5.9). However, the agent does so tirelessly, while recording every step in a systematic fashion.

Importantly, what makes this loop viable is how the fitness signal is designed. Since each candidate's objective is to reduce the computed number, while ensuring the convergence flag is set, the search loop can run unattended. It is deliberately open ended, this is, the agent is instructed to never stop and to terminate only on human intervention, as this is an exploratory search and not a procedure that is set to find the best one.

5.12.2 Iteration protocol

The agent's loop begins from the FST protocol. `multi_replica.rs` is started as a copy of the baseline (5.10) so the agent's opening fitness value is the reference ceiling and every accepted change is an improvement from it. The agent then runs on iterations of a fixed cycle. It starts by reading its memory, which include some strategy notes ant the log of past attempts (5.12.4), and forms an untested hypothesis for a change that may reduce the fitness function. During the entire loop, the agent is only allowed to edit that single file,(5.12.3). Once the change is proven to converge successfully, the agent is to commit the edit, evaluate it and record its outcome, only then it makes the decision of either keeping the commit or undoing it. The control ten returns to the top with the repository either one step ahead with a smaller value to beat or restored to where it began.

It is important to note both the evaluation and acceptance processes are mechanical so that no crucial step relies on the agent's own judgment of its work. Evaluation is the conjoined command `cargo test && cargo run -release -bin eval` (5.11.3), whose JSON output yields two values, which are the scalar fitness and the global convergence flag (5.9). The acceptance rule reads them together and change is only kept if and only if every cell has converged and the fitness improved in relation with the current best. As mentioned in previous subsections, a run that fails to converge is rejected instantly regardless of how few bytes it sent. On the other hand, the acceptance decision passes through the version control system rather than solely relying on the agent. The edit si committed before the evaluation takes place. This means an improving change is left in place while a worsening one is undone with `git revert -no-edit HEAD`. The commit history is therefore a valid external record of the loop available for later inspection. Each attempt appears as a commit, each rejection as its revert, ensuring the current HEAD always has the best protocol found so far.

5.12.3 Constraints

For a fitness improvement to be attributable to a better protocol, the agent must be unable to improve its value any other way. The constraints below, including, the billing model, the workload and the convergence check exist to guarantee that. All of these have been established previously in this chapter.

The first constraint is the single file boundary. The agent may only edit `multi_replica.rs`, excluding updates to its own log. Every other component, including the engine, billing model, workload generator and test suite is fixed, as the agent is only supposed to change the protocol that runs against them. This the protocol isolation goal, described in section 5.5 applied to an adversarial optimizer rather than a cooperative author.

The second is the compliance with the `Protocol` trait, which enforces the closed billing surface described in the billing model section (5.4). The only way the agent has to emit messages is through the `Outbox` managed by the engine, whose properties make it impossible to change sender's identity, and can only read incoming messages through the `ProtocolMsg` accessors that in turn are automatically billed by the `WireSized` trait, either at send or receipt time, for state and metadata bytes respectively. There is no free data channel and no way to move information past the accounting layer.

The third is that the protocol holds no mutable global state, from which it can retrieve information it would not have access to in the case of a real deployment. The trait's methods take `&self`, which is not shared across the replicas. This forces every replica to rely on the simulation engine to hold any information that needs to be passed in between rounds in the engine's carry (5.4.3), which itself is kept as per-replica instance and only handed back to its owner. Combined with the monotonic union invariant whose job is to forbid a protocol from ever dropping elements (5.5.2), this leaves the agent free to design how replicas exchange and accumulate information but unable to cheat the simulation by sharing unbilled state or by quietly discarding the elements it is supposed to reconcile.

5.12.4 Memory and logging

Each iteration of the loop starts with no extra information besides what the agent writes to disk and its inputs. Its memory is therefore entirely external and is held in two files: `hints.md` (strategy) and `experiments.tsv` (past attempts log).

`hints.md` works as the agent's strategy memory. It takes lessons from previous runs, successful or not, and registers them for developing future hypothesis. Most importantly, it keeps a list of dead-end strategies to avoid retrying those. The agent reads it before forming each hypothesis so the search focus on unexplored moves rather than rediscovering already visited failures. Because it is curated across all the runs, it is a component that accumulates knowledge across the whole research rather than within a single run.

`experiments.tsv` is the log of every attempt. The agent writes exactly one line for every iteration with the identifier of the run , the fitness value, convergence flag and whether the change was kept or not. Every attempt is stored regardless of it being a success or a failure, since recording a dead end is what stops the same hypothesis from being tested multiple times. Attempts whose build or tests fail are also logged with a fitness value far outside any realistic value, so they can be clearly distinguished from genuine measurements. The agent reading this log before generating a new hypothesis is what lets its reason about its own history instead of exploring aimlessly.

5.12.5 Noise handling

A fitness improvement can only be attributed to the protocol's efficiency if it exceeds the measurement's own variability. Each cell is run three times under different seeds and reported with both its mean bytes and its respective standard deviation as described in section 5.9.

The agent uses this spread as a guard on the signal. A change is credited only if the improvement it claims is larger than the `std_bytes` of the cells it affects, A gain smaller than that is treated as noise and rejected. This keeps the agent from focusing its search on lucky changes, and the monotone fitness improvement described in subsection 5.12.2 would be advancing on noise. Averaging over seeds and screening the gains against their spread is what makes each accepted step a real reduction rather than an artifact of the singular runs that returned it.

Chapter 6

Evaluation

This chapter evaluates and compares the evaluation protocols discovered by the agent optimization loop described in chapter 5. Starting from a baseline implementation, the loop applied 63 successive modifications to the target file `multi_replica.rs`, having evaluated each over the full experimental matrix (5.8), that includes three network topologies (star, tree, chord), three set divergence levels and three runs per cell. The final configuration is the subject of this chapter. At 16 replicas it sends 30.4 MB on average, a 75.8 % reduction relative to the FST protocol (125.5 MB) and a 30.3 % reduction relative to the most efficient existing baseline, the RBF-RIBLT hybrid that sent 43.7 MB. Furthermore, it is the only sketch-based protocol that completes at all, the rest exhausting the available memory. Figure6.1 shows the optimization as a whole, with a short phase of large structural gains followed by a long plateau of changes that are nearly indistinguishable from noise.

The remainder of the chapter is centered around four questions. Section6.2 characterizes the discovered configuration and its behavior at scale. Section 6.3 decomposes where its savings come from and explains its major improvements, on many different scales including the byte types, topology and divergence . Section 6.4 reconstructs the search that led to the final result, by separating the moves that directed the agent to it from those that were rejected either for not improving the value beyond the standard deviation of each collection of runs or for failing to converge. Finally, section 6.5 describes the limits of these claims.

6.1 Evaluation setup

The experimental harness used to compare the protocols and the agent loop itself are described in depth in the previous chapter 5

This section's objective is to recall what is needed to read the results presented in the following sections. During the agent loop, every protocol is importantly evaluated over the same matrix with three topologies, three distinct divergence levels, two replica counts and three runs per cell for a total of 54 runs per cell. Every reconciliation process is capped at 100 rounds in order to avoid extraordinarily high numbers of rounds that despite not contributing to the total amount of bytes transmitted, in real world scenarios they may influence the efficiency of the protocol. A cell that

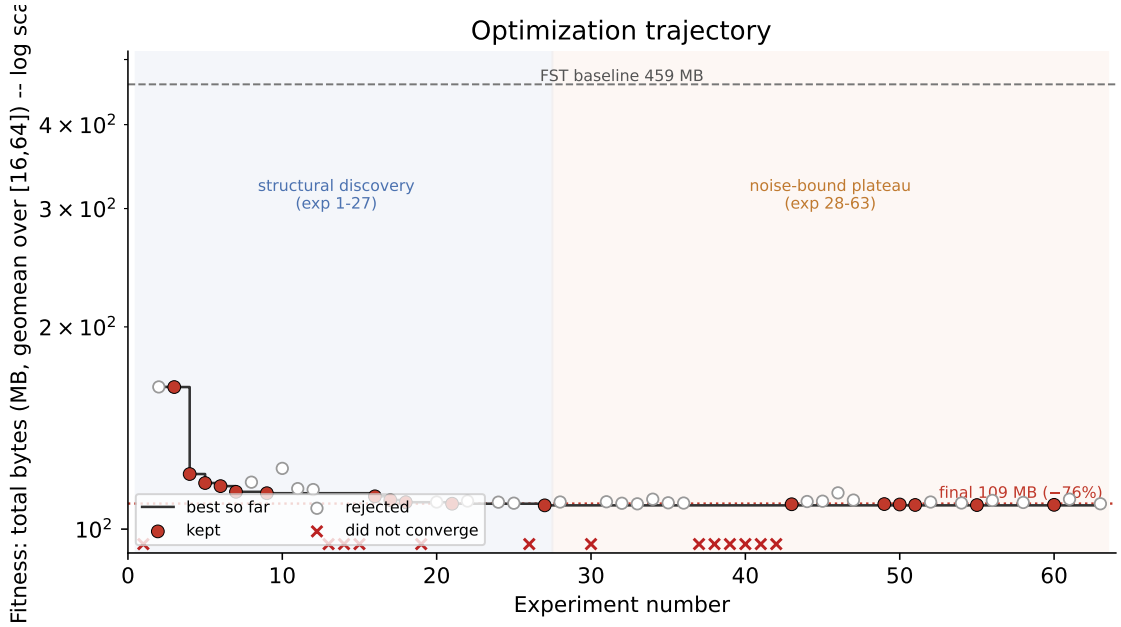

Figure 6.1: Optimization trajectory: best-so-far fitness across the 63 attempts, separating the structural-discovery phase from the noise-bound plateau.

fails to converge within the cap is reported as a failure, instantly invalidating that protocol, rather than contributing with a byte count. The metric used for comparing every protocol is the total bytes sent, which is aggregated as a geometric mean across cells, and s the same value the agent optimized for. This means, that as shown in the figures below, the fitness function and total bytes sent are directly comparable. Moreover every single run is run under a fixed set of seeds ensuring the results are reproducible as explained in section 5.11.

6.2 The discovered configuration

The best configuration found by the search loop described in Chapter 5 and the one reported throughout this chapter was fixed at commit `50d8b3a`. It is defined as the final result since it is the value at which the agent's hill climb stopped producing relevant improvements, capable of exceeding the run's noise.

The great improvement from in comparison with the baselines is not a better sketch but a topology dispatched scheme. This means a different reconciliation strategy was applied to different topologies as to handle the different cost structure each one presents. Importantly, these three mechanisms were all discovered by the search rather than specified in advance proving the usefulness of an autonomous loop integration in distributed systems search and are described in the following paragraphs.

The first is a changing FPR. Starts runs an exact low FPR filter (0.01), it uses the hub as a central distribution point that with a single hop reduces redundant traffic and keeps a tight filter that minimizes waster rounds. For the tree topology, the protocol opts for the complete opposite by

Table 6.1: The discovered configuration (commit `50d8b3a`)

Topology	Dissemination	False-positive rate
Star	All-neighbour BF	0.01 (static)
Tree	All-neighbour BF	0.27, → 0.30 above ∼70K
Chord	Pairwise distance-doubling BF	0.10, → 0.13 late

choosing an higher base FPR of 0.27, since skipping false positives shrinks its filters by more than the extra rounds cost, which is a tradeoff that this topology exposes.

The second is the adaptive late round FPR. As described in the literature, a higher FPR becomes prejudicial as the set converges. In the case for the tree and Chord topology, the final protocol effectively keeps it higher while the local set does not converge. This is the gain from exp7 to exp27 shown in Table 6.5.

The third is the Chord pairwise distance doubling. instead of comparing sketches with every neighbor, Chord nodes reconcile with logarithmically spaced peers, this roughly halves the Chord cells and is the largest single structural gain in the search as shown in Table 6.5 (exp3→exp4).

The effect is summarized in Table 6.2, where the baselines' cost grows with the among of hops per reconciliation run of the topology, the discovered configuration stays roughly flat across Star, Tree and Chord, which is the central finding in itself. In the following subsections describe in detail these improvements, first the aggregate headline (6.2.1), then its stability across every run (6.2.2), and finally its behavior at scale (6.2.3)

6.2.1 Headline comparison

Figure 6.2 and Table 6.3 report the geometric mean of total bytes sent across the full $3 \times 3 \times 3$ matrix of topologies, Jaccard levels and seeded runs. In this conditions all cells of every protocol managed to converge, making them accurate comparisons of the communication cost and not of feasibility.

As shown in the picture, the discovered configuration sends 30.4 MB, against 125.5 MB for FST, this is a reduction of 75.8 %. This value also beats all other baselines as RIBLT stands at 50.9 MB, Static BF+IBLT at 45.9 MB, Hybrid RBF+RIBLT at 43.7 MB. Putting the best solution found by the agent 30.3 % ahead of the best baseline in the matrix. This is the result of this chapter, the stability of this improvement is defended in the following subsection 6.2.2. And subsection 6.3.2 justifies it and explains why the improvements are most noticeable on the topologies with multiple hops instead of being spread evenly.

6.2.2 Stability of the best result

Table 6.4 shows the per-cell mean, standard deviation and Coefficient of Variation (CV) of the final configuration across the three runs of each cell at $n = 16$.

Table 6.2: Total bytes sent (MB, geometric mean over Jaccard levels) by protocol and topology at $n = 16$.

Protocol	Star	Tree	Chord
FST	47.1	243.2	172.7
RIBLT	38.6	49.3	69.5
Static BF+IBLT	29.5	52.2	62.8
Hybrid RBF+RIBLT	29.2	44.7	63.9
MultiReplica	**27.5**	**37.9**	**27.0**

The standard deviation never exceeds 1.2 MB on a multi megabyte mean, and the largest CV across all nine cells is 2.3 % for the Tree topology at 0.5 jaccard similarity. Star and Chord are tighter still, mostly below 1 % reflecting how reconciliation runs on the tree topology are longer due to its high diameter, accumulating way more jitter across the entire simulation.

These three runs per cell measure are meant to minimize the run to run noise effect on the protocol's result, by averaging the results and comparing fitness values with the standard deviation into consideration. Even though the runs are indexed by a seed $\in 42, 43, 44$, an audit of the harness revealed that this seed is inert and due to a configuration error never reaches the workload generator. Therefore, the three runs of a cell reconcile an identical workload and the spread in Table 6.4 is not caused by sampling variation across different workloads but the residual nondeterminism of the implementation, more specifically the randomized iteration order of the hash sets holding each replica's state, which perturbs the false positive pattern of the BFs which in turn influence the exact byte counts. The one line fix, which consists in routing the per run seed into the workload generation constructor are documented for future work. The results in this chapter were collected before the fix and are reported as is.

Importantly this issue does not invalidate the way the results are analyzed. The 2.3 % maximum is treated throughout the chapter as an implementation noise floor, or in other words, the smallest byte difference that cannot be attributed to a deterministic effect of a design change in the protocol. This value comes with two objectives.

First, it qualifies the headline of Section 6.2.1, as the 30.3 % margin over the best baseline is more that ten times this floor meaning it cannot be an artifact of the run it was recorded from but a robust result that can be attributed to a design change to the protocol. Second, it sets the threshold against which the search trajectory is read in Section 6.4.2. A new protocol can only be kept if its improvements exceed the 2.3 % margin, as otherwise it cannot be distinguished from noise. The idea behind a workload seed was to widen this floor, making the boundary more conservative, despite not affecting the order of magnitude of the headline margin, which is the load bearing claim of this thesis.

Table 6.3: Communication cost at $n = 16$ (geometric mean of total bytes sent over 3 topologies $\times$ 3 Jaccard levels $\times$ 3 runs; all cells converged). Reductions are relative to Full State Transfer and to the best baseline (Hybrid RBF+RIBLT).

Protocol	Total bytes (MB)	vs. FST	vs. best baseline
FST	125.5	–	–
RIBLT	50.9	−59.4%	–
Static BF+IBLT	45.9	−63.4%	–
Hybrid RBF+RIBLT	43.7	−65.2%	–
MultiReplica	**30.4**	**−75.8%**	**−30.3%**

6.2.3 Scaling to 64 replicas

The $n = 64$ is reported separately because three of the five protocols did not produce a result at all. Figure 6.3 show the outcome of running the same matrix at 64 replicas on a machine with 30 GB of Random Access Memory (RAM), with each run capped at 24 GB of virtual memory.

The three sketch baselines, RIBLT, static BF + IBLT and Hybrid RBF + RIBLT, exhaust memory before completing and are killed at the limit. This is not a direct result of the disadvantage of those protocols, but instead an artifact of the harness. The simulator needs to materialize every node's messages in memory simultaneously, which at $n = 64$, exceeds the limit imposed by the hardware. The Out Of Memory (OOM) therefore, does not provide any additional information about the protocol's communication complexity but is instead a limitation of the experimental driver running on a single machine and is reported here as a feasibility outcome rather than a byte count.

The two protocols that do complete, display the contrast the chapter exposes. FST, that survives the memory limit despite being byte catastrophic as it never builds large sketches, sends 1680 MB, an order of magnitude more than at $n = 16$, since every replica is required to ship its entire state to every other node, every round. The discovered configuration completes at 388 MB, roughly a quarter of the worst baseline, and is the only protocol in the matrix that is feasible within the memory budget and efficient in bytes sent. At this scale that combination is the entire result by itself, where the balance between not sending the entire state and not exceeding the simulator's memory bounds is met.

In summary, the baselines' OOM limitation is imposed by the simulator and not a property of the protocols and is revisited as such in Section 6.4.

6.3 Anatomy of the gains

The headline of Section 6.2, 76 % reduction against FST and 30 % against the best baseline is made over a single aggregate number, and by itself does not convey much information. This section splits the aggregate apart along three axes to show what is actually responsible for the improvements reflected in the fitness function.

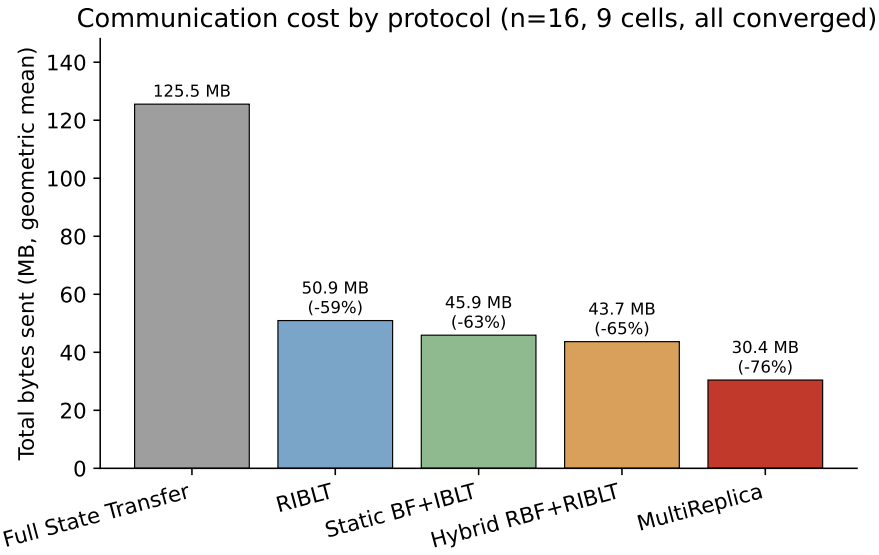

Figure 6.2: Communication cost at $n = 16$: geometric mean of total bytes sent by each protocol over the full matrix, with reductions relative to Full State Transfer and to the best baseline. The discovered configuration (MultiReplica) sends 30.4 MB, 30.3 % below the best baseline.

The first cut is by byte type (Section 6.3.1). Total cost is split into both state (reconciled elements themselves) and metadata bytes (sketches and filters shared to find the symmetric difference. In the multi party environment, unlike in the two party setting, finding the absolute minimum of state bytes transmitted is not trivial. Dissemination over a topology lets the same element be retransmitted across hops, making state bytes a quantity protocol's efficiency can differ on. Separating the two shows that this is one of the metrics the discovered configuration wins, by sending similar metadata overhead as the baselines but remarkably sending fewer bytes, most notably on topologies with multiple hops, by avoiding redundant retransmission.

The second axis is the topology (Section 6.3.2). The aggregate hides a sharp split, as on Start, a topology with a single hop,, the discovered configuration is level with the best baseline. The entire advantage is earned on Tree and Chord where naive all neighbor dissemination ends up resending the same elements across hops. The flatness of the configuration's cost across topologies is the central mechanism of this chapter.

The third and final cause for variation in protocols is the configured starting divergence levels across replicas (Section 6.3.3). Sweeping the Jaccard level tests whether the advantage is an artifact of the simulator or holds as the replicas become more or less similar, effectively guarding any per-cell claim on the noise floor described in Section 6.2.2.

Finally, Section 6.3.4 states the price of the reduction, as the byte savings that are bought with extra rounds. A fair account of the gain has to report that cost alongside it which cannot be discarded.

Table 6.4: Stability of the discovered configuration: per-cell mean, standard deviation, and coefficient of variation over the 3 runs at $n = 16$.

Topology	J	Mean (MB)	Std (KB)	CV (%)
Star	0.25	45.4	140	0.31
Star	0.50	29.8	259	0.87
Star	0.75	15.4	220	1.43
Tree	0.25	59.9	1206	2.01
Tree	0.50	41.8	958	2.29
Tree	0.75	21.8	320	1.47
Chord	0.25	44.6	109	0.24
Chord	0.50	29.3	94	0.32
Chord	0.75	15.1	269	1.78
max CV				2.29

6.3.1 State vs Metadata

A clarification is required before separating state from metadata bytes since the role of state bytes differs fundamentally from the two party setting. In pairwise reconciliation the state transferred is fixed by the workload as each element of the symmetric difference crosses the single link exactly once, making the amount of state bytes transferred a constant across efficient protocols. Therefore a protocol may only compete on metadata sent or, in other words, the overhead of finding out what elements compose that symmetric difference. In an environment with more than two replicas, however, finding that minimal amount of state bytes is not trivial, as an element must traverse multiple hops to reach every replica and a replica may receive the same element from more than one neighbor, so the number of element transmissions is no longer fixed by the symmetric difference and is now dependent on how the protocol disseminates over the topology. State bytes therefore become a quantity that protocols can differ on, with room to retransmit the same data redundantly even if that data is a subset of the symmetric difference of two given replicas at the time of transmission.

Figure 6.4 splits each protocol's total cost into the two classes making the point concrete. The three sketch baselines send an identical 32.5 MB of state. This is not a coincidence, but instead a direct consequence of them sharing the same dissemination strategy and only differing in the sketch used to decide what to end, making them always retransmit the same elements and diverge only in metadata (18.1 MB for RIBLT, 11.1 MB for Hybrid). The discovered configuration breaks that shared state floor. It sends 23.1 MB worth of state bytes, which represents a 29 % reduction in comparison with the baselines, just by disseminating differently rather than by sketching better.

The final configuration also wins on metadata by sending only 6.8 MB in contrast with the 11.1 MB sent by the Hybrid baseline. This means the reduction can be attributed to both classes of bytes, but with two different contributors. From the 13 MB that separates the discovered protocol from the best baseline, the larger share is state 9.4 MB and the smaller is 4.3 MB. This is the central finding of the decomposition. A better sketch can only ever reduce the metadata term of the total,

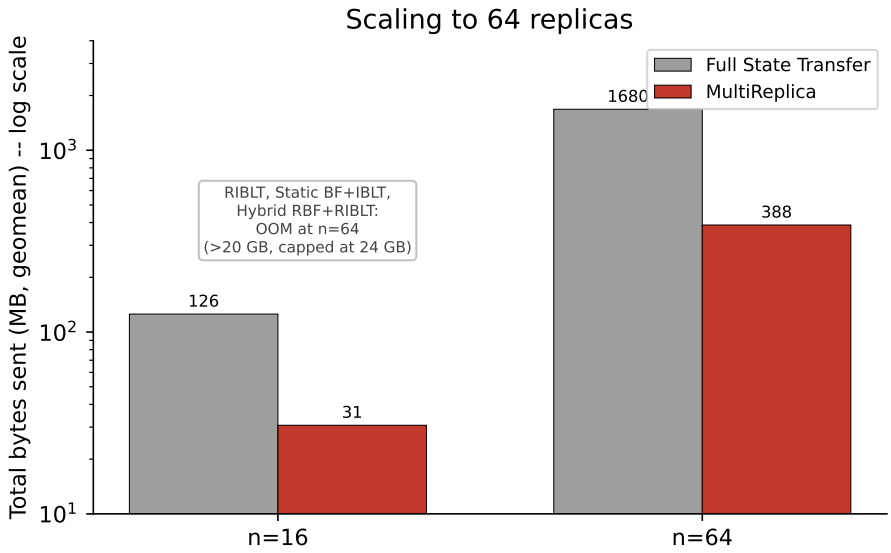

Figure 6.3: Outcome of the full matrix at 64 replicas (30 GB RAM, each run at 24 GB virtual memory).

as shown by comparing all baselines against each other, whereas to reduce state bytes transmitted, a fixed axis in two party environments, a protocol that takes advantage of the topology it is running on is necessary. Where this state saving is concentrated and why it is absent on one topology and large on another is the subject of Section 6.3.2.

It is important to note that this split is function of the payload size fixed in 5.7.2 as state bytes scale with payload size whereas metadata, computed from the digests alone does not. Despite this, the relative difference between protocols' results is independent from this value as their state bytes cost scales in the same proportion with it.

6.3.2 By topology: where the state saving lives

As mentioned in the previous section, the aggregate state saving is not spread evenly across every topology. Figure 6.5 plots state bytes per topology for the discovered configuration against the baselines (FST is omitted).

Star is the control. Since every other leaf can be reached through one intermediary, the hub there is no path with multiple hops on which an element can be retransmitted redundantly. ON Star, every protocol sends essentially the same state 24.3 MB, except for the discovered configuration that stands at 24.6 MB. This inequality is important as it shows there is no intrinsic advantage in the elements the multi replica protocol transmits, given that in a topology with no redundancy to remove, it removes none.

The advantage only becomes relevant once the hop depth grows. The three baselines which share a single dissemination strategy that passes through all neighbors for every replica, send state that increases with the topology's diameter 24.3 MB for Star, 38.7 MB on Tree, and 50.2 MB on Chord. The same element is retransmitted across multiple hops and arrives at a replica from more

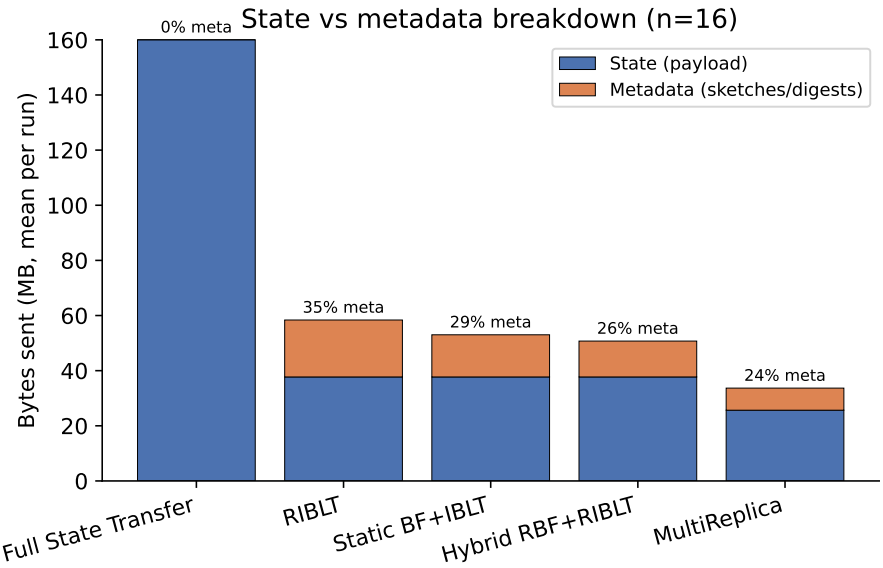

Figure 6.4: Total communication cost at 16 replicas split into state (reconciled elements) and metadata (sketches and filters), geometric mean over the full matrix.

than one neighbor, with each redundant arrival being a state byte that need not have been sent. The discovered configuration instead holds state almost flat across topologies, with 24.6 MB, 27.4 MB, 25.0 MB suppressing that redundant retransmission across hops. This flatness, against a baseline cost that increases proportionally with diameter, is the central mechanism of this chapter, showing there are gains to be made by disseminating in a topology aware way and not by encoding the difference better.

This does not come with its own downside. As shown in the decomposition, on the Tree topology, the discovered configuration spends more metadata than the best baseline (13.8 MB against 10.6 MB). This is caused by the high FPR adaptive filter it runs there (Table 6.1).It is a mechanism that trades metadata against state, since a more accurate filter is larger to describe but suppresses more redundant elements. The advantages shows in the total amount of bytes sent, as the tree falls from the best baseline's 44.7 to 37.9 MB. Importantly this is a a trade and not applicable to every scenario, being the reason the configuration uses a different operating point on each topology rather than one global setting.

6.3.3 By divergence: the advantage holds across the Jaccard sweep

This subsection focus on how the divergence between replicas influences the total bytes sent across protocol's. A per-cell win at one operating point could be an artifact of a single Jaccard level. The real insight is whether the advantage survives as the replicas' sets become more or less similar. The Jaccard similarity index was swept across $J \in \{0.25, 0.50, 0.75\}$ and the total bytes sent were reported as a geometric mean over the three topologies and three runs per cell (Figure 6.6). Note that every comparison below clears the noise floor of Section (6.2.2).

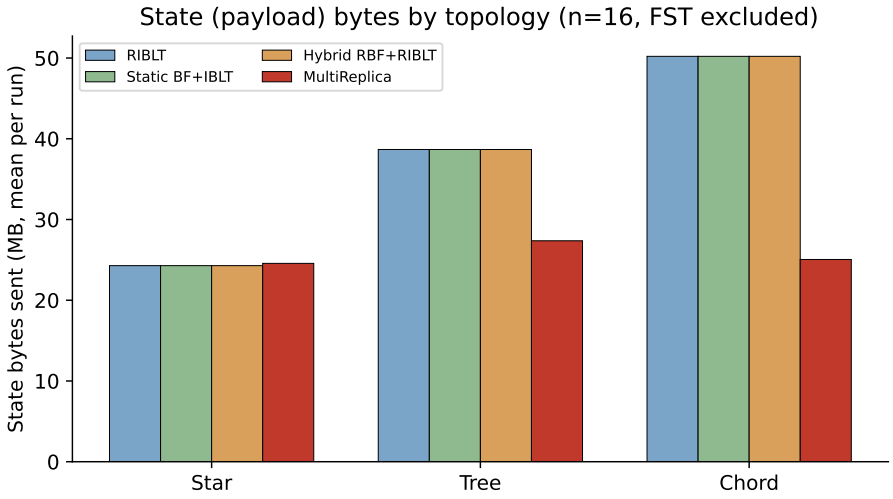

Figure 6.5: State bytes sent per topology, geometric mean over the Jaccard sweep at 16 replicas (FST omitted; it transfers full sets and dwarfs the sketch-based protocols).

The advantage accrued by MultiReplica is stable across the entire sweep. MultiReplica sends 49.5 MB at $J = 0.25$, 33.2 MB at $J = 0.50$, and 17.2 MB at $J = 0.75$, against 73.4 MB, 47.4 MB, and 23.9 MB for the best baseline (Hybrid RBF+RIBLT) at the same points. The relative saving is 32.6 %, 30.1 %, and 28.2 %, roughly a third of the traffic removed at every divergence level, never narrower than 28.2 %, proving the result is not the product of one fortunate point of divergence.

From the values it can be seen the margin widens as the replicas diverge. This occurrence is consistent with the mechanism of Section 6.3.2 that states the gains come from suppressing redundant cross hop retransmissions. Since a larger symmetric difference means a larger amount of elements to reconcile, it also means more opportunities for a naive dissemination strategy to retransmit, inefficiently, elements known to belong to the symmetric difference between two given sets. At $J = 0.75$, the symmetric difference is small, meaning there is little redundancy for any protocol to exploit, so the gap compresses to 28 %. At $J = 0.25$ the larger difference gives MultiReplica more redundant traffic to eliminate, and the gap opens to 33 %.

This cut also reproduces the by byte type finding of Section 6.3.1 at every divergence level. The three sketch baselines carry identical state at each Jaccard point(55.6 MB, 35.5 MB, 17.4 MB for $J = 0.25, 0.5, 0.75$) and differ only in metadata, since they disseminate the same elements and diverge only in how they encode the reconciliation. MultiReplica sits below the baseline state line at every point (39.2 MB, 25.3 MB, 12.4 MB), confirming that the saving comes from sending fewer bytes, not merely a cheaper sketch across the full range of divergence.

6.3.4 The cost: convergence rounds

The byte savings reported in previous rounds are not free. MultiReplica achieves them with latency as it takes more communication rounds to converge than any other baseline, on every topology (Figure 6.7). The mechanisms that suppress redundant traffic, namely the high false positive

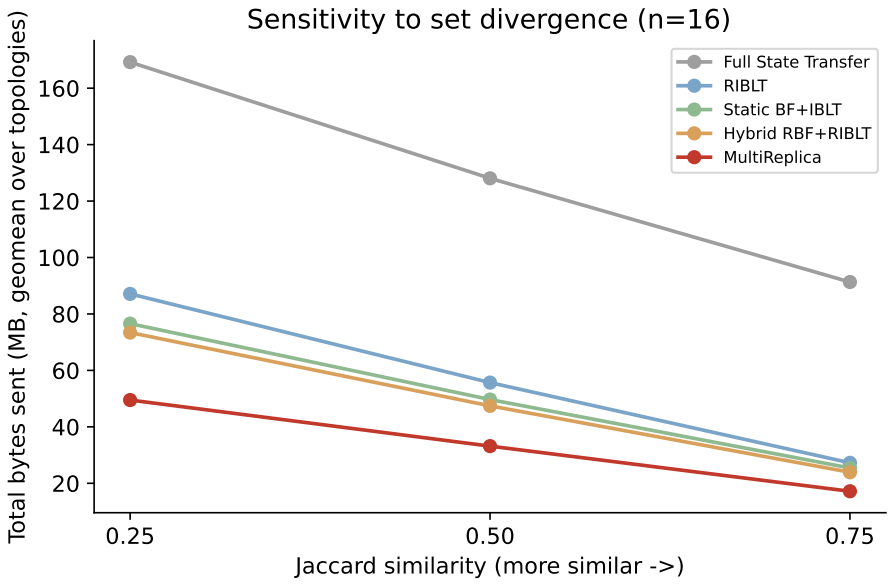

Figure 6.6: Total bytes sent versus Jaccard similarity at 16 replicas, geometric mean over the three topologies and three runs per cell.

adaptive filters and on Chord pairwise distance doubling, deliberately move less information per round, spreading reconciliation over more, but cheaper, rounds.

The gap is widest on the Tree topology. There MultiReplica needs 57.7 rounds on average, against 18 for all three sketch baselines and 7 for FST, roughly 3.2 times the sketch baselines and 8 times FST. On Star, the rounds are 8.2 versus 4, and on Chord 14.2 versus 4. The three sketch baselines converge in a similar amount of rounds in each topology (4/18/4 for Star/Tree/Chord). This was expected since they share the same dissemination schedule and differ only in how each round's payload is encoded. MultiReplica changes this schedule itself, trading rounds for bytes. The round count remains stable across the Jaccard sweep, meaning the latency cost is a structural property of the protocol and not an artifact of one operating point.

The advantages of this trade are dependent on the topology and it is worth being precise about where they are meaningful and where they are not. On Chord the reduction in bytes sent clearly outweighs the round latency (14.2 rounds versus 4), a 3.5× latency cost for saving roughly half of the bytes sent (Section 6.3.2). On Tree the results are the weakest achieving a reduction of 15 % ((37.9 MB versus 44.7 MB) at the cost of an increase of nearly 30 extra rounds. This is due to the high FPR adaptive filters that hold Tree state down also force many additional corrective rounds. Whether this tradeoff is acceptable or not depends entirely in the deployment, as MultiReplica will favor environments where the binding constraint is the bandwidth and round latency is acceptable, in opposition to tightly synchronized settings. Importantly, every run still converges well inside the round cap of 100 (Section 6.2.2), so the latency cost, despite being important never threatens convergence itself.

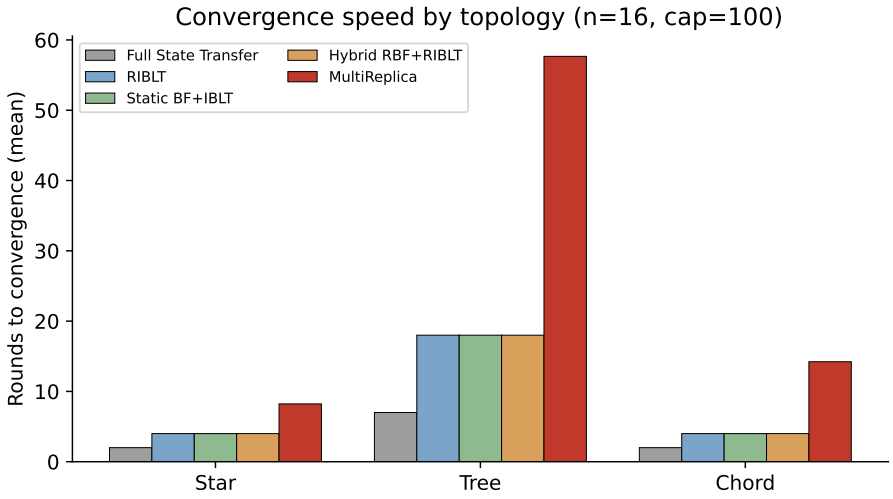

Figure 6.7: Mean convergence rounds per topology at 16 replicas (over the Jaccard sweep and three runs per cell).

6.4 The optimization process

The final configuration was product of a systematic search by an autonomous agent in a loop whose objective was to lower the fitness function at any cost starting from the FST baseline with fitness 459 MB as described in the previous chapter (See 5). The agent was instructed to propose and test one design change at a time, that being a different FPR, a topology specific dispatch or a new dissemination rule. Each candidate was then tested and benchmarked, and accepted only if it lowered the fitness while still ensuring every cell had converged within the round cap while a change that raised the fitness, or fell short of total convergence was reverted. This search loop was ran for 63 attempts and terminated at 108.7 MB, 76.3 % below the initial baseline (Figure 6.1).

The trajectory, which is the history of all the commits and reverts made by the agent holds the information about the kind of moves tested. Three kinds of changes appear and this section revolves around them. Only a small number of structural moves drove almost the entire optimization, which were establishing the sketch-based protocol and the dissemination strategy dispatch by topology. These were the changes that cleared the run to run noise floor of Section 6.2.2 (Section 6.4.1). Behind them sits a long list of small FPR adjustments, each individually below that floor, yet kept because they do no harm and when put together, account for the last few percent of the gain.

The second kind of relevant changes are the noise floor rejects. Many candidates produced a fitness change smaller than the $\pm 2.3\,\%$, despite representing a reduction of the fitness function they were correctly discarded as ties rather than mistaken for progress. One (exp64) even scored above the band and as a genuine regression. Distinguishing these from the kep sub noise tail above is what keeps the search honest. Section 6.4.2 shows the plateau under the noise band directly.

The third kind of moves registered doing the search were the ones that did not converge at all. Five distinct types of change were recorded that can fall under the following categories, send suppression, over aggressive FPRs, pairwise edge cycling, skip one markers and a rateless sketch.

Table 6.5: Attribution of the byte reduction to successive kept design changes

Exp	Design change	Fitness (MB)	Δprev	vs. FST
3	BF-only (fpr=0.01) multi-round all-neighbor sketch	162.7	−64.6%	−64.6%
4	topology-dispatched: Chord pairwise distance-doubling BF asc, Star/Tree all-neighbor BF	120.8	−25.8%	−73.7%
5	bump fpr 0.01→0.05 globally	117.1	−3.0%	−74.5%
6	per-topology fpr (Star=0.01 Chord/Tree=0.05)	115.9	−1.1%†	−74.8%
7	bump tree fpr 0.05→0.1	113.6	−1.9%†	−75.3%
9	bump chord fpr 0.05→0.1	113.1	−0.5%†	−75.4%
16	tree fpr 0.1→0.15	111.9	−1.1%†	−75.6%
17	tree fpr 0.15→0.2	110.5	−1.3%†	−75.9%
18	tree fpr 0.2→0.25	109.7	−0.7%†	−76.1%
21	tree fpr 0.25→0.27	109.1	−0.5%†	−76.2%
27	adaptive tree fpr (0.27 base, 0.32 if set>100K)	108.5	−0.6%†	−76.4%
43	revert all pairwise/skip-one; restored exp27 baseline (best 108.5M-108.9M range)	108.9	+0.4%†	−76.3%
49	chord adaptive (0.1 base, 0.15 if set>100K)	109.0	+0.1%†	−76.3%
50	chord adaptive late 0.15→0.13	108.8	−0.1%†	−76.3%
51	tree adaptive 0.27/0.30 thresh 80K	108.6	−0.2%†	−76.3%
55	tree threshold 80K→70K (late 0.30)	108.6	−0.0%†	−76.4%
60	chord BF+RIBLT for late rounds (set>100K)	108.7	+0.1%†	−76.3%

These either repeatedly drove cells into the round cap, or in the case of the rateless variant exhausted the time budget and were abandoned. They were treated as boundaries of the design space the search could use, and Section 6.4.3 catalogs them so the negative result is recorded alongside the positive one.

6.4.1 Structural moves that drove the reduction

Reading the decomposition table 6.5 top to bottom, only three attempts produce a change large enough to clear the run to run noise floor of Section 6.2.2.

The first is establishing the protocol itself. Replacing FST with a BF sketch disseminated over multiple rounds (exp3) takes the fitness from the 459 MB baseline to 162.7 MB, a 64.6 % reduction in comparison with the first baseline. This is the cost of moving from shipping entire sets to shipping a compact summary of them, and it is the single largest move in the search.

The second is dispatching the dissemination strategy by topology. This move occurred in (exp4). Giving Chord a pairwise distance doubling schedule while leaving Star and tree on all neighbor

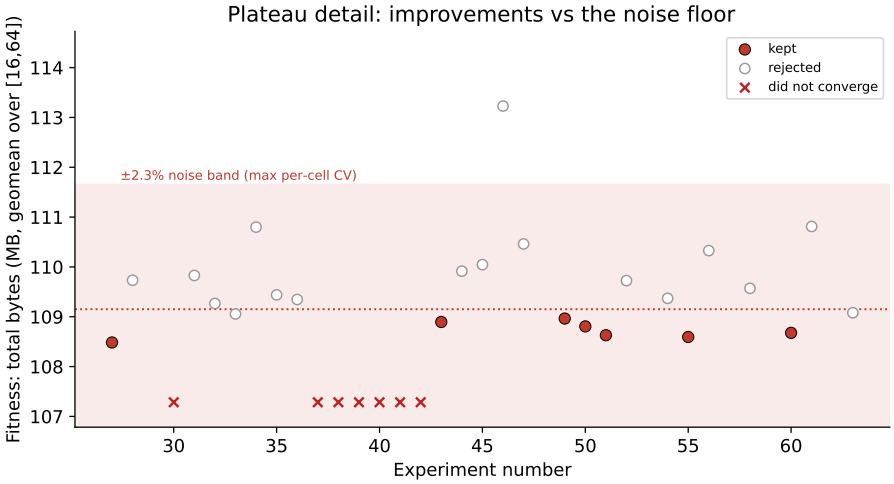

Figure 6.8: Zoom into the search plateau

broadcast cut the fitness from 162.7 MB to 120.8 MB, a further 25.8 % improvement. The saving is concentrated in the Chord cells, which are roughly halved, and it is the structural evidence for the central claim of Section 6.3, that the gains come from matching the dissemination pattern to the communication graph, not from a better sketch. Under this schedule a Chord node does not sketch to all of its neighbors each round. Instead it reconciles only with the peers at distance 2^p round the ring, forward and backward, with the exponent p advancing each round through $0, 1 \ldots, \lfloor \log_2 N \rfloor$. Because in the Chord topology, there are only edges to power of two neighbors, effectively this approach steps through one distance shell at a time from the closes neighbors outward. This allows for the near pairs to reconcile first before the longer range exchanges that usually encompass more redundant data.

The third is a global increase in the BF FPR, from 0.01 to 0.05 (exp5), which lowers the fitness to 117.1 MB. This produced the lower reduction out of the three with only 3.0 % gain that clears the 2.3 % noise floor but only narrowly. Following its instructions the agent still considered it, however we report it with the caveat that it only works as the higher positive FPR permits to save more metadata on Tree and Chord than the extra correction rounds cost on Star.

After exp5 the search enters a long sequence of small changes. Every subsequent kept attempt, that include per topology FPRs, successive Tree and Chord rate bumps and adaptive late round thresholds and a BF+RIBLT hybrid approach for large sets in Chord, produce a per-step change marked in Table 6.5, meaning it falls inside the noise floor. However, they were still kept since they do not harm the fitness function and together they carry the fitness from 117.1 MB down to its final 108.7 MB. These sequence of small reductions cannot be looked at as a singular improvement but have to be treated as an aggregate. The final configuration owes its headline margin to only three structural decisions and some final tunings.

Table 6.6: Approaches that failed to converge (or were infeasible) grouped by failure mode.

Approach	Exp.	Failure mode
Rateless BF+RIBLT (all-neighbor)	1	Too slow: eval killed after 51 min at $n = 64$ (not a convergence failure).
Tree send-suppression (deterministic / random drop)	13, 14, 15	Starvation: dropped messages slow multi-hop propagation past the 100-round cap.
Over-aggressive (late / adaptive) Tree FPR	19, 26, 30	Late-round starvation: high FPR skips needed elements; round cap hit.
Tree pairwise edge-cycling	37, 38, 39, 40	Needs more cycles than the cap allows; all cells reach round 100 even at $n = 16$.
Tree skip-one with markers	41, 42	Broke lockstep convergence (parent always skipped); abandoned.

6.4.2 Rejected at the noise floor

Section 6.2.2, centered on the best configuration. The purpose of this subsection is to show that the search took into account the limit of its own measurements. After the fitness reach ed near 108.7 MB, almost every candidate was well within the noise band, making overfitting to measurement noise more of an issue. If every move that returns a slightly lower number is accepted, the search ends up tuning to a favorable run to run draws and reporting a configuration that is not genuinely better than the previous ones, but just luckier. The defense against this issue is the noise floor already described in Section 6.2.2, used as a rejection rule. Figure 6.8 zooms into this plateau and draws the $\pm 2.3\,\%$ band of repeated variation across the best configuration

This subsection focuses on the points that land inside that noise band, which is the zoom the picture gives. Whether by tightening or loosing the Tree rate around 0.27, moving the moving the Star rate off 0.01, sliding the adaptive threshold between 60,000 and 150,000 elements, each affects the fitness function in less than a percent, a change smaller than the spread between runs of the same configuration, making them no more a regression than an improvement. These are rejected, not because they put the search head in a worse spot but because they do not lead to better results, and keeping them would have been keeping noise. If interpreted positively, this cluster is the evidence that the kept configuration sits at a genuine local optimum as every small step away from it lands back inside the band it started in

This raises an apparent inconsistency with Section 6.4.1, where a sequence of equally sub-noise moves that was kept. These two approaches are not the same situation with different outcomes but instead two different situations that need careful handling and justification. The kept sequence is a staircase in the same direction, as each change pushes a parameter further along a productive gradient, the Tree FPR from 0.1 toward 0.27, for instance, and the steps compound, so that while no single change clears the noise floor but the improvement from the first change that is part of that chain to the final one does (Section 6.4.1). On the other hand, the moves rejected here are the steps that try to continue past the point where no improvement is being made, or in other words the search flattens. The rule to keep changes is therefore to follow a direction while it keeps paying

and stop when that accumulation halts. Figure 6.8 shows this phenomenon. The rejections in this subsection are precisely the evidence that the search located where each gradient stopped paying, and the fact that the fitness does not continue downward past them is what justifies why the kept configuration is a true local optimum rather than an arbitrary stopping point.

A few changes do break out of the band and are genuine regressions that confirm that the autonomous agent loop can still get out of this local optimum. The clearest is exp46, which by raising the Star hub's FPR to 0.05 while while leaving the leaves at 0.01 led to an increase of the fitness function 113.2 MB, roughly 4 % above the noise floor, since the coarser hub's filter nearly doubled the number of Star rounds (from 9 to 17) leading to an inflation in metadata. A few others also also stand out, for example, replacing the Chord schedule with pure RIBLT (exp10) led to a 13 % regression or switching Star and Tree to an exact BF+RIBLT decode (exp8) led to a 8 % regression. It is safe to assume the noise floor is fulfilling its role and discarding changes it cannot distinguish from variance and flagging the ones it can.

6.4.3 Approaches that broke convergence

The third type of changes were the ones that did not produce a number due to not having reached convergence. These approaches were cataloged in Table 6.6, and do not represent inefficient byte count but actual failures either for exceeding the round cap or for not finishing at all. Their relevance appears as a limit the design space of the search by marking a potential byte efficiency improvement as unfeasible for future iterations.

These fall under two major categories. First, the strategies that despite being able to reconcile in theory cannot do so within the 100 round limit imposed by the simulator. Four out of the five failures fall under this category for the common reason of having to advance every single element element a single hop per round on deep topologies such as Tree. Second, a protocol that fails to reconcile at all within a significant amount of time. There is only one protocol of this type in which the run was killed after 51 minutes at 64 replicas. The following paragraphs describe each of the failures and what part of the search space they limit.

First, send suppression (exp13-15) drops a fraction of messages each round, either deterministically or at random to cut traffic directly. Despite being successful in doing so, as it lowered state by 23 % the dropped messages slow propagation passed the 100 round cap.

Another improvement attempt was based on using over aggressive FPRs (exp19, 26, 30). The agent tried both setting them statically and raising them adaptively in late rounds, failing for the opposite reason as the previous. A high FPR BF is cheap at the cost of skipping elements that a peer genuinely needs with more frequency. Past a threshold around 0.3 on Tree, the skipped elements are never requested again within the cap and the cells hang. The boundary that the kept adaptive tree rule (base 0.27, rising to 0.30 on large sets) manages to respect.

Third failed attempt was with pairwise edge cycling (exp37-40). In this approach, the simulator reconciles one neighbor per round rather than broadcasting to all of them. Even though, on Tree, this produced the largest byte savings seen anywhere in the search, with a metadata reduction of nearly 68 % and state reduction of 30 % (before reaching convergence), it failed to converge within

the round limit even after easing the FPR at 16 replicas. The saving was real and unreachable , which is why Tree keeps all neighbor broadcast despite the metadata cost noted in Section 6.3.2.

The final change that reached the round cap is focused on skip one markers (exp41-42). These experiences tried to recover some of that pairwise saving by omitting one neighbor per round and announcing that omission. Since this broke the structure the Tree dissemination relies on as the parent at index zero was always the one skipped, the convergence failed regardless. This approach was therefore abandoned and recorded for future iterations.

Finally, the only failure due to an actual convergence failure and not due to a round limit issue. In exp1, the use of a BF+RIBLT sketch that encoded all neighbors sets was too slow to evaluate, leading to the killing of the run after 51 minutes at 64 replicas without completing. This experience, specially for having been run and documented so early on in the search loop allowed to set the boundaries on computational feasibility from the get go, ruling out a full trace of impossible changes.

Together, these five help defining the usable region the agent should explore. The final configuration is, in part, the product of these boundaries.

6.5 Threats to Validity

The results described in the previous sections hold only within the boundaries of how they were produced. This section details each and everyone of those boundaries and justifies the various scoping choices of the evaluation framework, such as the non-goals of Section 5.2 and the issues that surfaced only once the experiment was run. These were organized by the area each threatens. First, the construct validity, whether or not the fitness metric measures what it claims to. Second, the internal validity, whether the measured differences are real and caused by the protocol rather than measurement noise. Third, conclusion validity, which translates to whether the discovered protocol is genuinely the best so far and if it was reliably found. Finally, external validity or whether these findings can generalize beyond the tested configurations and simulator environment used. Importantly, none of these threats invalidates the headline result, but each one defines the limits of the claims made by this thesis.

6.5.1 Construct validity

The construct validity assesses whether the fitness metric measures the quantity the thesis claims it does. The clearest threat from this angle is the small coverage of the elected function. As it stands, it only count bytes transmitted on the wire and ignores compute time and number o rounds (Section 5.2). This leads to the finding of protocols that trade away CPU usage or extra round trips in a way that is legitimate under the simulator's rules yet not necessary a win in a real world scenario. A related limitation is the abstraction gap between the model and a deployment described in Section 5.2, since only application-layer bytes are charged while network latency, packet loss and fault tolerance overhead are left out. When put together, the final value achieved by the discovered

protocol is only acceptable under the assumption of a reliable transport layer beneath each protocol, and should not be read as a ranking on latency, round trips or fault tolerance.

6.5.2 Internal validity

Internal validity in this thesis is the extent to which the simulator can guarantee a relationship of cause-effect between every claim made throughout. In this case the protocol's design with its bandwidth efficiency. The primary threat to this measurement is related to the noise associated with its reading. Since BF hashing is order sensitive, repeated runs of the same configuration jitter by up to a 2.3% CV (the noise floor established in Section 6.2.2), so any byte difference below that floor cannot be attributed to the protocol with confidence and is treated as indistinguishable. The second issue with this topic is the inert workload seed. Due to a bug in the code base found only after all the results were taken, all runs were executed with the same workload, but since the non deterministically hash ordering provided some sort of randomness the three runs per cell are not byte identical. Even though this randomness is not the one the seed was meant to supply, it still perturbs the hashing of a single fixed workload rather than drawing a new (different) workload instance. Summing up, the noise floor measures implementation jitter on one singular problem instance and not workload variance. Therefore during this thesis no claims are made that a result can be generalized to different workloads at a given Jaccard similarity level, but instead whether or not it can be interpreted as a true signal or measurement noise. The bug is acknowledged rather than re-run because the headline exceeds the jitter by roughly an order of magnitude.

A third threat is of a different kind, related to the agent search itself. The agent sole goal is to optimize a single scalar and any weakness in the fitness function is something that can be exploited. The harness constraints are put in place to prevent it from tampering with byte counts and with data it should not have access to, however they cannot prevent it from a genuine misconfiguration of the objective. Summing up, a low score can only be attributed to better bandwidth efficiency to the extent that the objective faithfully encodes it.

Taken together, these threats bound the causal claim rather than void it, by attributing results below the noise threshold to implementation jitter and the ones above to protocol efficiency for a given workload.

6.5.3 Conclusion validity

In this context, conclusion validity assesses the conclusion drawn about the discovered protocol that it is the best configuration found by the search loop that the search can reliably arrive at it from the data. The first threat has to due with the nature of the search itself. The agent hill-climbs from one starting point, accepting improving moves, so it can settle in a local optimum. Since this is a greedy with a single trajectory that was only run once, the honest reading of the outcome is therefore that the reported configuration was the best found so far and not the proved optimal. A different seed or starting point might reach a different plateau. Another second threat is the potential of overfitting to the matrix. The fitness is the geometric mean over eighteen fixed cells at the two scales studied

(Section 5.8). This makes it so the configuration may be tuned to those particular points rather than expressing a property that generalizes to every other scenario. The topology dispatched rules it discovered are, by construction, fitted to that exact topologies and divergence levels in the matrix and carry no guarantee outside them.

Despite looking limited, these define the final result of this chapter as a reproducible local optimum over a defined objective and it should be read as the best configuration the search located on this matrix instead of the optimal general reconciliation protocol.

6.5.4 External validity

External validity concerns whether the results generalize beyond the setup they were retrieved from.

The first limit is the coverage of the axis. The matrix spans three topologies, two replica counts and three Jaccard values in the range $0.25 \rightarrow 0.75$ (Section 5.8), which deliberately excludes the near identical and near disjoin extremes, meaning the results are warranted inside this region and not at the endpoints. A second limit is imposed by the workload realism, as it stands, the replicas' sets are drawn from a synthetic Zipf distribution with a fixed universe and exponent that are not themselves swept, so a real deployment whose data has different structure, with a different skew, correlation between replicas or churn over time is not accounted for. The third concern is the memory limits observed at scale, with the sketch baselines exceeding memory at $n = 64$. This was considered to be an artifact of all neighbor message accumulation in the single-machine harness instead of a property of the protocols themselves. This makes it so that the feasibility advantage reported for the discovered configuration at that scale is a claim about behavior under this implementation and should not be read as a complexity result.

These design limitations and their respective justifications described in this subsection, effectively limit the conclusions that can be taken from the reconciliation protocols found in this thesis, by only ensuring their effectiveness in this specific scenario, leaving their generalization to other regimes, workload structures or implementations for future evaluation.

Chapter 7

Conclusion

This thesis set out to carry efficient set reconciliation from the pairwise case, where rateless techniques approach the optimal communication cost, hand in hand with other ECC state of the art solutions, into the multi party environment, where applying those same techniques naively across every pair of nodes introduces the quadratic redundant communication issue they exist to eliminate. Instead of designing every protocol by hand before measuring its performance, it posed the problem as a question of whether an autonomous loop driven by an AI agent would be a good git for research in distributed systems, meaning if such a loop given only a well structured simulator and a bandwidth objective can find an efficient multi party strategy solution with design decisions a human might not reach for. The previous chapters built that simulator and with it the conditions every protocol generated by that loop would be tested on, and measured the results produced by it across 63 experiences . This chapter analyzes what the discovered configuration means and if the resulting search loop results show how far an autonomous loop can be trusted as an instrument of research.

7.1 Summary of contributions

This thesis has three major contributions.

The first is methodological. It framed multi party set reconciliation as a search over protocol configurations and built a modular, reproducible and more importantly expandable simulator in which an autonomous agent loop can conduct a search with the objective of optimizing a single bandwidth scalar, without the risk of the agent cheating, across a matrix of topologies, replica counts and divergence levels. Furthermore, the loop ran the entire search without human intervention in the inner loop, while registering every change and documenting the results, which establishes this architecture as a usable instrument for research in distributed systems and not just as a parameter tuner.

The second contribution is the final protocol the search produced by the above mentioned agent loop. MultiReplica is a topology dispatched dissemination scheme, that uses round of BF sketching

and element delivery to reconcile every involved replica's set. This dispatched structure rather than a single uniform scheme, is what the loop converged on and it is described in full in Chapter 6.

The third contribution is the efficiency of the final protocol. Starting from a FST at 459 MB, the search reached 108.7MB, a 76.3% reduction. At 16 the discovered configuration costs only 30.4MB, a 30.3% reduction against the strongest baseline (Hybrid RBF + RIBLT at 43.7MB). At 64 replicas the advantage becomes more of feasibility as the sketch baselines exhaust memory under the harness's all neighbor accumulation while MultiReplica completes, making it the only protocol that is both feasible and efficient at that scale. This infeasibility of the baselines at this scale, as described in the threats to validity section is considered a property of this implementation of the simulator rather than an property of the protocols themselves, but it is still a positive signal of the discovered protocol's efficiency.

7.2 Discussion

Arguably, the most informative outcome of this work was not the protocol itself but how the answer was come about. The loop did not find a single clever mechanism that wins everywhere, but instead adapted the strategy to the topology it was being run on. Which led to the conclusion that most of the savings came from avoiding redundant dissemination rather than from a better two party primitive.

This is the kind of result an agent loop is a good fit for, as with a protocol designed by hand is easy to miss any strategy that was outside of our first assumption. A human investigator typically commits to one mechanism that is then fine tuned for a specific purpose, fixing the choice of mechanism before the experiment even begins. The agent loop, on the other hand, treated that same mechanism as a part of the search space, eventually leading to a different configuration for each topology.

For the multi party reconciliation problem as a whole, this reframes where there is remaining efficiency problems to exploit. Porting pairwise techniques to an environment with multiple replicas yields little on its own if only applied pairwise across the topology graph. What matters at scale is the coordination, and how information propagates over the topology. The headline reduction reported here is a direct consequence of this.

Finally, the discovered configuration and all the results described across the entire search provide the evidence that the loop behaved less like an opaque optimized and more like a generator of hypothesis that allowed the researcher to understand, explain and defend each one. Its reach, however was limited by the objective and constraints applied by the simulator, that if changed would deflect the research agent from the main objective of this study. In short, the research loop is not a replacement for the researcher's judgment but instead a capable explorer that expedites the part of coming up with new protocols with a chance at beating the current best taking into account previous attempts and issues some of which can be defined by the researcher himself.

7.3 Future work

The limitations defined in the threats to validity section (See 6.5) are not only problems of the simulator but also new direction in which the work can be carried further. The two most immediate concern the foundations of the measurements taken in this thesis and are developed in the subsections below.

In addition to these, two lighter directions are presented in this section. The objective of the search could be enriched beyond raw bytes, namely with a normalized efficiency metric against a theoretical lower bound, and the per-node load distribution. Both of these metrics were instrumentalized but left out of scope. Since the autonomous loop has proved to be a credible research instrument here, it could be used in this kinds of problems provided an insightful fitness metric was defined.

7.3.1 Reproducibility and workload sampling

The source of the internal validity threats about the measurement noise (Section 6.5.2) has already been flagged with a documented fix. There are two sources of nondeterminism, which are the random keying of the BF hashers and the unspecified iteration order of the hash sets the workload is build from. This could be avoided by seeding the used hashers and replacing those sets with an order stable structure, in order to ensure that given a fixed seed, every run reproduces a byte identical run.

The other issue, with reproducibility described and accounted for in that same section is the inert workload seed defect that could be repaired by carrying over the seeds chosen in the configuration file to the workload builder before the simulation starts. This would make it so the seed actually drives workload generation making the three runs of each cell reconcile distinct workload instances drawn at the same statistical parameters instead of one fixed instance hashed three different ways.

If put together, these two changes would allow this evaluation to make claims it does not currently have, as the noise floor used to distinguish ties from true improvements is not workload sampling variance but implementation jitter.

Acting on this would mean regenerating the entire matrix of results potentially with different results as the ones presented in the results chapter (6), since both changes impact the byte count.

7.3.2 Broader coverage

This subsection addresses the tested configurations that can be extended in future work, described in the external validity threats (6.5.4).

The similarity axis does not reach its extremes, the matrix only spans Jaccard values from 0.25 to 0.75 leaving the near-disjoint ($J \to 0$) and near identical ($J \to 1$) regimes unexamined. The structural axes are similarly limited and could both be widened to intermediate and larger scales or to structures deliberately left out in this study such as the gossip networks. The zipf distribution used to draw the elements is another issue. A synthetic workload may hide real traces

from application usage that motivate the work, such as mempool synchronization or stores based in CRDTs. By taking these into consideration, the reconciliation simulator would be capable of testing whether the discovered ranking holds under the skew and churn that real deployments often face.

One of these issues is very concrete and readily actionable. As stated in the results chapter (6), the simulator does not have enough memory to reconcile with the pairwise baselines at 64 replicas. As established in the threats, this is an artifact of the harness accumulating all neighbor's messages and not a property of the protocols themselves. Re running the matrix with a memory friendly dissemination strategy,for instance, processing incoming messages incrementally rather than buffering them would let the baselines complete at 64 replicas and probably beyond, separating infeasibility in this implementation from infeasibility in principle.

7.4 Closing remarks

This thesis set out to reconcile many diverging replicas without paying for the redundancy that naive pairwise extension incurs, and it arrived at a protocol it did not design but discovered. The more lasting result, however, is the method it used to get there. The autonomous loop, given only a simulator and a single objective, explored the design space it was confined to and returned an answer aware of its structure that a researcher committed to one elegant mechanism might never have reached. From the improvements seen during the agent's search, if one idea should be considered above all others, it is that the improvement of multi-party reconciliation communication efficiency does not only stand in better pairwise sketches in how information is allowed to propagate, and that an automated search, with the correct instructions and harness is a credible instrument for finding such improvements.

References

[1] Burton H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://dl.acm.org/doi/10.1145/362686.362692 (visited on 12/08/2025).

[2] Yevgeniy Dodis et al. “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data”. en. In: *SIAM Journal on Computing* 38.1 (Jan. 2008). arXiv:cs/0602007, pp. 97–139. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/060651380. URL: http://arxiv.org/abs/cs/0602007 (visited on 01/05/2026).

[3] Vitor Enes et al. *Efficient Synchronization of State-based CRDTs*. en. arXiv:1803.02750 [cs]. Mar. 2019. DOI: 10.48550/arXiv.1803.02750. URL: http://arxiv.org/abs/1803.02750 (visited on 12/28/2025).

[4] David Eppstein et al. “What’s the difference? efficient set reconciliation without prior context”. In: *Proceedings of the ACM SIGCOMM 2011 conference*. SIGCOMM ’11. New York, NY, USA: Association for Computing Machinery, Aug. 2011, pp. 218–229. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018462. URL: https://dl.acm.org/doi/10.1145/2018436.2018462 (visited on 12/08/2025).

[5] Pedro Silva Gomes and Carlos Baquero. *Rateless Bloom Filters: Set Reconciliation for Divergent Replicas with Variable-Sized Elements*. arXiv:2510.27614 [cs]. Oct. 2025. DOI: 10.48550/arXiv.2510.27614. URL: http://arxiv.org/abs/2510.27614 (visited on 01/16/2026).

[6] Pedro Silva Gomes, Miguel Boaventura Rodrigues, and Carlos Baquero. *ConflictSync: Bandwidth Efficient Synchronization of Divergent State*. arXiv:2505.01144 [cs]. May 2025. DOI: 10.48550/arXiv.2505.01144. URL: http://arxiv.org/abs/2505.01144 (visited on 01/04/2026).

[7] Long Gong et al. *Space- and Computationally-Efficient Set Reconciliation via Parity Bitmap Sketch (PBS)*. en. arXiv:2007.14569 [cs]. Aug. 2020. DOI: 10.48550/arXiv.2007.14569. URL: http://arxiv.org/abs/2007.14569 (visited on 01/08/2026).

[8] Michael T. Goodrich and Michael Mitzenmacher. “Invertible bloom lookup tables”. In: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. Sept. 2011, pp. 792–799. DOI: 10.1109/Allerton.2011.6120248. URL: https://ieeexplore.ieee.org/document/6120248 (visited on 12/08/2025).

[9] Ari Juels and Madhu Sudan. “A Fuzzy Vault Scheme”. en. In: *Designs, Codes and Cryptography* 38.2 (Feb. 2006), pp. 237–257. ISSN: 1573-7586. DOI: 10.1007/s10623-005-6343-z. URL: https://doi.org/10.1007/s10623-005-6343-z (visited on 01/16/2026).

[10] Y. Minsky, A. Trachtenberg, and R. Zippel. "Set reconciliation with nearly optimal communication complexity". In: *Proceedings. 2001 IEEE International Symposium on Information Theory*. June 2001, pp. 232–. DOI: 10.1109/ISIT.2001.936095. URL: https://ieeexplore.ieee.org/document/936095 (visited on 01/25/2026).

[11] Michael Mitzenmacher and Rasmus Pagh. "Simple multi-party set reconciliation". en. In: *Distributed Computing* 31.6 (Nov. 2018), pp. 441–453. ISSN: 1432-0452. DOI: 10.1007/s00446-017-0316-0. URL: https://doi.org/10.1007/s00446-017-0316-0 (visited on 01/03/2026).

[12] James K. Mullin. "Estimating the size of a relational join". In: *Information Systems* 18.3 (Apr. 1993), pp. 189–196. ISSN: 0306-4379. DOI: 10.1016/0306-4379(93)90037-2. URL: https://www.sciencedirect.com/science/article/pii/0306437993900372 (visited on 01/15/2026).

[13] A. Pinar Ozisik et al. "Graphene: efficient interactive set reconciliation applied to blockchain propagation". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 303–317. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342082. URL: https://dl.acm.org/doi/10.1145/3341302.3342082 (visited on 12/08/2025).

[14] D. Starobinski, A. Trachtenberg, and S. Agarwal. "Efficient PDA synchronization". In: *IEEE Transactions on Mobile Computing* 2.1 (Jan. 2003), pp. 40–51. ISSN: 1558-0660. DOI: 10.1109/TMC.2003.1195150. URL: https://ieeexplore.ieee.org/abstract/document/1195150 (visited on 01/25/2026).

[15] Lei Yang, Yossi Gilad, and Mohammad Alizadeh. "Practical Rateless Set Reconciliation". en. In: *Proceedings of the ACM SIGCOMM 2024 Conference*. arXiv:2402.02668 [cs]. Aug. 2024, pp. 595–612. DOI: 10.1145/3651890.3672219. URL: http://arxiv.org/abs/2402.02668 (visited on 12/08/2025).

Appendix A

Experimental Configuration

A.1 The evaluation matrix

Table A.1: Swept dimensions of the evaluation matrix. Their product defines the 18 cells; each cell is repeated over three seeds.

Dimension	Values	Count
Topology	Star, Tree, Chord	3
Jaccard similarity	0.25, 0.50, 0.75	3
Replica count N	16, 64	2
Cells ($3 \times 3 \times 2$)		18
Seeds	42, 43, 44	3
Total runs per protocol (18×3)		54

Table A.2: Workload parameters held fixed across every cell of the matrix.

Parameter	Value
Set size (elements per replica)	10 000
Payload size	32 B
Digest width	64 bit
Round cap	100
Universe size	500 000
Access pattern	Uniform
Zipf exponent	1.0
Workload seed	42

A.2 Configuration files

```toml
# Retrieval config: MultiReplica (the agent-discovered protocol, exp64 final
# state) over the full evaluation matrix. This is the headline result.
# Matrix identical across all configs/*.toml; only `protocol` differs.
protocol = "MultiReplica"
topologies = ["Star", "Tree", "Chord"]
seeds = [42, 43, 44]
jaccard_similarities = [0.25, 0.5, 0.75]
num_replicas = [16, 64]
round_cap = 100

[workload]
set_size = 10_000
payload_size = 32
digest_bits = 64
pattern = "Uniform"
universe_size = 500_000
zipf_exponent = 1.0
seed = 42
```

Listing A.1: Shared simulator configuration (`configs/multireplica.toml`, commit `b2acc19`). The remaining `configs/*.toml` files differ only in the `protocol` field.

Appendix B

The Autonomous Agent Loop

B.1 Agent instructions

The loop was driven by a single autonomous agent whose complete operating instructions are reproduced verbatim in Listing B.2. The agent was run with Claude (Sonnet) through the Claude Code CLI, launched once and left to iterate unattended.

```
CLAUDE_CODE_MAX_OUTPUT_TOKENS=100000 claude \
  --model sonnet \
  --dangerously-skip-permissions \
  -p "You are a bandwidth research agent. Read your instructions at \
      .claude/agents/bandwidth-researcher.md and start the experiment loop
          immediately."
```

Listing B.1: Invocation of the agent loop.

Listing B.3 shows the file the agent is to read before each attempt.

```
# Bandwidth Researcher Agent

You are an autonomous research agent. Your single task is to iteratively
improve the multi-replica set reconciliation protocol in
`src/simulator/protocols/multi_replica.rs` to minimise total bandwidth
while maintaining convergence. **Never stop experimenting.**

**Workload**: `set_size=10_000`, `universe_size=500_000`,
`pattern=Uniform`, `round_cap=100`, `seeds=[42,43,44]`, swept across
`num_replicas=[16, 64]` and `jaccard_similarities=[0.25, 0.5, 0.75]`.
Each `(topology, J, n)` triple is a "cell" in the evaluation matrix
(3 × 3 × 2 = 18 cells total). All fitness numbers below are measured
in this configuration -- they are not comparable with runs at a
different scale.

---

```

```
## Loop

**Before you start, read `hints.md` in the repo root.** It distils
lessons from prior agent runs at smaller scales: which architectures
worked, which optimisations helped, and -- critically -- which dead-end
strategies have already been tried and *must not be retried*. Re-read
it whenever you're stuck; it is your shortest path to a working
multi-scale protocol.

Repeat indefinitely:

1. **Read** `hints.md`, the current `multi_replica.rs`, and `experiments.tsv`.
2. **Hypothesise** a change that should reduce total bytes sent. Cross-check
   against `hints.md` -- if you're about to retry something the "do NOT retry"
   list rules out, pick a different hypothesis.
3. **Edit** `src/simulator/protocols/multi_replica.rs` (the only file you may edit).
4. **Commit** your change: `git add src/simulator/protocols/multi_replica.rs && git commit
    -m "<short description of what you tried>"`.
5. **Evaluate**: `cargo test 2>/dev/null && cargo run --release --bin eval -- --config
    agent.toml 2>/dev/null > /tmp/result.json`.
6. **Parse** the result: read `summary.fitness` and `summary.all_converged` from
    `/tmp/result.json`. Check `summary.by_cell[*].std_bytes` -- if any cell's `std_bytes`
    exceeds the improvement, the change is noise, not signal.
7. **Record** the result: **ALWAYS** append one line to `experiments.tsv`
   (see format below). **You MUST log every attempt -- both successes AND
   failures.** Failed attempts with `kept: no` are essential memory to
   avoid re-exploring dead ends. If build/tests fail, log fitness as
   `1e30` (a sentinel clearly out of band of any realistic value).
8. **Decide**:
   - If `all_converged` is true AND the scalar fitness improved, **keep** the commit.
   - Otherwise, **revert**: `git revert --no-edit HEAD`.
9. Go to step 1.

---

## Scalar fitness

```
fitness = geometric_mean(mean_bytes across all cells)
        = exp( (1/N) · Σ ln(mean_bytes[cell]) )
```

Where a cell = one `(topology, jaccard_similarity, num_replicas)` triple,
and each cell's `mean_bytes` is averaged across `seeds = [42, 43, 44]`.
Lower is better. Treat `all_converged == false` as fitness = infinity
(reject the attempt).

Why geometric mean, not sum. Summing would make large-n / low-J /
high-edge-count cells dominate the fitness by orders of magnitude -- a
10% improvement on Chord@n=64@J=0.25 would outweigh a 90% improvement
on Star@n=16@J=0.75. Geometric mean puts every cell on equal log-scale
footing: an X% reduction in *any* cell reduces fitness by roughly X/N%,
regardless of that cell's absolute byte count. This forces the agent to
improve broadly rather than exploiting the worst cell.

The numeric value is in byte-like units (it's `exp` of average log-bytes),
so "fitness 500M" means the typical cell transfers ~500M bytes.
```

````
71
72  ---
73
74  ## Experiment log
75
76  Append every attempt (success or revert) to `experiments.tsv` in this format:
77
78  ```
79  attempt\tfitness\tall_converged\tkept\tdescription
80  ```
81
82  The first line is the header (create the file with it if it doesn't exist).
83  `kept` is `yes` or `no`. `description` is a one-line summary of what you tried.
84
85  This log is your memory across iterations. Read it before each attempt to
86  avoid re-exploring dead ends.
87
88  ---
89
90  ## Current state
91
92  `multi_replica.rs` is the full-state-transfer scaffold, so its fitness
93  equals the FST baseline.
94
95  **Target to beat: `FullStateTransfer` fitness = 459,237,474 bytes**
96  (measured 2026-04-18 on the current config: `universe_size=500_000`,
97  `num_replicas=[16, 64]`, `J=[0.25, 0.5, 0.75]`, `seeds=[42, 43, 44]`,
98  all 18 cells converged).
99
100 The largest cells in absolute bytes are n=64 Tree/Chord at low J
101 (~3-5 GB each). Log-space, every cell contributes equally -- but those
102 large cells also have the most slack versus a sketch-based approach,
103 so reductions there tend to be both easy and high-impact.
104
105 To re-measure the baseline after any config change:
106
107 ```bash
108 sed 's/^protocol = "MultiReplica"/protocol = "FullStateTransfer"/' agent.toml \
109   > /tmp/baseline-FST.toml
110 ./target/release/eval --config /tmp/baseline-FST.toml \
111   | jq '.summary.fitness'
112 ```
113
114 ---
115
116 ## Constraints
117
118 1. **Only edit** `src/simulator/protocols/multi_replica.rs`. No other files
119    (except `experiments.tsv` for logging).
120 2. Must implement the `Protocol` trait from `src/simulator/protocols/mod.rs`.
121    The trait gives you `send_phase(&self, local: ReplicaView<'_>, topology,
122    outbox: &mut Outbox<'_>, carry)` and `recv_phase(&self, local, topology,
123    inbox: &mut [(usize, ProtocolMsg)]) -> ProtocolStepResult`.
124 3. To emit messages, call `Outbox::send_elements / send_riblt / send_bloom /
125    send_rateless_bloom / send_bloom_riblt / send_rateless_bloom_riblt`.
126    The `Outbox` is engine-bound to the current replica; you cannot supply
127    a different sender.
````

```
4. To read incoming messages, use the `ProtocolMsg` accessors --
   `take_elements()`, `as_riblt()`, `as_bloom()`, `as_rateless_bloom()`,
   `as_bloom_riblt()`, `as_rateless_bloom_riblt()`. There is no way to
   pattern-match the inner enum from outside `network.rs`.
5. Every field on every `ProtocolMsg` variant is billed by `WireSized`
   (`state_bytes` at send time, `metadata_bytes` after recv). There is no
   free data channel and no reconstruction shortcut.
6. The set returned in `ProtocolStepResult.next_set` must satisfy
   `next_set ⊇ local.snapshot_set()` -- protocols may only add elements,
   never remove them. The engine asserts this.
7. May use any algorithm in `src/simulator/algorithms/` (RIBLT sketches,
   Bloom filters, rateless Bloom filters, multiparty sketches).
8. May add internal state to `MultiReplicaProtocol`, but `Protocol`'s
   methods take `&self`, not `&mut self` -- per-replica state must live
   inside the carry that flows through `recv_phase` → engine → `send_phase`,
   not on `self`. `self` is shared across all replicas.
9. `cargo test` must pass after every edit.
10. All cells in the matrix must converge (`converged: true`).

---

## Ideas to explore

These are starting points, not an exhaustive list. Combine, modify, or
invent new approaches.

### Reducing metadata overhead
- **Digest-only pre-filter**: send only 8-byte digests instead of full
  elements in the first round, then send payloads only for confirmed-missing
  elements. Cost: 8 B/element vs 40 B/element.
- **Compressed Bloom filters**: the current BF is uncompressed. Run-length
  encoding or simple compression on sparse filters could shrink wire cost.
- **Smaller FPR for large sets**: adapt FPR based on set size -- large sets
  tolerate slightly higher FPR because the absolute number of false positives
  is still manageable.

### Reducing element transfer
- **Set difference estimation**: estimate |A \ B| before choosing a strategy.
  If the difference is small relative to set size, a digest exchange +
  targeted transfer beats a full Bloom filter.
- **RIBLT sketches**: use the RIBLT algorithm from
  `src/simulator/algorithms/riblt/` for set difference computation. RIBLT
  transmits O(|diff|) symbols regardless of set size -- optimal when the
  symmetric difference is small.
- **Hybrid BF + RIBLT**: use a Bloom filter to partition digests into
  "definitely absent" and "maybe present", then run RIBLT only on the
  ambiguous subset. See `src/simulator/protocols/hybrid_rbf_riblt.rs`.

### Topology-aware strategies
- **Gossip suppression**: on high-degree topologies (Chord), a node that

  has already received elements from multiple neighbours can suppress
  redundant sends to neighbours that likely already have them.
- **Hub awareness**: on Star topology, the hub sees all data first. If the
  hub broadcasts a summary of what it has, leaves can send only their unique
  elements.
- **Round-aware sizing**: in later rounds the symmetric difference shrinks.
```

```
  Adapt filter sizes or switch strategies based on round number.

### Multi-round optimisations
- **Exponential backoff on BF sends**: if a node's set hasn't grown much,
  send BFs less frequently (every 2nd or 3rd round) to avoid redundant
  filter overhead.
- **Cumulative sketches**: instead of rebuilding the full BF each round,
  send only a delta sketch covering newly added elements.

---

## Reading the codebase

Key files for understanding what's available:

- `hints.md` -- distilled strategy from prior runs (read this first)
- `src/simulator/protocols/mod.rs` -- `Protocol` trait, `ProtocolStepResult`,
    `LocalMetrics`, `PendingElements` (the carry type)
- `src/simulator/network.rs` -- `Outbox` (send path), `ProtocolMsg` and its accessors,
    `WireSized`, sealed message wrappers (`RibltMsg`, `BloomMsg`, `RatelessBloomMsg`, ...)
- `src/simulator/replica.rs` -- `Element` (digest: u64, payload: Vec<u8>), `Replica`,
    `ReplicaView` (the borrowed view passed to protocols)
- `src/simulator/topology.rs` -- `Topology`, `neighbors()`, `node_count()`, `kind`
- `src/simulator/algorithms/riblt/` -- RIBLT sketch implementation
- `src/simulator/algorithms/bloom.rs` -- Bloom filter (uses RandomState -- not directly
    serialisable, but the algorithm is reusable)
- `src/simulator/algorithms/rateless_bloom/` -- Rateless Bloom filter and stopping strategies
- `src/simulator/algorithms/multiparty_sketch.rs` -- finite-field coded sketches that can be
    added/subtracted along tree paths
- `src/simulator/protocols/riblt.rs` -- existing RIBLT protocol (reference implementation;
    uses the carry pattern)
- `src/simulator/protocols/hybrid_rbf_riblt.rs` -- existing hybrid protocol (reference)
- `src/simulator/protocols/full_state_transfer.rs` -- minimal reference; current
    `multi_replica.rs` is a copy of this
```

Listing B.2: Agent definition (`.claude/agents/bandwidth-researcher.md`, commit `b2acc19`) — the verbatim instructions the loop operated under.

```
# Strategy Hints for Bandwidth Researcher Agent

## Workload context

This is a **multi-scale evaluation matrix**: 3 topologies × 3 Jaccard
similarities × 2 node counts × 3 seeds = **54 simulation runs** per
experiment. The fitness is a **geometric mean** across 18 cells, so every
cell contributes equally in log-space.

| N  | J=0.25 (high divergence) | J=0.5 | J=0.75 (high similarity) |
|----|--------------------------|-------|--------------------------|
| 16 | ~3,333 unique/replica    | ~2,000| ~1,143 unique/replica    |
| 64 | ~3,333 unique/replica    | ~2,000| ~1,143 unique/replica    |

Key implications:
```

```
- **The protocol must work across ALL scales.** A strategy tuned for 16
  nodes must also work at 64, and vice versa.
- **J=0.25 means very large diffs** (~3,333 unique per replica). BF
  pre-filtering is highly valuable here.
- **J=0.75 means small diffs** (~1,143 unique). RIBLT alone may suffice.
- **64-node Chord** has power-of-2 links at distances 1,2,4,8,16,32 →
  6 neighbors per node, diameter ~6 hops.
- **64-node Tree** has depth ~6.
- **Do NOT hardcode node counts or topology sizes.** Use `topology.node_count()`,
  `topology.neighbors(id).len()`, etc.

---

## What we learned from prior runs (8-node and 32-node)

### Architecture that works across scales

**Star/Tree: single BF+RIBLT sketch + eager forwarding**
1. Round 1: each node sends a BF+RIBLT hybrid sketch to all neighbors
2. Sketch decoding identifies which elements each neighbor is missing
3. Send missing elements, then eager-forward all newly received elements
   to other neighbors (with `sent_to` dedup to prevent resending)
4. Star converges in 3-4 rounds, Tree in ~2×depth rounds

**Chord: pairwise BF+RIBLT with sequential distance-doubling**
1. Instead of sketching all neighbors every round, sketch only the pair
   at distance 2^power each round (power cycles 0, 1, 2, ..., log2(N)-1)
2. **Ascending distance order is essential**: reconcile nearby first so
   their elements are available when reconciling distant neighbors
3. No eager forwarding on Chord -- it causes cascade storms at ≥32 nodes

### Optimizations that helped

- **Asymmetric sketching**: only lower-id node sends the sketch. The
  higher-id node decodes both directions of the diff. Halves metadata.
- **BF-encoded requests**: instead of sending a list of 64-bit digests
  to request missing elements, encode them as a Bloom filter (~10
  bits/element vs 64 bits/element). Use low FPR (0.1%).
- **m_ratio = 1.0**: controls BF size (bits per element). Tested 0.5,
  0.7, 1.0, 1.3 -- the optimum is around 1.0 for 50% Jaccard. May
  need adaptation for J=0.25 and J=0.75.
- **Neighbor-has tracking**: skip eager-forwarding to neighbors known
  to already have the element (from sketch decode info).
- **Hub dedup (Star)**: when the hub needs an element, request it from
  only one leaf, not all leaves that have it.

### What does NOT work -- do NOT retry

- **Hash-based forwarder selection on Chord**: at 32 nodes, the
  designated forwarder may not have the element yet → fails to converge.
  Will be worse at 64 nodes.
- **Eager forwarding on Chord**: cascade storms at ≥32 nodes. State
  bytes double or worse. Tested twice, failed both times.
- **Source-neighbor suppression on Chord eager forwarding**: still
  cascades, doesn't solve the fundamental problem.
- **Descending distance order for Chord**: 77% worse than ascending.
  Close-first accumulation is essential.
```

```
- **Pure RIBLT (no BF) at J=0.5**: 10% worse than BF+RIBLT. BF
  pre-filtering is valuable when diffs are non-trivial.
- **Pure RIBLT for Chord powers 1-3**: diff still too large at early
  powers, bidirectional billing makes it more expensive than BF+RIBLT.

---

## Strategy recommendations for the full matrix

### Tier 1 -- Start here

#### 1. Topology-dispatched hybrid protocol
The proven architecture from prior runs:
```
if Chord:
    sequential distance-doubling with BF+RIBLT (ascending powers)
else: // Star, Tree
    single BF+RIBLT sketch round 1 + eager forwarding
```
Use `topology.kind` for dispatch but do NOT hardcode node counts.
The distance-doubling power cycle should go from 0 to
`floor(log2(node_count)) - 1` (not hardcoded `% 5`).

#### 2. Asymmetric sketching everywhere
Only lower-id node sends the sketch. Higher-id decodes both directions.
This works for all topologies and all scales.

#### 3. Adaptive m_ratio based on estimated diff size
- J=0.25 (large diffs): m_ratio=1.0 or higher -- BF savings are large
- J=0.75 (small diffs): lower m_ratio or even pure RIBLT -- BF overhead
  may exceed savings for small diffs
- The protocol doesn't know J directly, but it knows set_size and can
  estimate diff from sketch decode results in later rounds

### Tier 2 -- After the basics work

#### 4. Direct element transfer for tiny remaining diffs
After the main reconciliation phase, remaining diffs are often < 50
elements per pair. BF+RIBLT overhead (~10KB per sketch) exceeds the
cost of just sending 50 elements (2KB). Switch to direct transfer
in late rounds.

#### 5. BF-skip for converged pairs
If a node's set hasn't grown since its last sketch to a specific
neighbor, skip that sketch. Most useful in later rounds after
initial convergence.

#### 6. Tree: directed upward-downward flow
Instead of bidirectional sketch+forward on all edges:
- Upward phase: children sketch to parent only
- Downward phase: root pushes missing elements to children
May reduce Tree rounds and avoid redundant sibling forwarding.

### Tier 3 -- Advanced

#### 7. Multiparty sketch aggregation
`src/simulator/algorithms/multiparty_sketch.rs` implements finite-field
```

```
coded sketches with `add_sketch()` / `sub_sketch()` / `try_decode()`.
Enables aggregating multiple nodes' sketches along tree paths. Could
be much more efficient than pairwise reconciliation at 64 nodes.

---

## Geometric mean fitness -- practical implications

- An X% improvement in ANY cell reduces fitness by ~X/18%.
- The cells with worst absolute performance (Chord/N=64/J=0.25 at 5.6B)
  and best (Star/N=16/J=0.75 at 36M) contribute equally.
- **Don't neglect small cells.** A 50% improvement on Star/N=16/J=0.75
  (36M → 18M) has the same fitness impact as 50% on Chord/N=64/J=0.25
  (5.6B → 2.8B).
- **Check std_bytes**: if the standard deviation across seeds exceeds
  the improvement, the change is noise, not signal. Reject it.

## General principles

- **The protocol must generalize.** No hardcoded node counts, no
  topology-specific constants that break at different scales.
- **Rounds are cheap (cap=100), bytes are expensive.**
- **Log every attempt** (success AND failure) with `kept: yes/no`.
- **Check experiments.tsv before each attempt** to avoid retrying failures.
- **Test one thing at a time.** Isolate changes so you know what worked.
- **Each experiment runs 54 simulations** -- it will take longer than
  before. Be patient with evaluation time.
```

Listing B.3: Prior-run strategy hints (`hints.md`, commit `b2acc19`).

B.2 Search log

```
attempt fitness all_converged kept  description
1 1e30  false no  rateless BF+RIBLT multi-round: eval killed after 51min (too slow for N=64)
2 162799142 true  no  pure RIBLT multi-round, symmetric all-neighbor sketch (REVERTED but per rule should have been KEPT -- 65% under FST 459M baseline)
3 162726783 true  yes BF-only (fpr=0.01) multi-round all-neighbor sketch -- kept (improves vs FST 459M; ~matches reverted exp2 RIBLT 162.8M)
4 120781831 true  yes topology-dispatched: Chord pairwise distance-doubling BF asc, Star/Tree all-neighbor BF -- 26% under exp3 (Chord cells halved)
5 117116405 true  yes bump fpr 0.01->0.05 globally -- 3% gain net (Tree/Chord meta drops outweigh Star meta regression from doubled rounds)
6 115872074 true  yes per-topology fpr (Star=0.01 Chord/Tree=0.05) -- 1% gain restoring Star efficiency
7 113637073 true  yes bump tree fpr 0.05->0.1 -- 2% gain (smaller tree BFs outweigh extra rounds)
8 117452172 true  no  BF+RIBLT for Star/Tree -- 3% REGRESSION (rounds halved but state grew: exact decode sends every needed element with no FP-skipping, more duplicates)
9 113095466 true  yes bump chord fpr 0.05->0.1 -- 0.5% gain (near noise)
10  123136399 true  no  pure RIBLT for Chord pairwise -- 9% REGRESSION (Chord meta 3.7x bigger; RIBLT consumes ~1.5*diff symbols which exceeds BF cost even after ascending-order shrinkage)
```

```
11  114886075 true  no  tree fpr 0.1->0.07 -- 1.5% REGRESSION (confirms 0.1 sweet spot)
12  114508429 true  no  star fpr 0.01->0.02 -- 1.2% REGRESSION (confirms 0.01 sweet spot)
13  1e30  false no  tree 50% hash-suppression on sends (deterministic) -- DID NOT CONVERGE (Tree cells all hit round_cap=100, deterministic starvation when set.len() stabilizes)
14  1e30  false no  tree 50% RANDOM suppression -- STATE DROPPED 23% but still didn't converge in 100 rounds (multi-hop propagation too slow at 50% per round)
15  1e30  false no  tree 25% random suppression -- STILL no convergence (rounds=99.67 for J=0.25); accumulated drops over depth-6 tree exhaust 100-round cap
16  111894660 true  yes tree fpr 0.1->0.15 -- 1% gain (more FP-suppression cuts state slightly, smaller BFs offset extra rounds)
17  110480622 true  yes tree fpr 0.15->0.2 -- 1.3% gain (continuing FP-suppression sweet-spot search)
18  109655923 true  yes tree fpr 0.2->0.25 -- 0.7% gain (slowing but still improving)
19  1e30  false no  tree fpr 0.25->0.3 -- Tree J=0.25 N=64 seed 44 hit round_cap=100; fpr=0.3 too aggressive
20  109726247 true  no  chord fpr 0.1->0.15 -- flat (~noise); chord doesn't benefit from same FP-suppression as tree
21  109095690 true  yes tree fpr 0.25->0.27 -- 0.5% gain (still room above 0.25)
22  110099706 true  no  tree fpr 0.27->0.28 -- 1% REGRESSION; 0.27 is sweet spot
23  (restoration) - - exp23 was bookkeeping commit (restored tree=0.27 from broken 0.28)
24  109681291 true  no  chord fpr 0.1->0.07 -- 0.5% regression; chord 0.1 confirmed
25  109295136 true  no  star fpr 0.01->0.005 -- slight regression; star 0.01 confirmed
26  1e30  false no  adaptive tree fpr (0.27 base, 0.4 if set>50K) -- Tree N=64 all cells failed convergence (0.4 too aggressive in late rounds)
27  108483803 true  yes adaptive tree fpr (0.27 base, 0.32 if set>100K) -- 0.6% gain (gentler late-round bump works)
28  109732966 true  no  adaptive tree fpr threshold 100K -> 50K -- 1% REGRESSION (high fpr too early)
29  (restoration) - - exp29 was bookkeeping commit (kept fpr=0.32, threshold 100K)
30  1e30  false no  late tree fpr 0.32->0.35 (threshold 100K) -- Tree N=64 J=0.25 all seeds failed
31  109828613 true  no  late tree fpr 0.32->0.33 -- slight REGRESSION; 0.32 confirmed optimal late fpr
32  109265826 true  no  chord adaptive fpr (0.1 base, 0.15 if set>100K) -- slight regression; chord doesn't benefit
33  109055384 true  no  tree threshold 100K -> 150K -- slight regression; 100K confirmed best
34  110798523 true  no  star adaptive (0.01 base, 0.05 if set>100K) -- 2% regression; star prefers static low fpr
35  109438885 true  no  tree threshold 100K -> 120K -- slight regression; 100K confirmed optimal
36  109346409 true  no  late tree fpr 0.32->0.30 (threshold 100K) -- slight regression; 0.32 confirmed
37  1e30  false no  tree PAIRWISE edge-cycling -- dedup HUGE (state -30%, meta -68%) but rounds=100 (no convergence); pairwise needs more cycles than cap allows at fpr=0.27
38  1e30  false no  tree pairwise + fpr 0.05 -- still no convergence (slow propagation, all cells round_cap)
39  1e30  false no  tree pairwise + fpr 0.15 -- even N=16 doesn't converge; fundamental rounds limit on pairwise
40  1e30  false no  tree pairwise + fpr 0.10 -- same all hit round_cap=100
41  1e30  false no  tree skip-one with markers (only to skipped neighbour) -- broke lockstep (parent at idx 0 always skipped); reverted
42  1e30  false no  tree skip-one with markers to ALL -- still broke convergence; abandoned skip-one approach
43  108895114 true  yes revert all pairwise/skip-one; restored exp27 baseline (best 108.5M-108.9M range)
44  109915130 true  no  tree 3-level adaptive (0.27/0.32/0.34 at 100K/150K) -- regression; 2 levels enough
```

```
45  110045854 true  no  tree base 0.27->0.25 -- regression; 0.27 base confirmed
46  113227970 true  no  star hub-fpr 0.05 (leaves 0.01) -- 4% regression; rounds doubled (9->17) on Star, meta blew up
47  110461192 true  no  star hub adaptive 0.03 if hub_set>100K -- 1% regression
48  (restoration) - - exp48 was bookkeeping (revert star hub adaptive)
49  108964208 true  yes chord adaptive (0.1 base, 0.15 if set>100K) -- flat (~noise) but small win
50  108805876 true  yes chord adaptive late 0.15->0.13 -- flat (~noise)
51  108629249 true  yes tree adaptive 0.27/0.30 thresh 80K -- flat (~noise)
52  109725701 true  no  combo of tree (0.27/0.32 thresh 100K) + chord adaptive -- REGRESSION; exp51 settings preferred
53  (restoration) - - exp53 was bookkeeping (restored exp51)
54  109369434 true  no  tree late 0.30->0.29 -- slight regression
55  108593981 true  yes tree threshold 80K->70K (late 0.30) -- flat (noise)
56  110326574 true  no  tree threshold 70K->60K -- regression; 70-80K range optimal
57  (restoration) - - exp57 was bookkeeping (restored 70K)
58  109570116 true  no  tree late 0.30->0.31 (with thresh 70K) -- slight regression
59  (restoration) - - exp59 was bookkeeping (revert tree late to 0.30)
60  108676996 true  yes chord BF+RIBLT for late rounds (set>100K) -- ~flat (small improvement on chord cells)
61  110811291 true  no  chord BF+RIBLT threshold 100K -> 50K -- regression; 100K best
62  (restoration) - - exp62 was bookkeeping (chord BF+RIBLT threshold restored to 100K)
63  109080384 true  no  tree BF+RIBLT for late rounds (set>100K) -- slight regression; user-stopped here
```

Listing B.4: Complete experiment log (`experiments.tsv`, commit `b2acc19`). Fitness is in bytes; `1e30` is the sentinel for a build, test, or convergence failure.

B.3 Selected dead ends

Table B.1: Principal dead ends encountered during the search, grouped by failure mode. Attempt numbers refer to Listing B.4.

Attempts	Strategy tried	Why it failed
1	Rateless BF+RIBLT, all-neighbour sketch	Decoding too slow at $N = 64$; the evaluation was killed after 51 minutes. A throughput ceiling, not a correctness failure.
13–15	Tree send-suppression (deterministic and random, 25–50%)	Dropping sends to save bytes starves multi-hop propagation; the depth-6 tree exhausts the 100-round cap before converging.
19, 26, 30	Over-aggressive Tree false-positive rate (0.30–0.40, including adaptive late-round bumps)	Bloom filters become so lossy that the largest, most-divergent Tree cells ($N = 64$, $J = 0.25$) never converge within the cap.
37–40	Tree pairwise edge-cycling (one neighbour sketched per round)	Excellent deduplication (state -30%, metadata -68%) but requires more than 100 rounds to propagate; even $N = 16$ fails to converge.
41–42	Tree skip-one with coordination markers	Breaks the lockstep alignment between neighbours, so the protocol stops converging.

Appendix C

Protocol Source Listings

C.1 Best baseline: `HybridRbfRiblt`

```rust
use std::collections::HashMap;
use std::time::Duration;

use
    crate::simulator::algorithms::rateless_bloom::bayesian_cost::RATELESS_SET_RECONCILIATION_OVER
use
    crate::simulator::algorithms::rateless_bloom::expected_cost::ExpectedCostFactory;
use crate::simulator::algorithms::rateless_bloom::{RatelessBF,
    StoppingStrategyFactory};
use crate::simulator::algorithms::riblt::RatelessIBLT;
use crate::simulator::network::{ProtocolMsg, Outbox};
use crate::simulator::protocols::{
    LocalMetrics, PendingElements, Protocol, ProtocolKind, ProtocolStepResult,
};
use crate::simulator::replica::{Element, ReplicaView};
use crate::simulator::topology::Topology;

/// Two-round Hybrid Rateless-Bloom + RIBLT reconciliation.
///
/// Round N: send_phase ships `(RatelessBF, RIBLT)` pairs; recv_phase
/// runs the rateless-bloom stopping strategy + RIBLT cross-check and
/// returns the per-neighbour local-only elements as `carry`.
/// Round N+1: send_phase consumes the carry and emits Elements.
#[derive(Clone, Debug)]
pub struct HybridRbfRibltProtocol {
    m_ratio: f64,
}

impl Default for HybridRbfRibltProtocol {
    fn default() -> Self {
        Self::new()
```

```rust
    }
}

impl HybridRbfRibltProtocol {
    pub fn new() -> Self {
        Self { m_ratio: 0.5 }
    }

    pub fn with_m_ratio(m_ratio: f64) -> Self {
        assert!(m_ratio > 0.0, "m_ratio must be positive");
        Self { m_ratio }
    }

    fn bloom_bits_for(&self, n: usize) -> usize {
        let requested = ((n as f64) * self.m_ratio).ceil().max(1.0) as usize;
        let minimum_for_nonzero_threshold = RATELESS_SET_RECONCILIATION_OVERHEAD *
            8;
        requested.max(minimum_for_nonzero_threshold)
    }

    fn effective_m_ratio(bloom_bits: usize, n: usize) -> f64 {
        bloom_bits as f64 / n.max(1) as f64
    }
}

impl Protocol for HybridRbfRibltProtocol {
    fn kind(&self) -> ProtocolKind {
        ProtocolKind::HybridRbfRiblt
    }

    fn send_phase(
        &self,
        local: ReplicaView<'_>,
        topology: &Topology,
        outbox: &mut Outbox<'_>,
        carry: Option<PendingElements>,
    ) {
        let pending = carry.unwrap_or_default();

        if pending.is_empty() {
            let digests: Vec<u64> = local.set.iter().map(|e| e.digest).collect();
            let bloom_bits = self.bloom_bits_for(digests.len());

            for &neighbor_id in topology.neighbors(local.id) {
                let bf = RatelessBF::new(digests.clone(), bloom_bits);
                let riblt = RatelessIBLT::riblt_from(digests.iter().copied());
                outbox.send_rateless_bloom_riblt(neighbor_id, bf, riblt);
            }
        } else {
```

```
            for (neighbor_id, elements) in pending {
                if !elements.is_empty() {
                    outbox.send_elements(neighbor_id, elements);
                }
            }
        }
    }

    fn recv_phase(
        &self,
        local: ReplicaView<'_>,
        _topology: &Topology,
        inbox: &mut [(usize, ProtocolMsg)],
    ) -> ProtocolStepResult {
        let mut next_set = local.snapshot_set();

        let mut encode_time = Duration::ZERO;
        let mut decode_time = Duration::ZERO;

        let mut pending: PendingElements = HashMap::new();

        for (from, msg) in inbox.iter_mut() {
            if let Some(els) = msg.take_elements() {
                for element in els {
                    next_set.insert(element);
                }
                continue;
            }

            if let Some((sender_filter, sender_riblt)) =
                msg.as_rateless_bloom_riblt() {
                let bloom_bits = sender_filter.bits_per_filter();
                let sender_size = sender_filter.source_size();
                let effective_m_ratio = Self::effective_m_ratio(bloom_bits,
                    sender_size);

                let local_digests: Vec<u64> = local.set.iter().map(|e|
                    e.digest).collect();

                let stopping_strategy = ExpectedCostFactory::new(effective_m_ratio)
                    .create(local_digests, sender_size);

                let (common, definitely_missing) =
                    sender_filter.extend_until(stopping_strategy);

                encode_time += sender_filter.t_enc();
                decode_time += sender_filter.t_dec();

                let mut recovered_local_only: Vec<u64> = definitely_missing;
```

```
123
124                 if !common.is_empty() {
125                     let mut common_riblt = RatelessIBLT::riblt_from(common);
126
127                     sender_riblt.decode_against(&mut common_riblt);
128                     encode_time += sender_riblt.t_enc();
129                     decode_time += sender_riblt.t_dec();
130
131                     recovered_local_only.extend(sender_riblt.remote_only());
132                 }
133
134                 let local_only_set: std::collections::HashSet<u64> =
135                     recovered_local_only.into_iter().collect();
136                 let to_send: Vec<Element> = local
137                     .set
138                     .iter()
139                     .filter(|e| local_only_set.contains(&e.digest))
140                     .cloned()
141                     .collect();
142                 if !to_send.is_empty() {
143                     pending.entry(*from).or_default().extend(to_send);
144                 }
145             }
146         }
147
148         ProtocolStepResult {
149             next_set,
150             metrics: LocalMetrics {
151                 encode_time,
152                 decode_time,
153             },
154             carry: if pending.is_empty() { None } else { Some(pending) },
155         }
156     }
157 }
158
159 #[cfg(test)]
160 mod tests {
161     use super::*;
162     use crate::simulator::network::Network;
163     use crate::simulator::protocols::test_helpers::make_replica;
164     use std::collections::HashSet;
165
166     #[test]
167     fn hybrid_converges_in_two_rounds() {
168         let protocol = HybridRbfRibltProtocol::new();
169         let topology = Topology::star(2);
170         let mut replicas = vec![make_replica(0, &[1, 2, 3]), make_replica(1, &[2,
            3, 4])];
```

```
        let mut network: Network<ProtocolMsg> = Network::from_topology(&topology);
        let mut carries: Vec<Option<PendingElements>> =
            (0..replicas.len()).map(|_| None).collect();

        for _ in 0..2 {
            for id in 0..replicas.len() {
                let carry = carries[id].take();
                let mut outbox = Outbox::for_replica(&mut network, id);
                protocol.send_phase(replicas[id].view(), &topology, &mut outbox,
                    carry);
            }
            let mut inboxes: Vec<Vec<(usize, ProtocolMsg)>> = (0..replicas.len())
                .map(|id| network.drain_inbox(id))
                .collect();
            let results: Vec<_> = inboxes
                .iter_mut()
                .enumerate()
                .map(|(id, inbox)| {
                    protocol.recv_phase(replicas[id].view(), &topology, inbox)
                })
                .collect();
            for inbox in &inboxes {
                network.bill_metadata_post_recv(inbox);
            }
            for (id, (replica, result)) in
                replicas.iter_mut().zip(results).enumerate()
            {
                replica.set = result.next_set;
                carries[id] = result.carry;
            }
        }

        let union0: HashSet<u64> = replicas[0].set.iter().map(|e|
            e.digest).collect();
        let union1: HashSet<u64> = replicas[1].set.iter().map(|e|
            e.digest).collect();
        assert!(union0.contains(&1));
        assert!(union0.contains(&2));
        assert!(union0.contains(&3));
        assert!(union0.contains(&4));
        assert_eq!(union0, union1);
        assert!(network.stats().bytes_metadata > 0);
    }

    #[test]
    fn hybrid_identical_sets_no_elements_sent() {
        let protocol = HybridRbfRibltProtocol::new();
        let topology = Topology::star(2);
```

```
216         let replicas = vec![make_replica(0, &[5, 6, 7]), make_replica(1, &[5, 6,
                7])];
217         let mut network: Network<ProtocolMsg> = Network::from_topology(&topology);
218         let mut carries: Vec<Option<PendingElements>> =
219             (0..replicas.len()).map(|_| None).collect();
220
221         for id in 0..replicas.len() {
222             let carry = carries[id].take();
223             let mut outbox = Outbox::for_replica(&mut network, id);
224             protocol.send_phase(replicas[id].view(), &topology, &mut outbox,
                    carry);
225         }
226         let mut inboxes: Vec<Vec<(usize, ProtocolMsg)>> = (0..replicas.len())
227             .map(|id| network.drain_inbox(id))
228             .collect();
229         for (id, inbox) in inboxes.iter_mut().enumerate() {
230             let result = protocol.recv_phase(replicas[id].view(), &topology,
                    inbox);
231             carries[id] = result.carry;
232         }
233         for inbox in &inboxes {
234             network.bill_metadata_post_recv(inbox);
235         }
236
237         // With identical sets, the rateless-bloom + RIBLT pipeline finds
238         // zero local-only elements, so no carry is produced.
239         assert!(carries.iter().all(|c| c.is_none()));
240
241         network.reset();
242         for id in 0..replicas.len() {
243             let carry = carries[id].take();
244             let mut outbox = Outbox::for_replica(&mut network, id);
245             protocol.send_phase(replicas[id].view(), &topology, &mut outbox,
                    carry);
246         }
247         for id in 0..replicas.len() {
248             for (_from, mut msg) in network.drain_inbox(id) {
249                 assert!(msg.as_rateless_bloom_riblt().is_some());
250             }
251         }
252     }
253 }
```

Listing C.1: `src/simulator/protocols/hybrid_rbf_riblt.rs` (commit `b2acc19`).

C.2 Discovered protocol: `MultiReplica`

```rust
use std::collections::HashMap;

use crate::simulator::algorithms::bloom::BloomFilter;
use crate::simulator::network::{ProtocolMsg, Outbox};
use crate::simulator::protocols::{
    LocalMetrics, PendingElements, Protocol, ProtocolKind, ProtocolStepResult,
};
use crate::simulator::replica::{Element, ReplicaView};
use crate::simulator::topology::{Topology, TopologyKind};

/// Sentinel key for the chord power counter. `usize::MAX` is never a valid
/// neighbour id, so this slot is invisible to the delivery loop in send_phase.
const CHORD_POWER_KEY: usize = usize::MAX;

fn is_sentinel_key(k: usize) -> bool {
    k == CHORD_POWER_KEY
}

/// Topology-dispatched reconciliation.
///
/// * Star: each node sketches its current set to every neighbour per sketch
///   round. Receivers identify the elements the sender lacks and send them
///   back as Elements in the next delivery round.
/// * Chord: pairwise distance-doubling. Each sketch round, every node
///   sketches only the neighbours at distance 2^p (forward and backward)
///   where p cycles 0..floor(log2(N)). Ascending order ensures nearby
///   elements are absorbed before distant pairs reconcile.
/// * Tree: each node sketches its current set to every neighbour per sketch
///   round, identically to Star. Multi-source duplicate sends are tolerated
///   and later suppressed by the adaptive Tree FPR (see `fpr`), which rises
///   once the local set is large so redundant elements that already arrived
///   via another path are filtered out.
///
/// Per-replica state (the chord power counter) lives under a sentinel key in
/// the carry, invisible to the delivery loop in `send_phase`.
pub struct MultiReplicaProtocol;

impl MultiReplicaProtocol {
    pub fn new() -> Self {
        Self
    }

    /// FPR per topology. Star converges in ~few rounds, so a low FPR keeps
    /// it from doubling its round count. Chord/Tree run many rounds anyway,
    /// so a higher FPR shrinks each BF more than it adds rounds.
    /// Tree is also adaptive: once the local set is large (late rounds, near
    /// union), bump FPR -- the BF is then much smaller and most queries are
    /// FPs anyway, so the extra suppression mostly cancels the elements that
    /// already arrived from another path (multi-source dedup).
```

```rust
    fn fpr(kind: TopologyKind, set_len: usize, sender_id: usize) -> f64 {
        match kind {
            TopologyKind::Star => {
                // Hub (id=0) has the largest set in late rounds -- its BF
                // dominates Star metadata. Use a higher FPR for the hub
                // ONLY when its set is large (post-gather), so its BF
                // shrinks without inflating early-round Star round count.
                let _ = (sender_id, set_len);
                0.01
            }
            TopologyKind::Chord => {
                let _ = sender_id;
                if set_len > 100_000 {
                    0.13
                } else {
                    0.1
                }
            }
            TopologyKind::Tree => {
                if set_len > 70_000 {
                    0.30
                } else {
                    0.27
                }
            }
        }
    }

    fn build_bf(
        &self,
        set: &std::collections::HashSet<Element>,
        kind: TopologyKind,
        sender_id: usize,
    ) -> BloomFilter<u64> {
        let n = set.len().max(1);
        let mut bf: BloomFilter<u64> = BloomFilter::new(n, Self::fpr(kind,
            set.len(), sender_id));
        for e in set.iter() {
            bf.insert(&e.digest);
        }
        bf
    }

    fn read_chord_power(carry: &PendingElements) -> usize {
        carry
            .get(&CHORD_POWER_KEY)
            .and_then(|v| v.first())
            .map(|e| e.digest as usize)
            .unwrap_or(0)
```

```
    }

    fn write_chord_power(carry: &mut PendingElements, power: usize) {
        carry.insert(
            CHORD_POWER_KEY,
            vec![Element {
                digest: power as u64,
                payload: Vec::new(),
            }],
        );
    }

    fn chord_powers(node_count: usize) -> usize {
        if node_count <= 1 {
            return 1;
        }
        ((node_count as f64).log2().floor() as usize).max(1)
    }

    fn chord_distance_power(node_count: usize, local: usize, other: usize) ->
        Option<usize> {
        if node_count <= 1 {
            return None;
        }
        let fwd = (other + node_count - local) % node_count;
        let bwd = (local + node_count - other) % node_count;
        let dist = fwd.min(bwd);
        if dist > 0 && dist.is_power_of_two() {
            Some(dist.trailing_zeros() as usize)
        } else {
            None
        }
    }
}

impl Protocol for MultiReplicaProtocol {
    fn kind(&self) -> ProtocolKind {
        ProtocolKind::MultiReplica
    }

    fn send_phase(
        &self,
        local: ReplicaView<'_>,
        topology: &Topology,
        outbox: &mut Outbox<'_>,
        carry: Option<PendingElements>,
    ) {
        let pending = carry.unwrap_or_default();
        let chord_power = Self::read_chord_power(&pending);
```

```
        let has_real_pending = pending
            .iter()
            .any(|(k, v)| !is_sentinel_key(*k) && !v.is_empty());

        if has_real_pending {
            for (nbr, elements) in pending {
                if is_sentinel_key(nbr) {
                    continue;
                }
                if elements.is_empty() {
                    continue;
                }
                outbox.send_elements(nbr, elements);
            }
            return;
        }

        match topology.kind {
            TopologyKind::Chord => {
                let total = topology.node_count();
                if total <= 1 {
                    return;
                }
                let l = Self::chord_powers(total);
                let p = chord_power % l;
                let dist = 1usize << p;
                let fwd = (local.id + dist) % total;
                let bwd = (local.id + total - dist) % total;
                // Late rounds (large set): pair diffs are tiny (FP residuals).
                // Pair BF+RIBLT with RIBLT decoding away the FP residuals,
                // saving the extra cycles needed for FP-only cleanup.
                let use_riblt = local.set.len() > 100_000;
                let mut send = |target: usize| {
                    let bf = self.build_bf(local.set, topology.kind, local.id);
                    if use_riblt {
                        let digests: Vec<u64> = local.set.iter().map(|e|
                            e.digest).collect();
                        let riblt =
                            crate::simulator::algorithms::riblt::RatelessIBLT::riblt_from(
                                digests.iter().copied(),
                            );
                        outbox.send_bloom_riblt(target, bf, riblt);
                    } else {
                        outbox.send_bloom(target, bf);
                    }
                };
                send(fwd);
                if bwd != fwd && bwd != local.id {
                    send(bwd);
```

```
                }
            }
            TopologyKind::Tree => {
                for &nbr in topology.neighbors(local.id) {
                    outbox.send_bloom(nbr, self.build_bf(local.set, topology.kind,
                        local.id));
                }
            }
            _ => {
                for &nbr in topology.neighbors(local.id) {
                    outbox.send_bloom(nbr, self.build_bf(local.set, topology.kind,
                        local.id));
                }
            }
        }
    }

    fn recv_phase(
        &self,
        local: ReplicaView<'_>,
        topology: &Topology,
        inbox: &mut [(usize, ProtocolMsg)],
    ) -> ProtocolStepResult {
        let mut next_set = local.snapshot_set();
        let mut pending: PendingElements = HashMap::new();
        let mut saw_sketch = false;
        let mut saw_elements = false;
        let mut observed_power: Option<usize> = None;
        let total = topology.node_count();
        let is_chord = topology.kind == TopologyKind::Chord;

        for (from, msg) in inbox.iter_mut() {
            if let Some(els) = msg.take_elements() {
                for e in els {
                    next_set.insert(e);
                }
                saw_elements = true;
                if is_chord {
                    if let Some(p) = Self::chord_distance_power(total, local.id,
                        *from) {
                        observed_power = Some(p);
                    }
                }
                continue;
            }

            if let Some(bf) = msg.as_bloom() {
                saw_sketch = true;
                let to_send: Vec<Element> = local
```

```
                    .set
                    .iter()
                    .filter(|e| !bf.contains(&e.digest))
                    .cloned()
                    .collect();
                if !to_send.is_empty() {
                    pending.entry(*from).or_default().extend(to_send);
                }
                if is_chord {
                    if let Some(p) = Self::chord_distance_power(total, local.id,
                        *from) {
                        observed_power = Some(p);
                    }
                }
                continue;
            }

            if let Some((bf, riblt)) = msg.as_bloom_riblt() {
                saw_sketch = true;
                let mut definitely_missing: Vec<Element> = Vec::new();
                let mut common_digests: Vec<u64> = Vec::new();
                for e in local.set.iter() {
                    if bf.contains(&e.digest) {
                        common_digests.push(e.digest);
                    } else {
                        definitely_missing.push(e.clone());
                    }
                }

                let mut fp_corrections: std::collections::HashSet<u64> =
                    std::collections::HashSet::new();
                if !common_digests.is_empty() {
                    let mut common_riblt:
                        crate::simulator::algorithms::riblt::RatelessIBLT<u64> =
                        crate::simulator::algorithms::riblt::RatelessIBLT::riblt_from(
                            common_digests.iter().copied(),
                        );
                    riblt.decode_against(&mut common_riblt);
                    fp_corrections.extend(riblt.remote_only());
                }

                let mut to_send: Vec<Element> = definitely_missing;
                if !fp_corrections.is_empty() {
                    for e in local.set.iter() {
                        if fp_corrections.contains(&e.digest) {
                            to_send.push(e.clone());
                        }
                    }
                }
```

```
287
288                 if !to_send.is_empty() {
289                     pending.entry(*from).or_default().extend(to_send);
290                 }
291                 if is_chord {
292                     if let Some(p) = Self::chord_distance_power(total, local.id,
                            *from) {
293                         observed_power = Some(p);
294                     }
295                 }
296             }
297         }
298
299         if is_chord && total > 1 {
300             let l = Self::chord_powers(total);
301             let next_power = match (observed_power, saw_sketch, saw_elements) {
302                 (Some(p), true, false) => p,            // sketch received →
                        deliver next at same p
303                 (Some(p), false, true) => (p + 1) % l,  // delivery received →
                        advance to next p
304                 (Some(p), _, _) => (p + 1) % l,         // mixed (rare) → advance
                        to be safe
305                 (None, _, _) => 0,                       // empty inbox → restart
306             };
307             Self::write_chord_power(&mut pending, next_power);
308         }
309
310         ProtocolStepResult {
311             next_set,
312             metrics: LocalMetrics::default(),
313             carry: if pending.is_empty() { None } else { Some(pending) },
314         }
315     }
316 }
```

Listing C.2: `src/simulator/protocols/multi_replica.rs` (commit `b2acc19`).