

2023

# Synchronization of data in heterogeneous decentralized systems

---

<https://hdl.handle.net/2144/46286>

*"Downloaded from OpenBU. Boston University's institutional repository."*

BOSTON UNIVERSITY  
COLLEGE OF ENGINEERING

Dissertation

**SYNCHRONIZATION OF DATA IN HETEROGENEOUS  
DECENTRALIZED SYSTEMS**

by

**NOVAK BOŠKOV**

B.S., University of Novi Sad, Serbia, 2015  
M.S., University of Novi Sad, Serbia, 2016

Submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

2023

© 2023 by  
NOVAK BOŠKOV  
All rights reserved

## Approved by

First Reader

---

Ari Trachtenberg, PhD  
Professor of Electrical and Computer Engineering  
Professor of Systems Engineering

Second Reader

---

Alan Liu, PhD  
Assistant Professor of Electrical and Computer Engineering  
Assistant Professor of Computer Science

Third Reader

---

David Starobinski, PhD  
Professor of Electrical and Computer Engineering  
Professor of Systems Engineering

Fourth Reader

---

Gianluca Stringhini, PhD  
Assistant Professor of Electrical and Computer Engineering

*To Ida and Nera*

*Excuse me, sir. Can you tell me where Chestnut Street is? You don't know? But it's got to be here somewhere. Maybe I have the name wrong, but I know for a fact it's a street lined with chestnut trees. What's that? There isn't any such street? Oh, but there must be, sir. Memories can't possibly be so misleading.*

Danilo Kiš, "Early Sorrows (For Children and Sensitive Readers)"

## Acknowledgments

I would like to express my sincere gratitude to my advisor, Prof. Ari Trachtenberg, who inspired me to challenge myself and motivated me to persist. His outstanding support was invaluable to my academic and personal development alike. Furthermore, I would like to thank Prof. David Starobinski for his meticulous suggestions and careful guidance. Likewise, I owe gratitude to Prof. Alan Liu and Prof. Gianluca Stringhini for their valuable time and constructive feedback throughout the dissertation writing process. I acknowledge the National Science Foundation and the Boston University Red Hat Collaboratory for partially supporting this research.

I thank Muhammad Anas Imtiaz, Johannes K Becker, Jonathan Chamberlain, Nataša Trkulja, Şevval Şimşek, Stefan Gvozdenović, Trishita Tiwari, and Zhenpeng Shi for the privilege of being their lab mate. You have all had your distinct and irreplaceable roles in this journey.

I am particularly grateful to my friends Rachel, Ada, Burcu, Mertcan, and Mert. To my sister Katarina, my mother Svetlana, my late father Miloš, and my wife Ida, I owe more than words of gratitude.

# SYNCHRONIZATION OF DATA IN HETEROGENEOUS DECENTRALIZED SYSTEMS

NOVAK BOŠKOV

Boston University, College of Engineering, 2023

Major Professor: Ari Trachtenberg, PhD  
Professor of Electrical and Computer Engineering  
Professor of Systems Engineering

## ABSTRACT

Data synchronization is the problem of reconciling the differences between large data stores that differ in a small number of records. It is a common thread among disparate distributed systems ranging from fleets of Internet of Things (IoT) devices to clusters of distributed databases in the cloud. Most recently, data synchronization has arisen in globally distributed public blockchains that build the basis for the envisioned decentralized Internet of the future. Moreover, the parallel development of edge computing has significantly increased the heterogeneity of networks and computing devices. The merger of highly heterogeneous system resources and the decentralized nature of future Internet applications calls for a new approach to data synchronization. In this dissertation, we look at the problem of data synchronization through the prism of set reconciliation and introduce novel tools and protocols that improve the performance of data synchronization in heterogeneous decentralized systems.

First, we compare the analytical properties of the state-of-the-art set reconciliation protocols, and investigate the impact of theoretical assumptions and implementation

decisions on the synchronization performance. Second, we introduce *GenSync*, the first unified set reconciliation middleware. Using *GenSync*'s distinctive benchmarking layer, we find that the best protocol choice is highly sensitive to the system conditions, and a bad protocol choice causes a severe hit in performance. We showcase the evaluative power of *GenSync* in one of the world's largest wireless network emulators, and demonstrate choosing the best *GenSync* protocol under a high and low user mobility in an emulated cellular network. Finally, we introduce *SREP* (*Set Reconciliation-Enhanced Propagation*), a novel blockchain transaction pool synchronization protocol with quantifiable guarantees. Through simulations, we show that *SREP* incurs significantly smaller bandwidth overhead than a similar approach from the literature, especially in the networks of realistic sizes (tens of thousands of participants).

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Blockchain Example . . . . .	2
1.2	Data Synchronization in Traditional Distributed Systems . . . . .	3
1.3	Cloud/Edge Convergence . . . . .	4
1.4	Research Questions . . . . .	6
1.5	Contributions . . . . .	7
1.6	Publications . . . . .	11
1.7	Outline . . . . .	12
<b>2</b>	<b>Background and Related Work</b>	<b>13</b>
2.1	Set Reconciliation . . . . .	13
2.1.1	Definitions . . . . .	13
2.1.2	Assumptions . . . . .	15
2.1.3	Set Reconciliation as a Data Synchronization Approach . . . . .	16
2.2	Set Reconciliation Protocols in the Literature . . . . .	16
2.3	Approximate Set Membership Data Structures . . . . .	18
2.3.1	Invertible Bloom Filter . . . . .	18
2.3.2	Cuckoo Filter . . . . .	21
2.4	Blockchain Peer-to-Peer Network . . . . .	23
2.4.1	Transaction Dissemination and Memory Pools . . . . .	24
2.4.2	Block Propagation . . . . .	26
2.4.3	Topology of Popular Blockchain Networks . . . . .	27

<b>3</b>	<b>End-to-End Synchronization</b>	<b>29</b>
3.1	The Structure of the State-of-the-Art Protocols . . . . .	30
3.1.1	Invertible Bloom Lookup Table-based Set Reconciliation . . . . .	30
3.1.2	Cuckoo Filter-based Set Reconciliation . . . . .	31
3.1.3	Characteristic Polynomial Interpolation-based Set Reconciliation . . . . .	33
3.1.4	Protocol Extensions . . . . .	34
3.2	Analytical Performance Comparison . . . . .	36
3.3	Practical Implications of Protocol Assumptions . . . . .	38
3.3.1	Symmetric Difference Upper Bound . . . . .	38
3.3.2	Interactive Characteristic Polynomial Interpolation . . . . .	41
3.3.3	Characteristic Polynomial Interpolation with Priorities . . . . .	42
3.3.4	“Peeling” Process in Invertible Bloom Filters . . . . .	46
3.3.5	Set Reconciliation Protocol Symmetry . . . . .	48
3.3.6	Hash Function Implementation for Cuckoo Hashing . . . . .	50
<b>4</b>	<b>Network-Wide Synchronization</b>	<b>59</b>
4.1	<i>SREP</i> Algorithm . . . . .	60
4.2	<i>SREP</i> Performance Analysis . . . . .	64
4.2.1	Network Model . . . . .	65
4.2.2	Elementary <i>SREP</i> ( <i>E-SREP</i> ) . . . . .	67
4.2.3	Elementary Parallel <i>SREP</i> ( <i>EP-SREP</i> ) . . . . .	68
4.2.4	Multi-element <i>SREP</i> . . . . .	72
4.3	Simulations . . . . .	75
4.3.1	Configuring Network Model Parameters . . . . .	75
4.3.2	<i>SREP</i> Properties Validation . . . . .	77
4.3.3	Comparison with <i>MempoolSync</i> . . . . .	80
4.3.4	Communication Cost in Large-Scale Networks . . . . .	83

4.4	Summary . . . . .	83
4.5	Discussion . . . . .	85
<b>5</b>	<b><i>GenSync</i> Framework</b>	<b>87</b>
5.1	Core Abstractions . . . . .	89
5.2	<i>GenSync</i> Testbed . . . . .	94
5.3	Experimental Performance Comparison . . . . .	96
5.3.1	Impact of Differences Size . . . . .	99
5.3.2	Impact of Data Cardinality . . . . .	102
5.3.3	Impact of System Parameters . . . . .	104
5.3.4	Impact of Very Low Bandwidth . . . . .	107
5.4	Benefits of Interactive Protocols . . . . .	110
5.5	Bitcoin Data Set Evaluation . . . . .	113
5.6	Case Study — Synchronization at the Edge . . . . .	118
5.6.1	Colosseum Scenarios . . . . .	119
5.6.2	Measuring Transport-layer Variations . . . . .	120
5.6.3	Emulation Results and Takeaways . . . . .	122
<b>6</b>	<b>Conclusion and Future Work</b>	<b>124</b>
6.1	Contributions Summary . . . . .	125
6.2	Future Directions . . . . .	128
<b>A</b>	<b><i>SREP</i></b>	<b>131</b>
A.1	Redundant Transmissions . . . . .	131
A.2	On The Counting Argument From Theorem 2 . . . . .	131
A.3	The “Triangle Problem” . . . . .	133
A.4	On Rejecting $(M, s)$ Pairs . . . . .	135
A.5	On Constructing Pools Assignment from Set Sizes . . . . .	136

B <i>GenSync</i> Abstractions Detailed	138
Bibliography	140
Curriculum Vitae	161

# List of Tables

2.1	Set reconciliation literature summary. <b>BF</b> <i>abbr.</i> Bloom filter. <b>CF</b> <i>abbr.</i> Cuckoo filter. <b>IBLT</b> <i>abbr.</i> Invertible Bloom Lookup Table. <b>ASM</b> <i>abbr.</i> Approximate Set Membership. <b>CPI</b> <i>abbr.</i> Characteristic Polynomial Interpolation. <b>BCH</b> <i>abbr.</i> Bose-Chaudhuri-Hocquenghem (codes). <b>ECC</b> <i>abbr.</i> Error-Correcting Codes. . . . .	17
2.2	Cuckoo filter notation. . . . .	22
3.1	Common set reconciliation approaches and their suitability for the variations of the set reconciliation problem. Some approaches support certain variations only via separately proposed techniques (reference here). . . . .	35
3.2	Asymptotic communication and computation complexities for common set reconciliation approaches. $d$ is the actual number of mutual differences between $S_A$ and $S_B$ . $b$ is the constant size of set elements. $\bar{m}, k, \eta$ are constants known in advance. . . . .	36
3.3	Counterexample used in Proposition 1. . . . .	54
4.1	Summary of notation. . . . .	64
4.2	<i>SREP</i> over a 10,000 nodes network. $p = 0.24$ . . . . .	82
5.1	Set reconciliation protocols currently supported by <i>GenSync</i> and their capabilities. . . . .	88

5.2	Network configurations chosen to mimic both mobile broadband and consumer Internet networks. We consider both symmetrical and asymmetrical bandwidth Internet connections. . . . .	98
5.3	Compute configurations include both symmetrical ( <i>i.e.</i> , $\mathcal{A}$ , and $\mathcal{B}$ ) and asymmetrical ( <i>i.e.</i> , $\mathcal{CA}$ and $\mathcal{CB}$ ) compute scenarios. . . . .	99
5.4	Illustrative system configuration. . . . .	101
5.5	Emulation parameters for the Colosseum's Boston cellular scenario [172].	119

# List of Figures

1·1	Three tiers of computing continuum: smart devices, edge/fog, cloud.	5
2·1	Subtraction of $IBF_A$ and $IBF_B$ . Both IBFs use two hash functions to determine the buckets where to insert elements. . . . .	20
2·2	The structure of a Cuckoo filter with 6 inserted elements. . . . .	21
2·3	End-to-end block propagation as a one-way set reconciliaiton problem.	26
3·1	Protocol diagram of the IBLT-based set reconciliation. . . . .	30
3·2	Protocol diagram of the Cuckoo filter-based set reconciliation. Alice keeps set $S_A$ , Bob keeps set $S_B$ . . . . .	32
3·3	Protocol diagram of the CPI-based set reconciliation. Alice keeps set $S_A$ , Bob keeps set $S_B$ . . . . .	34
3·4	Average load factor $\alpha$ for cuckoo filters of sizes $2^{14}$ , $2^{15}$ , $2^{16}$ ( $b = 4$ , $f = 12$ ), over 100 experiments. . . . .	51
3·5	Space overhead as a function of $n_{max}$ for $b = 4$ (referential implementation [77]). . . . .	56
3·6	Space utilization of the referential Cuckoo filter implementation [77] and the Bloom filter implementation from Bitcoin Core [145]. . . . .	57
3·7	Empirical false negatives rate for $b = 4$ , $f = 8$ cuckoo filters, with $n_{max}$ not a power of two. . . . .	58

4.1	One iteration of <i>SREP</i> on a tractably small network $G = (V, E)$ . Each node initially contains only one element equal to the node's unique identifier. The replicas at node $n$ correspond to its neighbors and we denote them as $S_n^i$ , for $i \in G[n]$ . Transaction pools at each node are denoted as $S_n$ for $n \in V$ . . . . .	62
4.2	The small world property in random graphs generated through Watts-Strogatz model with $k = \overline{deg} = 19$ and rewire probability $p = 0.24$ . . . . .	65
4.3	Average pair-wise path lengths in Watts-Strogatz model for various $k$ and rewire probability $p = 0.24$ . . . . .	71
4.4	Amount of redundant t network of 100 nodes ( $p = 0.24$ ). . . . .	72
4.5	Empirical distributions of transaction pool sizes $\mathcal{S}$ for two adjacent Bitcoin nodes (up) and their mutual differences $\mathcal{P}$ (down). Best distribution fits in red (using Error Sum of Squares). . . . .	78
4.6	Empirical differences distribution for two adjacent Bitcoin nodes versus the differences distribution generated by Procedure 5 for various $\psi$ . Watts-Strogatz network with 100 nodes ( $\overline{deg} = 19$ and $p = 0.24$ ). . . . .	79
4.7	Maximal number of <i>SREP</i> iterations at any node ( $I_{100\%}$ ) bounded by the network diameter for Watts-Strogatz graphs with 1000 nodes ( $p = 0.24$ ). 95% confidence intervals. . . . .	79
4.8	Relative communication cost ( $C_{100\%}$ ) and time to fully synchronize the network ( $T_{100\%}$ ). Network with 1000 nodes ( $p = 0.24$ ). . . . .	80
4.9	Normalized overall communication cost of <i>SREP</i> ( $C_{100\%}$ ) and <i>MempoolSync</i> as a function of network size. Transaction pool sizes and differences from our Bitcoin measurement campaign (Section 4.3.1). $DefTXtoSync = 1000$ . $Y$ is the <i>MempoolSync</i> heuristic constant. . . . .	81
5.1	<i>GenSync</i> middleware structure. . . . .	89

5.2	Relation between four core <i>GenSync</i> abstractions. . . . .	90
5.3	<i>GenSync</i> usage example. Alice (left) and Bob (right) sync their corresponding sets $S_A$ and $S_B$ . Each time Alice wants to sync with Bob, she can choose between CPI-over-TCP and IBLT-over-TCP. When Alice wants to sync with Chuck, a third participant, she always uses Cuckoo-over-TCP. . . . .	93
5.4	Total time to reconcile (TTR) as the time elapsed between the protocol initiation and completion. . . . .	97
5.5	TTR when $ S_A  =  S_B  = 10^4$ in the $(I, \mathcal{C})$ configuration (log scale). The blue vertical line marks the position of $\bar{m}$ in I-CPI. . . . .	101
5.6	Communication cost when $ S_A  =  S_B  = 10^4$ in the $(I, \mathcal{C})$ configuration (log scale). . . . .	102
5.7	Difference size $d = 30$ . Set sizes $ S_A  =  S_B $ vary from 100 to $10^6$ (in $(I, \mathcal{C})$ configuration). TTR and communication cost shown in log scale.	103
5.8	Logarithm of TRR as a function of set cardinality (size) and the number of differences ( $d$ ). Illustrative complete system configuration from Table 5.4. . . . .	104
5.9	The step increase in communication cost for Cuckoo as the cardinality increases. The number of mutual differences is constant at 100. . . .	105
5.10	The impact of the network performance and available computing power to TTR-performance. Symmetrical compute (left) versus asymmetrical compute (right). . . . .	106
5.11	The impact of the network conditions to TTR-performance for the $\mathcal{CB}$ compute configuration. Network configuration $I$ (left) versus network configuration $II$ (right). $\mathcal{CB}$ has a 3.5 times <i>smaller</i> compute power discrepancy than $\mathcal{CA}$ . . . . .	107

5·12	The impact of asymmetrical <i>network bandwidth</i> to TTR-performance. Symmetrical <i>compute</i> (left) and asymmetrical <i>compute</i> (right). . . . .	108
5·13	TTR for network configuration <i>III(b)</i> (the lower the better). $ S_A  =  S_B  = 10^4$ . 95% confidence intervals shaded. 100 iterations per point. . . . .	109
5·14	TTR in each round of incremental sync. System configuration ( <i>II, C</i> ). . . . .	112
5·15	Distribution of <i>mempool</i> sizes ( $ S_A $ and $ S_B $ ) and symmetric difference sizes ( <i>d</i> ). . . . .	116
5·16	<i>GenSync</i> protocols applied to <i>mempool</i> reconciliation. Configuration ( <i>I, C</i> ). 95% confidence intervals. . . . .	116
5·17	Bandwidth (up) and latency (down) traces from Colosseum [172], one of the world's largest wireless network emulators. . . . .	121
5·18	Sync time for the three main sync protocols in <i>good</i> (solid line) and <i>bad</i> (dashed line) network conditions. Data cardinality is $10^8$ . Confidence intervals are shaded. . . . .	121
5·19	Bandwidth consumed for the two most bandwidth-efficient sync protocols that still complete under 2 seconds in <i>bad</i> network conditions. . . . .	123
A·1	Element 3 gets transmitted from nodes 1 and 2, which makes one of these transmissions redundant. . . . .	131
A·2	Three-node sub-network of every connected network with $ V  \geq 3$ and the initial states of the local replicas at nodes 0 and 2. . . . .	132
A·3	Constructing an <i>A</i> for the given <i>M</i> may fail. . . . .	135
B·1	Simplified UML diagram of the four core <i>GenSync</i> abstractions. . . . .	139

# List of Abbreviations

API	.....	Application Programming Interface
AR	.....	Augmented Reality
ASMDS	.....	Approximate Set Membership Data Structure
CDN	.....	Content Delivery Network
CPI	.....	Characteristic Polynomial Interpolation
CPU	.....	Central Processing Unit
CSC	.....	Complete System Configuration
DCCS	.....	Distributed Computing Continuum Systems
DDoS	.....	Distributed Denial of Service (attacks)
DeFi	.....	Decentralized Finance
DLT	.....	Distributed Ledger Technology
DOCSIS	.....	Data Over Cable Service Interface Specification
ER	.....	Enhanced Reality
FPGA	.....	Field-programmable Gate Array
IBF	.....	Invertible Bloom Filter
IBLT	.....	Invertible Bloom Lookup Table
I-CPI	.....	Interactive Characteristic Polynomial Interpolation
IoE	.....	Internet of Everything
IoT	.....	Internet of Things
IoV	.....	Internet of Vehicles
LXC	.....	Linux Containers
MCHEM	.....	Massive Digital Channel Emulator
NTL	.....	Number Theory Library

QoS	.....	Quality of Service
SCTP	.....	Stream Control Transmission Protocol
SDR	.....	Software Defined Radio
SHA	.....	Secure Hash Algorithm
SRN	.....	Standard radio Nodes
TCP	.....	Transmission Control Protocol
TTR	.....	Total Time to Reconcile
UDP	.....	User Datagram Protocol
UE	.....	User Equipment
USRP	.....	Universal Software Radio Peripheral
V2X	.....	Vehicle to Everything
VR	.....	Virtual Reality

# Chapter 1

## Introduction

The next-generation decentralized systems (*e.g.*, Web3 [1], [2] and Decentralized Finance (DeFi) [3]) in conjunction with the breakthroughs in Artificial Intelligence (AI) [4], Virtual Reality (VR) [5]–[7], and Internet of Everything [8] (IoE) promise to reshape the Internet [9]. Recently both public and private sector entities have demonstrated interest in exploring the potentials of this novel family of distributed systems. The Bank of Canada is building a central bank digital currency [10], and according to McKinsey & Company, venture capital investors invested 32.4 billion dollars in Web3 in 2021, and over 18 billion dollars in the first half of 2022 only [11].

The vision of the future “decentralized Internet” differs significantly from today’s Internet clustered around a few “hypergiants” [12]. In contrast to the centralized Internet of today, decentralized Internet is built on top of a highly heterogeneous distributed system of peers (*e.g.*, IoT sensors, mobile clients, dedicated blockchain validator nodes) that produce and exchange information in a secure and transparent manner [13]. To challenge the status quo, the proponents of decentralized Internet promise user ownership of data and digital assets, data usage auditing (*e.g.*, through smart contracts [14]), and distributed trust [15]. Despite the promises, it is still unclear if state-of-the-art technologies can live up to the expectations, especially when it comes to scalability and performance.

## 1.1 The Blockchain Example

The distributed ledger technologies (*e.g.*, Blockchain), the core of Web3, have been known to have an inferior transaction throughput performance compared to traditional payment systems [16]. In contrast to traditional payment systems, blockchain transactions are grouped into blocks, and a collection of blocks is combined into a chain using cryptographic techniques. The consensus protocol of the blockchain (*e.g.*, proof-of-work or proof-of-stake) ensures that no single participant can independently alter such created chain of blocks [17].

To submit a new transaction to the blockchain, a participant needs to create the transaction and submit it to an honest *validator* node. The validator first makes sure that the transaction is meaningful, *e.g.*, that blockchain contains no two transactions spending the same funds (*double-spending* [18]), and then disseminates the valid transaction to the other participants in the network. In this process often referred to as *transaction dissemination*, validator nodes typically record the received transactions to their *transaction pools* (see Section 2.4.1).

However, a transaction is not considered *finalized* as long as it is not included in a block accepted by the majority of the validators. For this to happen, a *block-creating* node (*e.g.*, miner in the Bitcoin network) needs to include the transaction in a block and propagate the block to its peers in the process that is often called *block propagation*. The block propagation process is critical to blockchain performance because the number of transactions that a blockchain can handle per unit of time (transaction throughput) is determined by two main factors: (1) the number of transactions that are included in a block, and (2) the time needed for the block to reach the majority of the network. The latter not only impacts the performance of the blockchain but also its security [19], [20], and researchers have exploited imperfect synchronization to mount various attacks against the popular blockchain implementations such as

Bitcoin [21].

In the core of modern block propagation protocols such as Compact Block [22] and Graphene [23] is the idea of synchronizing the transaction pools of the nodes that participate in block propagation. First, the sender announces the block to the receiver by sharing its globally unique identifier. If the block is not present at the receiving end, the receiver requests the newly created block. The sender encodes the block as a collection of transactions and sends it to the receiver. As transaction dissemination and block propagation happen simultaneously, many transactions contained in the block that is being propagated are likely present in the transaction pool of the receiving node. Hence, the efficiency of a block propagation protocol is the measure of the protocol’s ability to learn the minimal amount of information that the receiver needs to receive. More generally, the propagation can be viewed as a synchronization of the two sets: the set of transactions included in the block, and the transaction pool of the receiving node (see Section 2.4.2).

## 1.2 Data Synchronization in Traditional Distributed Systems

As apparent from the example of modern block propagation protocols, a fundamental problem that the envisioned future decentralized systems share with traditional distributed systems is *data synchronization* (sync). Historically, sync has been studied as a subproblem in various systems involving *data replication* — a common technique for increasing fault tolerance, availability, and performance in distributed systems [24].

In Content Delivery Networks (CDN), the content is replicated from origin servers to many “edge” servers that are closer to the content consumers [25]. In so doing, CDNs decrease the average latency of content delivery and can even improve the resilience to Distributed Denial of Service (DDoS) attacks [26], [27].

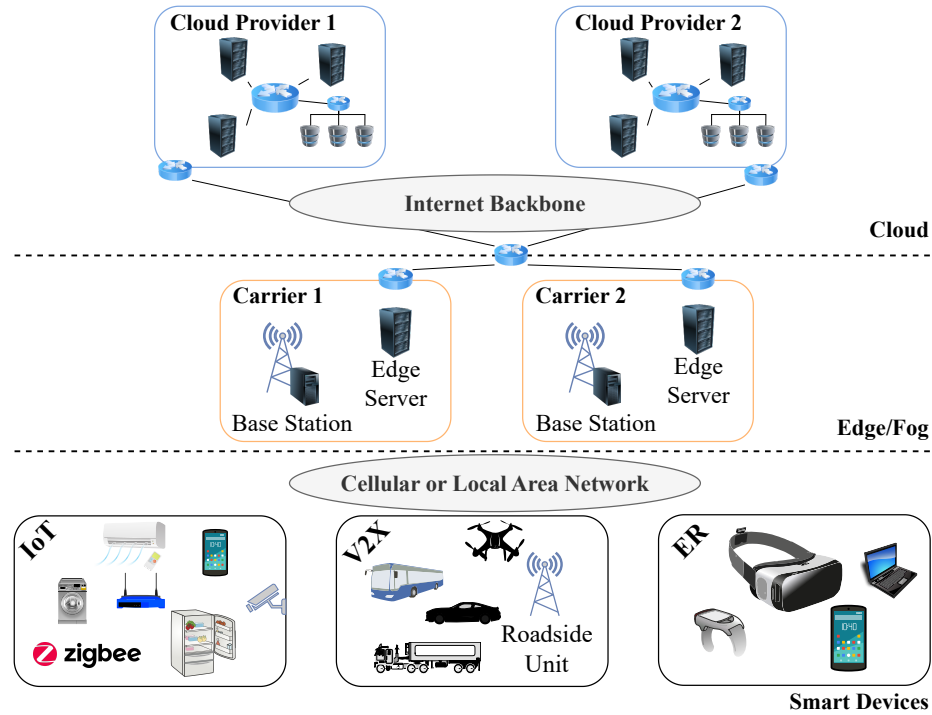
In cloud storage services such as Apple iCloud, Dropbox, Google Cloud, and

Microsoft OneDrive, sync is utilized to incorporate the changes from user devices into the cloud. As cloud storage services are known to generate tremendous amounts of network traffic for the cloud providers [28], [29], the reduction in wasted bandwidth is the main goal. The problem gets even more challenging in mobile computing where user devices may have constrained connectivity, reduced bandwidth, or power-saving constraints [30]. In that case, we are not only concerned about the traffic generated on the cloud provider’s end but also the computational burden on the user device. Recently, MongoDB has introduced their product named Atlas Device Sync [31] to automatically propagate data updates to all registered devices via their cloud-hosted database. Approaches similar to that of MongoDB are likewise being explored by the researchers [32].

In IoT, sync is used for data offloading [33] and device-to-device synchronization [34]. The former is motivated by the discrepancy between the amount of data that the devices produce and their ability to store it [33]. The latter is driven by the need to combine the data from nearby devices to support efficient decision making. While syncing with the cloud is sometimes less time-sensitive and can be done in batches (*e.g.*, storing logs in the cloud), device-to-device synchronization is often done in real-time [35], [36].

### 1.3 Cloud/Edge Convergence

Cloud computing paradigm has been dominating the last decade. However, the technology shifts inspired by 6G applications [37] such as virtual/enhanced reality (VR/ER) and federated machine learning [38], [39] drive the innovations that transform the traditional cloud computing paradigm into a multi-tiered computing continuum [40], [41]. Instead of offloading the majority of computation and storage to the centralized cloud platforms, future applications will fully exploit the capabilities of



**Figure 1.1:** Three tiers of computing continuum: smart devices, edge/fog, cloud.

the networks and devices on the lower layers of the computing continuum. Operating in a more *decentralized* framework, the future applications can achieve better resource utilization, enhance user security and privacy, and support critical low latency applications.

This 6G technology shift has distilled the three main layers in the computing continuum, which we depict in Fig. 1.1. At the lowest layer are the smart devices that communicate directly with each other using dedicated communication protocols. This layer comprises of the traditional IoT devices (*e.g.*, smart home), Enhanced Reality (ER) devices, smart vehicles with supporting infrastructure, and other. In the middle is the edge layer with dedicated edge servers that connects to the devices through an access network. The cloud is only the last and the furthest layer of the computing continuum that offers the largest compute power and vast storage, but

incurs the largest latency for the applications.

## 1.4 Research Questions

The heterogeneity of the computing continuum and the decentralized nature of the applications that aim at using it call for new approaches to data synchronization. While traditional applications used centralized architectures and relied on homogeneity of the underlying systems, the future applications will use decentralized architectures and operate in highly heterogeneous systems. We see this new reality of distributed computing as an opportunity to improve the overall application performance. In particular, we focus on *set reconciliation* as a data synchronization approach, and in this regard, we formulate the following thesis statement.

**Thesis statement:** *Set reconciliation protocols can improve data synchronization performance in decentralized applications by leveraging the properties of the underlying networked systems and the data that is being synchronized.*

We explore the above thesis statement through the following main research questions:

**Research Question RQ1:** How do state-of-the art set reconciliation protocols compare to each other in practice?

**Research Question RQ2:** What are the factors that determine the best set reconciliation protocol choice for the given system conditions? Is there a *universally dominant* protocol with superior performance in diverse conditions?

**Research Question RQ3:** How do applications utilize set reconciliation protocols, and what design choices can affect application performance?

**Research Question RQ4:** How do we choose the best set reconciliation protocol given the emulation of the target networked system?

**Research Question RQ5:** Can set reconciliation improve transaction pool synchronization in large-scale blockchains?

## 1.5 Contributions

By tackling the above questions, we made several research contributions.

**Comparative analysis of the state-of-the art.** Many disparate set reconciliation protocols have been proposed in the literature. While one family of these protocols is based on *approximate set membership data structures* (e.g., Bloom and Cuckoo filter variants), the other are inspired by the results from the coding theory (e.g., Reed-Solomon codes). Prior to our work [42], there was a lack of systematic analytical comparison among the state-of-the art set reconciliation protocols. That is, the protocols were typically compared only against those that use the similar algorithmic principles.

In Chapter 3 we explore our **RQ 1** and present a comprehensive analytical comparison of the existing state-of-the art set reconciliation protocols with respect to (1) communication overhead (network bandwidth overhead), (2) computational complexity, and (3) message complexity (rounds of communication). Beyond these three standard measures, we analyze the other factors that may impact the practical performance of set reconciliation. Precisely, we analyze the problem of estimating the upper bound on the number of symmetric differences and identify a trade-off between message complexity and communication overhead. While some protocols can achieve very competitive communication efficiency, they do so at the cost of additional round trips, which can be particularly harmful in the high latency networks. On the other hand, even with the given tight bound on the number of symmetric

differences, some protocols may fail to synchronize with arbitrarily small probability while others always succeed. Finally, we identify the sources of synchronization inaccuracy in Cuckoo filter-based protocols as originating from both hash collisions and the concrete design choices in referential implementations.

***GenSync* framework.** To explore our **RQ 2**, we designed and implemented *GenSync*, the first unified data synchronization framework based on set reconciliation. *GenSync* is implemented as a self-contained C++ middleware library that can be seemingly integrated in existing implementations through a minimal API, and comes with several generic implementations of both approximate set membership data structures-based and coding theory-inspired set reconciliation protocols. *GenSync* users can straightforwardly extend the framework with platform-specific implementations or even new set reconciliation protocols by reusing the existing framework components.

One of the distinctive features of *GenSync* is its benchmarking layer for experimental comparison of the state-of-the-art set reconciliation protocols under realistic system conditions. The application designers can model their target system properties in terms of computational power of nodes and the properties of the communication channel (*e.g.*, network latency and bandwidth). Upon configuring *GenSync*'s benchmarking layer to mimic their target conditions, the designers can independently perform the comparative evaluation of included protocols and pick the best one for their application. On the other hand, the researchers can utilize *GenSync* as a fair benchmark to exercise their proposed solutions against the state-of-the-art.

Using *GenSync*, we perform an extensive comparative analysis among the three common state-of-the-art set reconciliation protocols: Characteristic Polynomial Interpolation (CPI), Invertible Bloom Lookup Table-based (IBLT), and Cuckoo filter-based. We find that there is *no universally dominant* protocol among those included in *GenSync*. The best protocol choice is highly sensitive to the target conditions,

and a bad protocol choice for the given conditions may lead to a five-fold hit in the real-time synchronization performance. Furthermore, the asymmetry of set reconciliation protocols such as the IBLT-based one can make them an unfavorable choice in highly asymmetrical compute scenarios (*i.e.*, when one participant has much more computing power than the other).

**Analysis of common synchronization styles.** The applications that repeatedly synchronize their data utilize the data synchronization protocols in two main styles: *cold-start* and *incremental*. The former denotes the family of scenarios when the participants do not maintain the synchronization metadata between consecutive rounds of synchronization. In the later, the participants build the synchronization data structures incrementally as data arrives and keep track of what has been done in the previous synchronization rounds.

We explored our **RQ 3** through comparing these two main usage styles under practical system conditions and found that interactive protocols such as Interactive Characteristic Polynomial Interpolation (I-CPI) can beat their non-interactive counterparts by orders of magnitude in real-time synchronization performance. While the interactive protocols exhibit stable synchronization performance over time as long as the arrivals of new data points do not exceed an upper bound, their non-interactive counterparts deteriorate in performance as a function of both time and the average amount of newly arrived data per unit of time.

**Modeling performance in emulated wireless networks.** A faithful emulation of the target networked system can help application designers to assess the practical performance of their applications. As part of our **RQ 4**, we integrated *GenSync* into Colosseum, one of the world largest wireless emulators, with the main goal to perform the cost-benefit analysis among the protocols included in *GenSync*.

We configure Colosseum to emulate the cellular network in the surroundings of

the Boston Common public park in Boston, Massachusetts. In one set of experiments, we configure the network users to move at the pedestrian speed (*i.e.*, 5 m/s) within the 20 meters radius around their corresponding base stations, and in the other we make them stationary. As these movement patterns cause the network performance variations in Colosseum, we analyzed the experimental performance of *GenSync* protocols under the *good* and *bad* network conditions. We found that IBLT-based set reconciliation performs the best for the general usage. However, for the applications that want to be conservative about their bandwidth usage, CPI may be the right choice. Given the target maximal duration of synchronization, CPI beats IBLT in bandwidth overhead by up to nine times.

**Set Reconciliation Enhanced Propagation (*SREP*).** As we touched upon in Section 1.1, the synchronization of transaction pools has shown a great potential in reducing the block propagation delay in blockchain systems, and consequently their scalability and security. Indeed, several heuristics that integrate the synchronization of transaction pools into block propagation protocols have been proposed [23], [43].

In tackling our **RQ 5**, we take a different approach, maintaining transaction pool synchronization outside (and independently) of the main block propagation channel. In doing so, we formalize the problem of network-wide set reconciliation within a graph theoretic framework and introduce a novel distributed algorithm for memory pool synchronization called *SREP* (*Set Reconciliation Enhanced Propagation*) with quantifiable performance guarantees. We analyze *SREP*'s performance in various realistic network topologies, and show that *SREP* converges on any connected graph in the number of steps bounded by the network diameter. We confirm our analytical findings through extensive simulations, including the comparison with *MempoolSync*, a similar heuristic approach from the literature. Our simulations show that *SREP* incurs reasonable overall network bandwidth overhead, and unlike *MempoolSync*, scales

gracefully with the size of the network.

Our results are particularly significant when considered in conjunction with the recent results from the blockchain network measurement research [44], [45] that confirmed the “small-world” property of the underlying peer-to-peer networks in popular blockchains (*e.g.*, Ethereum). In such networks, *SREP* converges in the number of steps *logarithmic* in the number of participants, which makes it amenable for large-scale blockchains (*e.g.*, tens of thousands of participants). Furthermore, our novel simulation technique incorporates real world transaction pool data obtained from a measurement campaign performed in the live Bitcoin network.

## 1.6 Publications

We provide a wider context to our research in a number of publications [42], [46]–[53], and in this dissertation, we draw most heavily from:

- [46] N. Boškov, S. Simsek, A. Trachtenberg, and D. Starobinski, “*SREP: Out-Of-Band Sync of Transaction Pools for Large-Scale Blockchains*,” in 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC) (*accepted*), <https://arxiv.org/abs/2303.16809>.
- [47] N. Boškov, A. Trachtenberg, and D. Starobinski, “*Enabling Cost-Benefit Analysis of Data Sync Protocols*,” IEEE Computer, 2023 (*accepted*), <https://arxiv.org/abs/2303.17530>.
- [42] N. Boškov, A. Trachtenberg, and D. Starobinski, “*GenSync: A New Framework for Benchmarking and Optimizing Reconciliation of Data*,” IEEE Transactions on Network and Service Management, vol. 19, no. 4, pp. 4408–4423, 2022.
- [50] N. Boškov, A. Trachtenberg, and D. Starobinski, “*Birdwatching: False Negatives in Cuckoo Filters*,” in Proceedings of the Student Workshop, ser.

CoNEXT'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 13-14.

In particular, Chapter 3 is primarily based on our results from [42] and [50], Chapter 4 is based on [46], and Chapter 5 is based on [42].

## 1.7 Outline

The rest of this dissertation is organized as follows. We lay out the background and related work in Chapter 2. In Chapter 3 we focus on end-to-end data synchronization and analyze the state-of-the-art set reconciliation protocols. First, we compare the structure of various protocols and summarize their asymptotic behavior. Then, we discuss the theoretical assumptions that impact the practical performance of the protocols. In Chapter 4, we introduce *SREP*, a novel distributed transaction pool synchronization protocol based on efficient end-to-end set reconciliation. In Chapter 5, we present *GenSync*, a novel framework that allows for easy integration of various set reconciliation protocols in decentralized applications. We conclude this dissertation and outline the future research directions in Chapter 6.

## Chapter 2

# Background and Related Work

In this section, we first introduce the set reconciliation problem and give a brief overview of the approaches from the literature. Then, we introduce the approximate set membership data structures that we use later in Chapter 3 and Chapter 5. Finally, we give a brief overview of the blockchain concepts that we rely on in Chapter 4.

### 2.1 Set Reconciliation

The set reconciliation problem involves two parties Alice and Bob with their corresponding sets  $S_A$  and  $S_B$ , and the goal is to make the two sets equal while minimizing the communication complexity. The problem has first been formulated by Minsky, Trachtenberg and Zippel [54].

#### 2.1.1 Definitions

**Local elements.** In the set reconciliation problem, we use the term *local elements* to denote the elements known to only one of the parties. In other words, we define the elements local to Alice as  $S_A \setminus S_B$ , and the elements local to Bob as  $S_B \setminus S_A$ .

**Symmetric differences.** The *set of symmetric (mutual) differences*, denoted as  $S_A \oplus S_B$ , is defined as the union of the elements local to Alice and the elements local to Bob:

$$S_A \oplus S_B = \underbrace{S_A \setminus S_B}_{\text{“Elements local to Alice”}} \cup \underbrace{S_B \setminus S_A}_{\text{“Elements local to Bob”}} .$$

The cardinality of the set of symmetric differences is denoted as:

$$d = |S_A \oplus S_B|.$$

In this dissertation, we focus on the cases when Alice and Bob keep the sets that are large relative to their symmetric difference (*i.e.*,  $|S_A \cup S_B| \gg d$ ). In general, set reconciliation protocols draw their power from the similarity between the sets: the larger the set intersection ( $S_A \cap S_B$ ), the more benefits these protocols provide.

***Universe.*** We use  $\mathcal{U}$  to denote the *universe* from which the set elements are drawn. That is,  $S_A \cup S_B \subseteq \mathcal{U}$ .

***One-way versus full reconciliation.*** When only one party is interested in learning the elements local to the other, we use the term *one-way* set reconciliation. Otherwise, when both parties are interested in learning the local set of the other, we use the term *full* set reconciliation.

***Message complexity.*** If not otherwise stated, we use the term *message complexity* to denote the number of messages exchanged between the parties until the completion of the protocol.

***Computational complexity.*** The computational complexity of a set reconciliation protocol is the sum of the computational complexities of the procedures executed by both parties.

***Data point.*** The set reconciliation protocols that we consider in this dissertation represent set elements as constant-sized identifiers called *data points*. Each data point identifies one concrete record.

The mapping between concrete records and data points is often implemented through hashing, and comes naturally in some applications. For example, Bitcoin transactions can vary in size, but each transaction has a globally unique identifier that is a 32-byte long word. Therefore, Bitcoin block propagation protocols often use

transaction identifiers as data points.

**Communication overhead.** If not otherwise stated, we use the term *communication overhead* (cost) to denote the sum of the sizes of all messages transmitted over the network to complete a set reconciliation protocol. For example, when data points are chosen from a universe of  $b$ -bit words, a naive protocol that exchanges the entire sets has the communication overhead of  $b(|S_A| + |S_B|)$ .

### 2.1.2 Assumptions

Although some set reconciliation protocols from the literature operate under slightly modified set of assumptions, the following assumptions are common among most protocols.

**No prior context** — in this dissertation, we focus on the scenarios when the involved parties do not keep reconciliation-related memory. Alice has no knowledge of Bob other than what is required to connect with him, and vice versa. Alice and Bob further have no knowledge about any previous synchronizations. However, in interactive set reconciliation (Chapter 3 and Chapter 5), the parties are allowed to keep reconciliation-related data structures *during* the multi-round protocol. This is because the entire duration of the interactive protocol is considered a single reconciliation session.

**No handling of deletions** — related to the prior assumption is the fact that set reconciliation protocols typically do not handle deletions. Suppose that Alice and Bob reconcile their sets periodically starting with  $S_A = \{1\}$  and  $S_B = \{2\}$ , meaning that after the first reconciliation has been completed  $S_A = S_B = \{1, 2\}$ . Assume now that Bob deletes element 2 in between the first and the second reconciliation rounds. Since there is no notion of deletions, Alice will detect element 2 as a symmetric difference and send it to Bob in the second reconciliation round, thus annihilating Bob's deletion. This is related to the assumption of no reconciliation-related memory.

There are straightforward potential mechanisms for dealing with deletions. For example, we can keep a separate set of deleted elements and reconcile that set as well. Prior to reconciling the main sets, the parties can reconcile the set of deletions. Any new elements obtained in this process would get deleted from the main set prior to its reconciliation. Another potential mechanism is to add a presence bit to each set element. Notwithstanding, handling deletions is out of scope of this dissertation.

### 2.1.3 Set Reconciliation as a Data Synchronization Approach

Set reconciliation is the data synchronization approach that we focus on in this dissertation. The other data synchronization approaches include the variants of *rsync* [28], [29], [55], [56], the file synchronization protocol included in many Linux distributions. The main difference between set reconciliation protocols and *rsync*-like approaches is that the latter works with ordered collections (*e.g.*, strings) and not unordered sets.

Set reconciliation protocols can be specialized to work efficiently with files. For instance, Song and Trachtenberg [57] used a technique called Recursive Content-Dependent Shingling in conjunction with the Interactive Characteristic Polynomial Interpolation-based set reconciliation (see Section 3.3.2) to implement an efficient file synchronization protocol. For large files, the protocol of Song and Trachtenberg beats *rsync* in communication efficiency.

Throughout this dissertation, we view set reconciliation as more general approach than file synchronization and use *data synchronization* as a synonym to data set synchronization.

## 2.2 Set Reconciliation Protocols in the Literature

In the last two decades, many disparate set reconciliation protocols have been proposed in the literature. Besides the protocols proposed for the general purpose of synchronizing large highly similar sets, there are also protocols proposed in more

Author	Algorithm	Applicability	Class
Skjegstad and Torleiv [58]	BF		
Tian <i>et al.</i> [59]			
Fan <i>et al.</i> [60]	Counting BF		
Luo <i>et al.</i> [61]	CF		
Li <i>et al.</i> [62]	Counting CF	General	
Luo <i>et al.</i> [63]	Marked CF		<b>ASM</b>
Eppstein and Goodrich [64]	IBLT		
Lázaro and Matuz [65]			
Lee <i>et al.</i> [66]	Ternary IBLT		
Ozisik <i>et al.</i> [23]	BF + IBLT	Blockchain	
Ding <i>et al.</i> [67]	CF		
Minsky and Trachtenberg [68]	CPI		
Dodis <i>et al.</i> [69]	BCH	General	
Gong <i>et al.</i> [70]	Parity Bitmap Sketch		<b>ECC</b>
Jin <i>et al.</i> [71]	CPI	Disruption Tolerant Networks	
Naumenko <i>et al.</i> [72]	BCH + low-fanout flooding	Blockchain	

**Table 2.1:** Set reconciliation literature summary. **BF** *abbr.* Bloom filter. **CF** *abbr.* Cuckoo filter. **IBLT** *abbr.* Invertible Bloom Lookup Table. **ASM** *abbr.* Approximate Set Membership. **CPI** *abbr.* Characteristic Polynomial Interpolation. **BCH** *abbr.* Bose-Chaudhuri-Hocquenghem (codes). **ECC** *abbr.* Error-Correcting Codes.

specific frameworks such as blockchain and disruption tolerant networks. Most set reconciliation protocols proposed in the literature draw their inspiration from either approximate space-efficient set representations or error correcting codes (*e.g.*, Reed-Solomon, or Bose-Chaudhuri-Hocquenghem). In Table 2.1, we summarize the related work in the area.

In the rest of this section, we describe the approximate set membership data structures used in the protocols that we analyze in Chapter 3, and then introduce the concepts that we rely on in Chapter 4 such as blockchain block propagation. Finally, we introduce the results from the blockchain topology measurement research that we use in Chapter 4.

## 2.3 Approximate Set Membership Data Structures

Approximate set membership data structures (ASMDS) are a family of compact set representations that can approximately answer membership queries in constant time [73]. Suppose that an ASMDS represents set  $S$ , a membership query for an element  $e \notin S$  may falsely return a positive answer with a probability at most  $\epsilon$  (*false positive rate*). Due to their compactness, ASMDSs are often utilized in set reconciliation protocols. Besides space-efficiency, particularly important for set reconciliation is that many ASMDS come with fast construction times. In this section, we touch upon the construction of the ASMDSs used later in this dissertation.

### 2.3.1 Invertible Bloom Filter

Traditional Bloom filter [74] is a simple ASMDS that uses a bit array of size  $m$  and  $k$  independent hash functions ranging from 0 to  $m - 1$ . Insertions consist of calculating  $k$  hashes and setting corresponding bits in the bit array. The membership query calculates the same  $k$  hashes and checks whether all  $k$  bits are set.

Invertible Bloom Filter (IBF) is an extension to the traditional Bloom filter originally introduced by Eppstein and Goodrich [75], and later extended to Invertible Bloom Lookup Table by Goodrich and Mitzenmacher [76]. The construction process of IBF is similar to traditional Bloom filters in that there are  $k$  hashes used for each inserted element to determine the buckets to store the element. In contrast to the traditional Bloom filter, IBF uses one more hash  $H$ . Each bucket of an IBF stores three fields:

- (1) ***hashSum*** — the sum of hashes of all elements inserted into the bucket.

Suppose that a bucket stores  $\{x_1, x_2, x_3\} \subseteq \mathcal{U}$ , then  $hashSum = H(x_1) \oplus H(x_2) \oplus H(x_3)$ , where  $\oplus$  denotes bitwise XOR.

- (2) ***count*** — the number of elements inserted in the bucket, and
- (3) ***idSum*** — the sum of all elements inserted into the bucket. For the previous example,  $idSum$  is  $x_1 \oplus x_2 \oplus x_3$ .

### Subtraction

In contrast to traditional Bloom filters, we can subtract two IBFs to get the third one that encodes mutual differences. As illustrated in Fig. 2-1, subtraction is done bucket-wise. That is, we traverse buckets in  $IBF_A$  and  $IBF_B$  in parallel, subtract the corresponding *count* fields, and XOR-ing the corresponding *idSum* and *hashSum* fields. The common elements cancel out, and with an arbitrarily high probability, leave one or more buckets keeping only one element.

We can verify that there is only one element in the bucket by computing the hash of the *idSum* field and comparing it with the *hashSum* field. If there is a match, then we know that the element is present in only one of the filters involved in subtraction. To determine the exact filter that has originally encoded the element,

$$IBF_A = \{A, B, C, E, F\}$$

	bucket 0	bucket 1	bucket 2	bucket 3
<i>idSum</i>	$A \oplus B$	$B \oplus E \oplus F$	$C \oplus E \oplus F$	$A \oplus C$
<i>hashSum</i>	$H(A) \oplus H(B)$	$H(B) \oplus H(E) \oplus H(F)$	$H(C) \oplus H(E) \oplus H(F)$	$H(A) \oplus H(C)$
<i>count</i>	2	3	3	2

$$IBF_B = \{A, B, D, E\}$$

<i>idSum</i>	$A \oplus B \oplus D$	$B \oplus D \oplus E$	$E$	$A$
<i>hashSum</i>	$H(A) \oplus H(B) \oplus H(D)$	$H(B) \oplus H(D) \oplus H(E)$	$H(E)$	$H(A)$
<i>count</i>	3	3	1	1

$$IBF_A - IBF_B = \{C, D, F\}$$

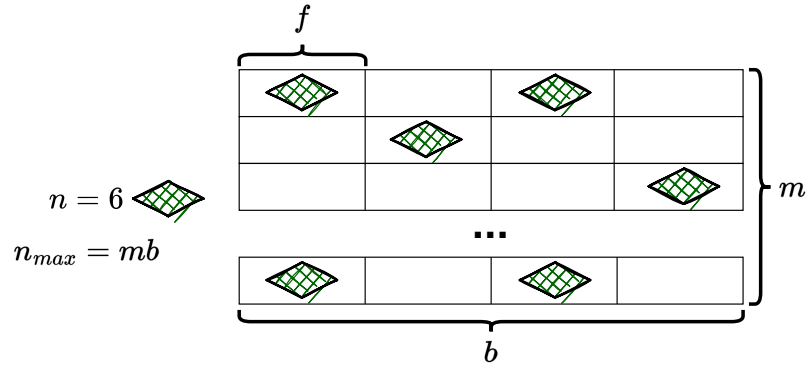
<i>idSum</i>	$D$	$F \oplus D$	$C \oplus F$	$C$
<i>hashSum</i>	$H(D)$	$H(F) \oplus H(D)$	$H(C) \oplus H(F)$	$H(C)$
<i>count</i>	-1	0	2	1

**Figure 2-1:** Subtraction of  $IBF_A$  and  $IBF_B$ . Both IBFs use two hash functions to determine the buckets where to insert elements.

we can look at the *count* field. If the *count* field is positive, then the element has been originally encoded into  $IBF_A$ . Otherwise, it has been originally encoded into  $IBF_B$ . We can now proceed with decoding the resulting filter by subtracting the recovered elements from all other buckets in the filter. This iterative decoding process of IBFs is commonly referred to as the “peeling” process.

In the example from Fig. 2-1, buckets 0 and 3 in  $IBF_A - IBF_B$  contain only one element each. The first bucket contains  $D$ , and its *count* field indicates that  $D$  has been originally encoded into  $IBF_B$ . On the other hand, the positive sign in the *count* field of bucket 3 indicates that  $C$  has been initially encoded in  $IBF_A$ . We can then subtract  $C$  and  $D$  from all buckets in  $IBF_A - IBF_B$  to continue the decoding process.

As apparent from our example in Fig. 2-1, the peeling process may fail for adversarially chosen elements. Given the IBF, an adversary can choose the set of elements



**Figure 2.2:** The structure of a Cuckoo filter with 6 inserted elements.

to be inserted so that there is no bucket that contains only one element. When that happens, the peeling process cannot proceed, and the elements stored in the filter cannot be recovered. In practice, the probability of decode failures can be controlled through choosing IBF parameters. We discuss the subtleties of IBF parameterization later in Section 3.3.4.

For subtraction to work, two IBFs need be of the same size and use the same collection of hash functions to determine buckets and compute *hashSum*. This constraint is particularly significant in IBF-based set reconciliation protocols, as Alice and Bob need to agree on the parameters of their filters at the beginning of the reconciliation protocol.

### 2.3.2 Cuckoo Filter

Cuckoo filters are introduced by Fan *et al.* [77] as a more space-efficient replacement for traditional Bloom filters. Similarly to the traditional Bloom filters, Cuckoo filters supports insertions and lookups. In addition, Cuckoo filters also implements deletions. As depicted in Fig. 2.2, a Cuckoo filter is divided into  $m$  buckets that each contain  $b$  slots. Each slot is  $f$  bits wide and stores a fingerprint of an inserted element. A fingerprint is obtained through the fingerprint hash  $x_f$ , and the collisions on  $x_f$  cause certain amount of false positives of lookup. The amount of false positives in Cuckoo

---

$\epsilon$	false positive rate
$f$	fingerprint size in bits
$b$	number of slots per bucket
$m$	number of buckets in filter
$n$	number of items currently in filter
$n_{max} = mb$	maximum number of items
$\alpha = n/n_{max}$	load factor ( $0 \leq \alpha \leq 1$ )
$x_f$	fingerprint hash
$H$	bucket hash
$MaxNumKicks$	maximum number of kicks

---

**Table 2.2:** Cuckoo filter notation.

filter  $C$  is bounded by its false positive rate  $\epsilon_C$ .

When an element is being inserted into a Cuckoo filter, the fingerprint hash is used together with an independently chosen bucket hash  $H$  to compute the candidate buckets  $i_1$  and  $i_2$  ( $\oplus$  denotes bitwise XOR):

$$\begin{aligned} \phi_x &= x_f(x) \\ i_1 &= H(x) \pmod m \\ i_2 &= i_1 \oplus H(\phi_x) \pmod m \end{aligned} \tag{2.1}$$

The insertion process then proceeds as in Algorithm 1. If  $i_1$  can accommodate the new element, it is inserted there. Otherwise, the insertion is attempted in the *alternative* bucket  $i_2$ .

If neither of the two buckets can accommodate the new element, then one fingerprint  $\phi_x$  from the bucket  $i_c$  is chosen at random and its own alternative bucket is calculated as:

$$i_a = i_c \oplus H(\phi_x) \pmod m \tag{2.2}$$

The process continues recursively until a bucket that can accommodate the new

---

**Algorithm 1:** Insert( $x$ ) from Fan *et al.* [77]

---

```

1  $\phi_x = x_f(x)$ ;
2  $i_1 = H(x) \bmod m$ ;
3  $i_2 = i_1 \oplus H(\phi_x) \bmod m$ ;
4 if bucket  $i_1$  or bucket  $i_2$  has an empty slot then
5   |   add  $\phi_x$  to the bucket;
6   |   return True;
7  $b =$  randomly pick  $i_1$  or  $i_2$ ;
8 for  $k \leftarrow 1$  to  $MaxNumKicks$  by 1 do
9   |    $\phi_v =$  randomly selected fingerprint from  $b$ ;
10  |   write  $\phi_x$  to  $\phi_v$ 's slot;
11  |    $\phi_x = \phi_v$ ;
12  |    $b_{alt} = b \oplus H(\phi_x) \bmod m$ ;
13  |   if  $b_{alt}$  has an empty slot then
14  |   |   write  $\phi_x$  to alt;
15  |   |   return True;
16 return False;

```

---

element is found, or a constant threshold of attempted insertions,  $MaxNumKicks$ , is exceeded. In the latter case, the insertion process fails.

Note the following property of the Cuckoo insertion process: if the eviction of a fingerprint  $\phi$  from bucket  $\alpha$  sends it to bucket  $\beta$ , then eviction of the same  $\phi$  from  $\beta$  should send it back to  $\alpha$ . We use the term *Cuckoo hashing idempotence* to denote this property. Given the idempotence of cuckoo hashing, the lookup procedure can compute the same candidate buckets as insertion, traverse all  $b$  slots in each of the two buckets, and determine whether the element has previously been inserted.

## 2.4 Blockchain Peer-to-Peer Network

As we touched upon in Section 1.1, Blockchain is a distributed ledger technology (DLT) built on top of a decentralized peer-to-peer network. The term “blockchain” comes from the 2008 Bitcoin paper authored by an anonymous author (or group of authors) under the presumed pseudonym Satoshi Nakamoto [78]. At the core of the blockchain idea is decentralization, *i.e.*, many participants in the peer-to-peer

network can keep the copy of the entire history of records and independently verify the validity of transactions. The integrity of the history of record is protected using cryptographic techniques.

A blockchain transaction consists of its inputs and outputs. The inputs of a blockchain transaction must be valid outputs of some previous transactions, and the authenticity of the outputs is protected using cryptographic signatures. Each transaction in the blockchain can be uniquely identified using its transaction hash (ID). In Bitcoin, for example, a transaction ID is a 256 bits long string generated through double hashing with SHA256. This assures sufficiently low probability of collisions even for the large number of transactions in the system.

When a new block is created, a group of valid but yet unprocessed transactions is combined into a block and the integrity of the block is cryptographically protected (*e.g.*, using a Merkle tree [79]). Multiple blocks are combined into a sequence, and the integrity of the sequence itself is cryptographically protected (*e.g.*, using cryptographic hashes). An important property of such created sequence of blocks is that tampering with a block noticeably breaks the chain and forging an entire chain is impractical.

#### 2.4.1 Transaction Dissemination and Memory Pools

The time that the recipients of the transaction outputs must wait until they can use the outputs as inputs to their own transactions is often referred to as the transaction ***confirmation latency*** [80]. For a transaction to get confirmed by the blockchain *at least* the following two must hold:

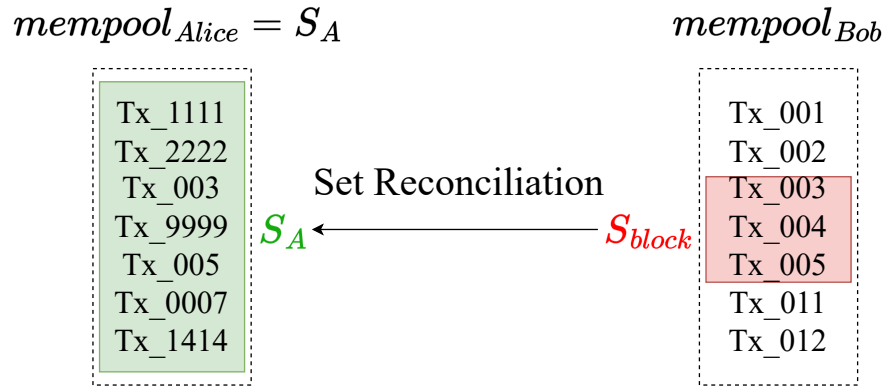
- (1) The transaction must get included into a block by a block-creating node, and
- (2) The block containing the transaction must reach the significant fraction of the network.

For the former to be satisfied, the author of the transaction has a strong incentive to share its transaction with as many block-creating nodes as possible. This is because the election of the block-creating node that will be awarded the right to create the next block is random, and ideally uniformly so.

The process in which the author of the transaction gets it to as many block-creating nodes as possible is called *transaction dissemination*. In practice, the author submits its transaction to one participant in the network and the network is responsible for disseminating it further. A straightforward method to disseminate the transaction is simple flooding (*i.e.*, everyone sends the transaction to all its neighbors). However, this consumes significant networking resources, and, instead, non-redundant flooding is typically used. The sender announces a digest of the transaction to its neighbors, and the neighbors request the full transaction only when they determine that they had not received it before. Each block-creating node keeps a collection of transactions that it has received in a data structure called *memory pool*, *transaction pool*, or *mempool*. A transaction that is being disseminated through the network stays in the memory pool of a block-creating node until one of the following happens:

- (1) The node gets awarded with the right to create the next block and it decides to include the transaction into the block. The exact method that the node uses to decide whether to include the transaction varies across implementations, but a common method is to sort the transactions by their transaction fees — the award to the block creator. By including transactions with higher fees, the block-creating node maximizes its profit from participating in the network, or
- (2) The node receives a valid block that contains a transaction from its memory pool.

When one of the two events happen, the node removes the transaction from its mem-



**Figure 2.3:** End-to-end block propagation as a one-way set reconciliation problem.

ory pool.

### 2.4.2 Block Propagation

The process in which a block-creating node propagates the block to other participants in the network is called *block propagation* and often happens independently from transaction dissemination. Traditionally (*e.g.*, first versions of Bitcoin), it has been done in the following way. First, the node that has the block sends the block identifier to a neighbor. The neighbor responds with the request for the block if it does not already have it. The block sender then sends the entire block with all its transactions. The process runs in parallel for many, or even all neighbors, contingent on the implementation.

However, the blocks can be several megabytes in size [81] and the transaction dissemination process may have already disseminated many transactions from the block to the receiving end. As a result, the amount of wasted bandwidth is prohibitively high [23], [72], [80]. Several protocols have recently been proposed to improve the efficiency of block propagation [23], [67]. For instance, Ozisik *et al.* have proposed a protocol called Graphene [23] to reduce the bandwidth usage of block propaga-

tion in Bitcoin and Ethereum using a set reconciliation protocol based on Invertible Bloom Lookup Tables. One of the significant contributions of Ozisik *et al.* is explicit modeling of block propagation as a set reconciliation problem.

We depict end-to-end (*i.e.*, two nodes) block propagation as a one-way set reconciliation problem in Fig. 2-3. Suppose that Bob keeps  $mempool_{Bob}$  and propagates the block  $S_{block} = \{Tx\_003, Tx\_004, Tx\_005\}$  to Alice whose mempool already contains  $Tx\_003$  and  $Tx\_005$ . In traditional block propagation, Bob would send all three transactions from the block to Alice, regardless of what Alice has in her memory pool. A better approach, Ozisik *et al.* have shown, is for Alice and Bob to establish a set reconciliation protocol between  $S_{block}$  on Bob's side and Alice's entire memory pool on the other. By doing so, they can efficiently learn that only  $Tx\_004$  must be transmitted, while  $Tx\_003$ , and  $Tx\_005$  are already present at the receiving end, which significantly reduces the bandwidth overhead in general case. Note that this is a one-way set reconciliation, as Bob is not interested in learning the elements local to Alice.

### 2.4.3 Topology of Popular Blockchain Networks

As opposed to centralized networks (*e.g.*, in data centers), where several entities can control the network topology, the overlay networks of *public* blockchains are decentralized and no single entity can enforce a topology on other participants. Each node can run its own version of the software, and independently decide with whom to connect. Therefore, the topology of these blockchains must be learned experimentally.

Discovery of the blockchain network topology typically happens through either passive or active methods. In the former method, one injects nodes in a diverse set of locations and records the nodes that reach out to connect. In the latter, one explicitly connects to a subset of nodes and uses some bookkeeping protocol messages to request the list of their neighbors. For instance, Ethereum has the `FINDNODE` message that

can be used for this purpose [82].

Using a combination of passive and active method, Gao *et al.* [82] measured the topology of the Ethereum blockchain in 2019 and found that it is a “small world” network — most nodes can reach any other node in a small number of hops relative to the network size. Their findings have later been confirmed by Wang *et al.* [83] using a similar but distinct measurement technique.

### Watts-Strogatz Network Model

One technique for modeling small-world networks is the Watts-Strogatz [84] model, which we utilized in Chapter 4. The parameters to Watts-Strogatz model are the network size  $|V|$ , node degree  $\overline{deg}$ , and rewiring probability  $p$ . In the base Watt-Strogatz model, the graph is generated as follows:

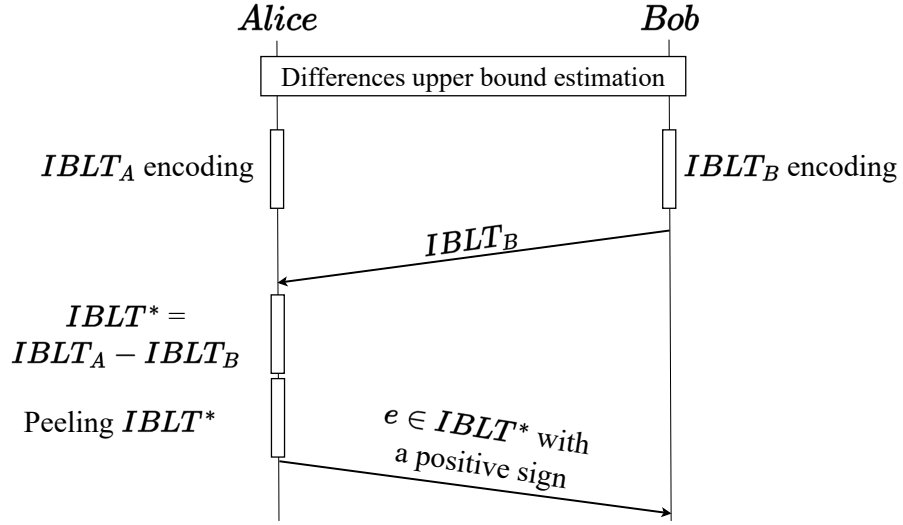
- Arrange  $|V|$  nodes in a circle and connect each node to  $\overline{deg}/2$  nearest neighbors on the left and  $\overline{deg}/2$  nearest neighbors on the right.
- For each node  $i \in V$  and each of its  $\overline{deg}/2$  rightmost neighbors  $j$ , “rewire”  $(i, j)$  with probability  $p$ . To “rewire” means to pick some  $k \neq i \neq j \in V$  uniformly at random and replace  $(i, j)$  with  $(i, k)$ . If edge  $(i, k)$  already exists, pick a different  $k$ .

## Chapter 3

# End-to-End Synchronization

End-to-end data synchronization protocols are the subcategory of data synchronization protocols that deal with two parties. Besides Alice and Bob, who are connected via only one communication channel, there are no other parties involved in the protocol. The assumptions of end-to-end synchronization are similar to those in the original formulation of the set reconciliation problem from Section 2.1 (*e.g.*, no prior context). In this section, we deal with the set reconciliation protocols for end-to-end synchronization.

As we outlined previously in Table 2.1, the set reconciliation protocols from the literature differ in various ways (*e.g.*, approximate set membership data structures versus coding theory). Therefore, choosing the right protocol for the envisioned application is a complex task that calls for an analytical as well as a methodical experimental analysis. Here we first describe the state-of-the-art set reconciliation protocols, analyze the boundaries of their practical applicability, and compare their core analytical properties. Subsequently, we analyze specific protocol assumptions in the light of practical constraints such as the knowledge of the exact upper bound on the number of mutual differences.



**Figure 3-1:** Protocol diagram of the IBLT-based set reconciliation.

### 3.1 The Structure of the State-of-the-Art Protocols

We describe and analyze three commonly utilized set reconciliation protocols: Invertible Bloom Lookup Tables-based (IBLT), Characteristic Polynomial Interpolation-based (CPI), and Cuckoo filter-based one. In particular, we describe the algorithmic components of each protocol and analyze how the protocols utilize their algorithmic components to solve the problem of set reconciliation defined in Section 2.1. Later in Section 3.2, we summarize these findings and compare the protocols with respect to their main analytical properties.

As discussed earlier in Chapter 2, there are many variations of these three base approaches. While each protocol variation is significant in its own merit, their analytical properties are often similar to those of the base approach. At the end of this section, we briefly discuss useful extensions to the protocols that we analyze.

#### 3.1.1 Invertible Bloom Lookup Table-based Set Reconciliation

The IBLT-based set reconciliation protocol [64] is primarily motivated by the compactness and decode efficiency of IBLTs, which we discussed earlier in Section 2.3.1.

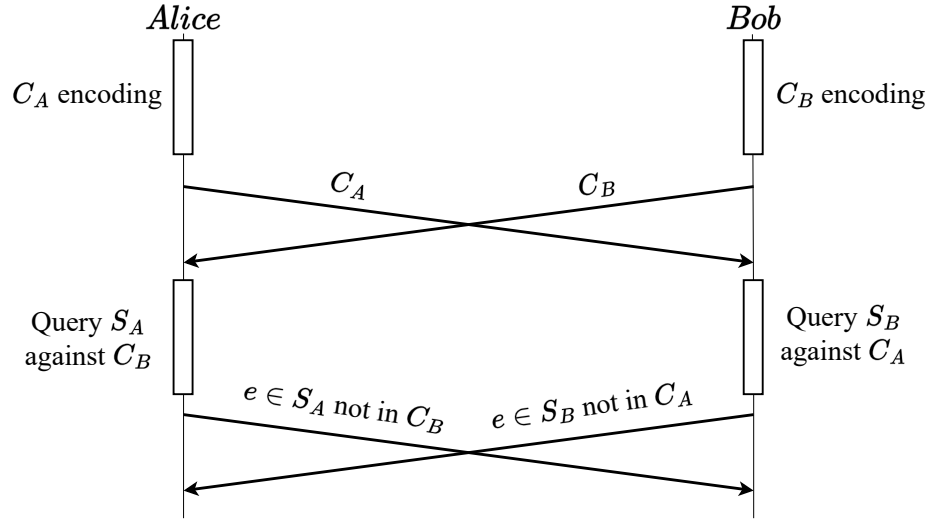
Given an accurate estimate on the maximum number of mutual differences, Alice and Bob use the estimate to construct two equally long  $IBLT_A$  and  $IBLT_B$ , and proceed as in Fig. 3-1. Bob sends  $IBLT_B$  to Alice and waits until Alice computes a new  $IBLT^* = IBLT_A - IBLT_B$ . At this point, Alice can learn Bob’s local elements by decoding  $IBLT^*$  and listing the elements with negative *counts*. On the other hand, the elements with the positive sign are local to Alice, and she should transmit them to Bob. Bob updates its local set and potentially sends a success status code back to Alice.

Note that only Bob’s IBLT needs be transmitted over the network, while Alice’s IBLT is only used locally. Hence, the communication cost of the IBLT-protocol is the size of  $IBLT_B$ . On the other hand, the computational complexity of the protocol is the sum of the computational complexities of decoding the two IBLTs, subtracting  $IBLT_B$  from  $IBLT_A$ , and decoding the resulting  $IBLT^*$ . Since all the IBLTs involved in the protocol are equally long and their size is proportional to the number of symmetric differences ( $d$ ), the communication cost of the protocol is  $O(d)$ . Likewise, the computational complexities of the encoding, subtraction, and “peeling” process are linear in the size of the IBLT, which yields the protocol’s computational complexity of  $O(d)$ .

The error rate of the IBLT-based set reconciliation protocol is equal to the probability of the decoding failure for  $IBLT^*$ , which we discuss separately in Section 3.3.4. When Alice fails to decode  $IBLT^*$ , she notifies Bob about her failure and Bob can re-initiate the protocol from scratch.

### 3.1.2 Cuckoo Filter-based Set Reconciliation

In contrast to the IBLT-based protocol, the Cuckoo [61] filter-based one does not require an estimate of the upper bound on the number of symmetric differences. This property makes the Cuckoo filter-based protocol particularly amenable for applica-



**Figure 3:2:** Protocol diagram of the Cuckoo filter-based set reconciliation. Alice keeps set  $S_A$ , Bob keeps set  $S_B$ .

tions that want to save on the rounds of communication, *e.g.*, due to a high network latency.

As depicted in Fig. 3:2, the protocol starts by Alice and Bob both encoding their sets into corresponding Cuckoo filters  $C_A$  and  $C_B$ . Each party can configure the parameters of the filter (*e.g.*, the number of slots per bucket and the size of the fingerprints) at their will. The parties then exchange the filters and query their local elements against the received filter. For each element that they do *not* find in the received filter, they put the element in the collection of elements that they will eventually send to the other party. For each element that is found in the filter, they conclude that the other party already has it (*i.e.*, they should not send).

The accuracy of the Cuckoo filter-based protocol is determined by the amount of false positives in the two filters. Since there is certain probability that a looked up element actually absent from the filter appears as present, the party that queries the filter may falsely conclude that the other party has certain amount of elements that it actually misses. This property of the protocol is often referred to as “false positives of

lookup, false negative of reconciliation” [61], where the false *negative* of reconciliation denotes an element that is mistakenly detected as common. The number of false *negatives* in the protocol is therefore bounded by the sum of the false *positive* rates of the two Cuckoo filters,  $\epsilon_{C_A} + \epsilon_{C_B}$ . On the other hand, there are no false *positives* of reconciliation, *i.e.*, all the elements that are being transmitted are actually missing at the other party. This is a direct consequence of the “no false negatives” property of the Cuckoo filters, which we further discuss in Section 3.3.6.

Furthermore, note that the sizes of  $C_A$  and  $C_B$  do not have to match, as opposed to the IBLT-based protocol. In the Cuckoo filter-based protocol, each party determines the size of its Cuckoo filter independently of the other and can make it proportional to its own set. For instance, if we are reconciling a set of one and a set of one million elements, the filter of the party with the smaller set can make its Cuckoo filter as small as the fingerprint size  $f$ . However, this is exactly the case when set reconciliation has the smallest advantage over transmitting the entire sets.

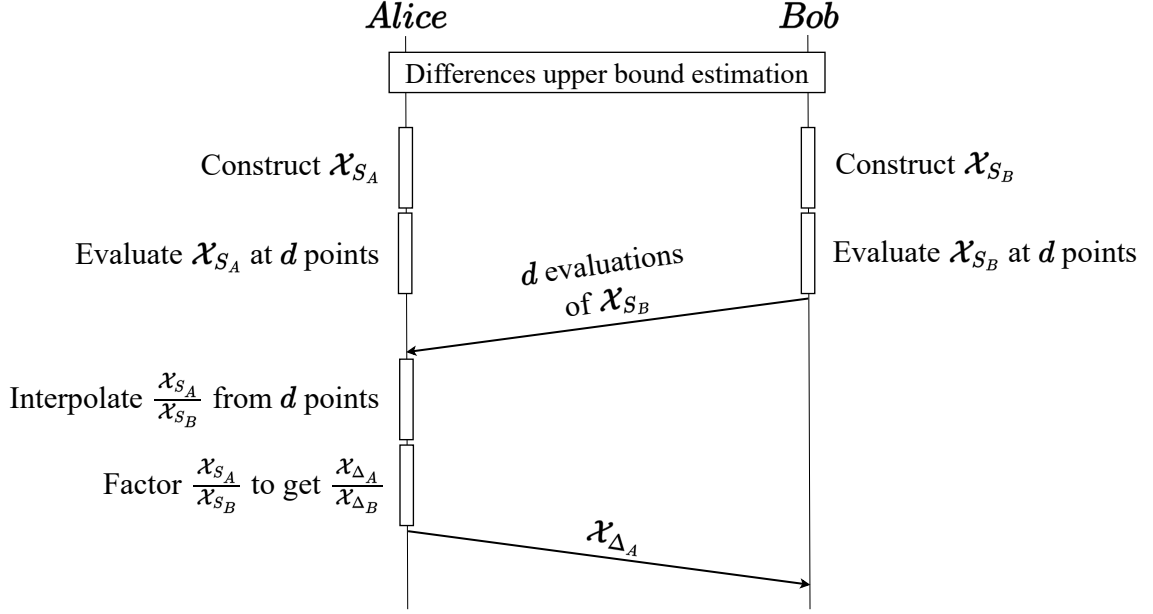
### 3.1.3 Characteristic Polynomial Interpolation-based Set Reconciliation

A characteristic polynomial of set  $S = \{x_1, x_2, \dots, x_n\}$ , denoted as  $\mathcal{X}_S(Z)$ , is the following univariate polynomial:

$$\mathcal{X}_S(Z) = (Z - x_1)(Z - x_2) \cdots (Z - x_n).$$

In other words, the set elements are represented as the roots of the characteristic polynomial. Given the characteristic polynomials of two sets  $S_A = \{x_1, x_2, \dots, x_n\}$  and  $S_B = \{y_1, y_2, \dots, y_m\}$ , we can construct the following rational function:

$$\frac{\mathcal{X}_{S_A}(Z)}{\mathcal{X}_{S_B}(Z)} = \frac{(Z - x_1)(Z - x_2) \cdots (Z - x_n)}{(Z - y_1)(Z - y_2) \cdots (Z - y_m)} = \frac{\mathcal{X}_{S_A \setminus S_B}(Z)}{\mathcal{X}_{S_B \setminus S_A}(Z)}.$$



**Figure 3-3:** Protocol diagram of the CPI-based set reconciliation. Alice keeps set  $S_A$ , Bob keeps set  $S_B$ .

This is a key fact that inspires characteristic polynomial interpolation-based set reconciliation protocols, and it is used in the following way (see Fig. 3-3). Given an upper bound on the number of mutual differences  $d$ , Bob can construct his characteristic polynomial  $\mathcal{X}_{S_B}(Z)$ , evaluate it in some  $d$  evaluation points, and send it to Alice. Alice can then evaluate her  $\mathcal{X}_{S_A}(Z)$  at the same  $d$  points and interpolate the resulting rational function. She can then factor the interpolated rational function to learn the elements local to herself and the elements local to Bob. Finally, she can send Bob the missing elements and update her set with the elements that are local to Bob, which she learned upon factoring the rational function and canceling the common terms.

### 3.1.4 Protocol Extensions

Some of the base protocols that we previously described can be extended to support slightly different usage scenarios. Here we outline the common variations of the set reconciliation problem and indicate which of the variations are supported by which

Algorithm Class	Multiset	Set of Sets	Prioritized	Multi-Party
CBF [60]	✓	✗	✗	✗
CPI [85]	✗	✗	✓ [71]	✓ [86]
BCH [69]	✗	✗	✗	✗
IBLT [64]	✓ [87]	✓ [88]	✗	✓ [89]
CF [61]	✓ [62]	✗	✗	✓ [63]

**Table 3.1:** Common set reconciliation approaches and their suitability for the variations of the set reconciliation problem. Some approaches support certain variations only via separately proposed techniques (reference here).

base protocol in Table 3.1:

**Multiset reconciliation:** each set element can have a *multiplicity*. Besides synchronizing the missing element identities, the protocol needs to synchronize their multiplicities. This variation of the set reconciliation problem is particularly useful in string reconciliation [90].

**Reconciliation of sets of sets:** the synchronizing sets consist of other *child* sets. The protocol needs to synchronize the child sets as well as any missing elements in the matching child sets. This variation is particularly useful for synchronizing graphs, databases, and collections of documents [91]–[93].

**Prioritized set reconciliation:** each set element has an assigned priority such that the set elements with higher priorities get synchronized first. This is particularly useful in disruption tolerant networks [94] (DTN) when we need to assure that the most important data has been synchronized prior to a disruption [71]. We further discuss this problem variation when CPI is used as the base protocol in Section 3.3.3.

**Multi-party set reconciliation:** a generalization of the set reconciliation problem to  $n > 2$  parties that keep sets  $S_0 \dots S_{n-1}$  and the goal is to achieve  $S_0 = \dots = S_{n-1} =$

Algorithm Class	Communication	Computation
IBLT [64]	$O(d)$	$O(d)$
CF [61]	$O( S_A  +  S_B )$	$O( S_A  +  S_B )$
CPI [85]	$(b + 1)d + b$	$O(d^3)$
CBF [60]	$O(d)$	$O( S_A  +  S_B )$
P-CPI [71], [85]	$O(d \cdot \bar{m}\eta b \cdot \log(\eta d))$	$O(d \cdot \eta b \bar{m}(\bar{m}^2 + k) \cdot \log(\eta d))$
BCH [69]	$d \cdot b$	$O(d^2)$

**Table 3.2:** Asymptotic communication and computation complexities for common set reconciliation approaches.  $d$  is the actual number of mutual differences between  $S_A$  and  $S_B$ .  $b$  is the constant size of set elements.  $\bar{m}, k, \eta$  are constants known in advance.

$\cup_{i \in \{0 \dots n-1\}} S_i$  [63], [86], [89]. The primary motivation for this problem variation are the distributed key-value stores such as Amazon’s Dynamo [95] and distributed storage systems such as Microsoft’s Azure Storage [96] that typically operate with many loosely consistent replicas. In Chapter 4, we consider a different but similar problem. Instead of focusing on extending the protocol to multiple parties, we use a generic end-to-end protocol over graphs in a pairwise fashion.

## 3.2 Analytical Performance Comparison

Based on the protocol descriptions in Section 3.1, we can compare the protocols’ analytical properties. In particular, we are interested in the asymptotic upper bounds on the *computational* and *communication* complexity. We define computational complexity under the standard RAM model [97]. The communication complexity of a set reconciliation protocol is the sum amount of bits transmitted by the protocol until Alice knows the elements local to Bob and Bob knows the elements local to Alice.

As discussed in Section 3.1.1, in the IBLT-based protocol, one party sends their IBLT filter to the other where the peeling process takes place. The peeling process then discovers the local elements of both parties. The side that received the IBLT can then finalize the synchronization process by sending the missing elements to

the party that sent the IBLT. Since there is only one IBLT being transmitted over the communication channel, and that filter is proportional in size to the number of symmetric differences  $d$ , the overall communication complexity of IBLT is  $O(d)$ . On the other hand, its computational complexity is dominated by the peeling process which is also  $O(d)$ .

In contrast to the IBLT-based protocol, in the Cuckoo filter-based one parties need to exchange their Cuckoo filters. Each party decodes the received Cuckoo filter to figure out what elements are not present at the sender's side. Since each filter is proportional to the size of the data set that it represents, the communication complexity of the Cuckoo-based protocol is  $O(|S_A| + |S_B|)$ . Its computational complexity is the same as communication because the receiving party needs to query each of its elements against the received filter to figure out whether the element is present.

The CPI-based protocol is from the coding theory-inspired family and thus differ structurally from the previous two. Its communication complexity counts the size of the evaluations of the characteristic polynomial, while its computational complexity counts the operations needed for constructing the characteristic polynomials, evaluating them in  $d$  points, and interpolating and factoring the resulting rational function. The dominant factor is function interpolation, and if solved through Gaussian elimination [85] takes  $O(d^3)$ . On the other hand, the communication cost is  $(b + 1)d + b$ , where  $b$  is the universe size.

The summary of the protocols' asymptotic bounds is given in Table 3.2, which also includes P-CPI that we discuss in Section 3.3.3, Counting Bloom filter-based (CBF) [60] protocol, and the one based on the Bose-Chaudhuri-Hocquenghem codes [69], [98]–[100].

### 3.3 Practical Implications of Protocol Assumptions

Beyond the common set reconciliation assumptions that we outline in Section 2.1.2 (*e.g.*, no prior context), certain protocols make additional assumptions that affect their performance in practice. In this section, we focus on several different assumptions that may affect the communication cost and message complexity of the protocols that rely on them. We begin with the problem of estimating the initial upper bound on the number of symmetric differences and describe a CPI extension that alleviates this problem. Then we proceed to the problem of estimating the failure probability of the IBF decoding process. Finally, we discuss certain constraints of Cuckoo filters with respect to choosing the hash functions for fingerprinting and calculating alternative buckets.

#### 3.3.1 Symmetric Difference Upper Bound

Most set reconciliation protocols require an upper bound on the number of symmetric differences to achieve the promised level of efficiency. However, the problem of estimating the degree of similarity between two sets (or analogously the extent to which they differ) can be considered independently from set reconciliation, and has attracted a significant attention from the research community. For instance, Feigenbaum *et al.* [101] proposed a technique for estimating  $L^1$ -difference ( $\sum_i |a_i - b_i|$ ) between the two general functions with values  $a_i$  and  $b_i$  being data streams. Cormode and Muthukrishnan [102] have initially focused on estimating the set sizes by constructing a hierarchy of data streams sketches, and Cormode *et al.* [103] and Schweller *et al.* [104] proposed a similar sketch-based techniques for detecting large differences among sets. Another technique introduced by Indyk and Motwani [105] consists of comparing random samples from the two sets. Broder [106], [107] proposed min-wise hashing, where  $k$  independent hash functions  $h_1 \dots h_k$  are chosen to

permute the elements in universe  $\mathcal{U}$  and represent the set  $S$  as an array of hashes of size  $k$ , denoted as  $M_S$ . The similarity between sets  $A$  and  $B$ , defined as  $r = \frac{S_A \cap S_B}{S_A \cup S_B}$ , is then estimated by counting the matching hashes in  $M_A$  and  $M_B$ . Given that the number of matching hashes is  $m$ , the estimated similarity is  $r = m/k$ . Although effective for comparing large sets with a low degree of similarity and relatively simple to implement, these techniques deteriorate in performance when the sets differ in only a few elements, which is often the case in practical set reconciliation.

The symmetrical difference estimation technique proposed in the original IBLT-based set reconciliation protocol is called *strata estimator* [64] and resembles the well-known technique of Flajolet and Martin for counting distinct elements in streams [108]. The crux of the idea is to stratify the space of the set elements into a logarithmic number of strata (*i.e.*, for a universe  $u = 2^{32}$ , we need  $\log(u) = 32$  strata), then construct a constant-sized IBFs to encode each strata. Suppose that we denote the strata as a vector  $E$  of size  $\log(u) + 1$ , the protocol for estimating the number of symmetric differences proceeds as follows. Alice exchanges her  $E_A$  with Bob who then constructs his own strata  $E_B$  and subtracts  $E_A[i]$  from  $E_B[i]$ , starting from  $i = \log(u)$  and ending with  $i = 0$ . For the strata difference (*i.e.*,  $E_B[i] - E_A[i]$ ) that decode successfully, Bob adds the number of recovered elements to his estimate of symmetric difference. If the decoding of strata fails at index  $i$ , Bob sets his estimate to the number of elements that he successfully recovered so far scaled by  $2^{i+1}$ . By a similar argument as in [108], this procedure assures that the actual number of the symmetric difference is less than what has been estimated. Compared to min-wise sketches and random sampling, strata estimator gives more accurate estimates when the actual number of symmetric difference is small relative to the size of the sets, while has a poor performance otherwise.

The most recent works propose techniques other than strata estimator, min-wise

sketches, and random sampling. For instance, Lee *et al.* [66] use a combination of *ternary* (TBF) [109] and invertible Bloom filters (IBF) instead of strata estimator. Since the summary size of the ternary and invertible Bloom filter is smaller than only the size of the invertible Bloom filter needed in the original protocol, they achieve a better communication cost. A somewhat similar approach with utilizing a small sketch in the first message to reduce the size of the sketches transmitted in the subsequent rounds is proposed by Ozisik *et al.* [23]. The difference is that Ozisik *et al.* use a regular Bloom filter instead of a ternary one.

On the other hand, there have been modifications to the IBLT-based set reconciliation protocol that mitigate the problem of establishing an initial upper bound on symmetric differences altogether. For instance, Lázaro and Matuz [65], [110] have proposed *multi-edge-type* IBLTs as a building block of a new fountain codes-inspired [111] IBLT-based set reconciliation protocol that can generate (and transmit) additional IBLT cells as they are needed in the “peeling” process, effectively achieving a competitive communication cost at the expense of adding message complexity.

In conclusion, learning an *exact* number of symmetric differences requires as much communication as learning the differences themselves [85]. However, there exist communication-efficient protocols for obtaining an upper bound that can then be supplied to the subsequent set reconciliation protocol. The techniques that allow for this kind of upper bound estimation typically rely on one of the following two techniques.

- (1) Utilization of auxiliary data structures in the initial round (*e.g.*, strata estimator, ternary Bloom filters, or traditional Bloom filter), or
- (2) Introduction of additional rounds of communication (*e.g.*, fountain codes-inspired approaches such as that of Lázaro and Matuz [65], Naumenko *et al.* [72], and partitioned CPI, which we discuss in Section 3.3.3).

Later in Section 5.3.1, we discuss the practical ramifications of trading-off communication cost for additional round trips.

### 3.3.2 Interactive Characteristic Polynomial Interpolation

CPI is nearly optimal in communication [85], but assumes a known, tight upper bound on the number of symmetric differences. As in the case of IBLT, there are CPI protocol variations that alleviate this constraint by trading it off for some increase in message complexity. The protocol that implements this trade-off for the CPI base protocol is called Interactive CPI (I-CPI) [68]. I-CPI belongs to the family of “divide-and-conquer” algorithms and relies on the recursive partitioning of the set element space (*i.e.*, the space of  $b$ -sized bitvectors). The bottom of the recursion is reached when the base CPI succeeds in reconciling some small portion of the set element space given a small constant  $\bar{m}$ .

Initially, the divide-and-conquer algorithm tries to reconcile the entire sets at once using a typically underestimated  $\bar{m}$  (when  $\bar{m}$  is overestimated or guessed correctly,  $\bar{m} \geq d$ , the algorithm terminates with a success). After any initial failure, the algorithm partitions the bitvector space into  $p$  non-overlapping sub-partitions,  $P_i, i \in \{1..p\}$ , and invokes CPI on each sub-partition. As the algorithms progresses, Alice and Bob independently keep an auxiliary data structure called  $p$ -tree, which represents the partitioning process. At any moment, each node in  $p$ -tree is in one of the three states (1) *active*; the sub-partition is currently being reconciled, (2) *terminal*; CPI invoked on this sub-partition has succeeded, (3) *inactive*; the reconciliation on this sub-partition has not been attempted yet. The divide-and-conquer algorithm terminates successfully once all sub-partitions are marked *terminal* (*i.e.*, the entire bitvector space is successfully reconciled). The correctness of I-CPI is rooted in the simple fact that for any pair of sets  $(S_A, S_B)$  and any non-overlapping partitioning,  $S_A \cap P_i = S_B \cap P_i$  for all  $i \in \{0..p\}$  implies that  $S_A = S_B$ . Note that each *active*

an *terminal* partition requires one more round of communication between Alice and Bob, which makes this divide-and-conquer approach interactive.

We want to stress here that the divide-and-conquer approach is partially allowed by CPI’s ability to recognize the reconciliation failure, thus instructing the algorithm to further partition the space of elements. Not all state-of-the-art set reconciliation approaches allow for signaling reconciliation failures. For instance, the Cuckoo-based protocol that we described in Section 3.1.2 does not support failure signaling. That is, after completing the Cuckoo-based protocol, Alice or Bob may still miss certain elements from the other party due to the possible false negatives of reconciliation, and the protocol cannot detect this state on its own. One way to add the support for failure signaling to the protocols such as the Cuckoo-based one is to exchange some small, fixed-size checksums, at the end of the protocol [112], [113], concluding that there are more differences to reconcile if the checksums do not match. Note that this method would only work with its own error probability dependent on the set sizes.

### 3.3.3 Characteristic Polynomial Interpolation with Priorities

Having the space of set elements partitioned according to some criteria, such as the case with I-CPI that we described above, allows for imposing *priorities* to the set elements [68], [71]. By doing so, we can assure that the elements with a higher priority get reconciled first, which is particularly useful when the reconciliation process runs over a disruptive link (*i.e.*, the communication channel that can be cut at any time). In that regard, prioritized reconciliation assures that at the moment of the link breakage, the following holds (1) if *no* lower priority elements are transmitted, then all the elements that have been transmitted are of a high priority, and (2) if *some* lower priority elements are transmitted, then all high priority elements have already been transmitted.

Given a set of elements with assigned priorities, we can partition the set such

that all elements with the same priority form separate partitions. We can then run the divide-and-conquer algorithm of I-CPI on the sequence of partitions sorted in the descending order of their priorities. Denoting the proportion of the highest priority elements as  $\eta$ , it is easy to see that for  $\eta = 1$  we exactly have I-CPI, meaning that the prioritized set reconciliation (P-CPI) is a generalization of I-CPI. Based on the results from [68], denoting the number of symmetric differences between the sets as  $m$ , the maximum number of CPI invocations is

$$I(m) \leq 1 + \frac{m}{\bar{m}} p [b \log_p(2)] \quad (3.1)$$

Since one invocation of CPI requires the computation of  $\Theta(b\bar{m}^3 + b\bar{m}k)$ , we have that the worst case computational complexity of P-CPI is

$$\Theta\left(m\bar{m}^2 b^2 \frac{p}{\log p}\right).$$

Given that the worst case communication complexity of CPI is  $\Theta(mb)$ , the worst case communication complexity of P-CPI is

$$\Theta\left(m \frac{p}{\log p} b^2\right).$$

However, under the assumption that the set differences are distributed uniformly over  $GF(2^b)$ , Jin *et al.* [71] give the following theorem:

**Theorem 1:** When there are two sets with  $\eta m$  uniformly distributed symmetric differences, P-CPI reconciliation makes  $O(\eta m \log(\eta m))$  invocations to CPI, with probability at least  $1 - \frac{1}{\eta m}$ .

As a corollary, the computational complexity of P-CPI is  $O(m \cdot \eta b \bar{m} (\bar{m}^2 + k) \cdot \log(\eta m))$  with high probability. To generalize and clarify the conclusions of Jin *et al.* [71], we give a revised proof [42] of the above theorem.

*Proof:* Without the lost of generality, we consider a binary partition tree ( $p = 2$ )

and write  $m' = m'_0 = \eta m$  to denote the overall number of the high-priority differences. We call a node in the partition tree *good* if each of its children contains at least one third of its differences. Otherwise, the node is *bad*. If we now consider a  $t$  nodes long root-to-leaf path that consists of only good nodes, we will see that the number of differences at level  $t$  is

$$m'_t \leq \left(\frac{2}{3}\right) m'_{t-1} \leq \left(\frac{2}{3}\right)^t m'_0 \quad (3.2)$$

Since we know that each good node must contain at least one difference, we can use Eq. (3.2) to derive the maximal number of good nodes in any path as

$$t \leq \frac{\log_2 m'}{\log_2 \left(\frac{3}{2}\right)} < 2 \log_2 m' \quad (3.3)$$

The next step in our proof is to show that the following holds for any root-to-leaf path  $P$  that may contain both bad and good nodes

$$Pr[|P| > 4 \log_2 m'] < \frac{1}{m'^2} \quad (3.4)$$

Let  $X_i$  be a random variable that takes value 1 when the  $i$ -th node on path  $P$  is bad, and 0 otherwise. If we denote the number of differences at node  $i$  as  $s_i$ , then by the definition of good nodes we have

$$Pr[X_i = 1] = 1 - \frac{\sum_{j=s_{i-1}/3}^{2s_{i-1}/3} \binom{s_{i-1}}{j}}{2^{s_{i-1}}} \leq \frac{2}{3} \quad (3.5)$$

Now, let  $X$  be the random variable that denotes the number of bad nodes along  $P$ . Since all  $X_i$  are independent, we can use Eq. (3.5) and the union bound to derive

$$E[X] = \sum_i^{|P|} X_i \leq \frac{2}{3} |P| \quad (3.6)$$

Then, we assume  $|P| \geq \frac{3}{2e} \log_2 m'$ , and apply the Chernoff-Hoeffding bounds for

some  $q > 2eE[X]$  as follows [114]:

$$\Pr[X > q] \leq 2^{-q} \leq 2^{-2 \log m'} = \frac{1}{m'^2} \quad (3.7)$$

Since the length of any path is the sum of its good and bad nodes (*i.e.*,  $|P| = X + t$  by definition), we can combine Eq. (3.3) and Eq. (3.7) to show that Eq. (3.4) holds for all paths  $P$

$$\Pr[X > q] = \Pr[|P| > 4 \log m'] \leq \frac{1}{m'^2} \quad (3.8)$$

Importantly, for  $|P| < \frac{3}{2e} \log_2 m'$ , we have that Eq. (3.4) is trivially satisfied as  $\Pr[|P| > 4 \log_2 m']$  is zero.

Next, we see that our partition tree can have at most  $m'$  leaves, since each leaf must contain at least one difference. Hence, the maximal number of root-to-leaf paths is  $m'$ . We can apply the union bound to prove that all the root-to-leaf paths will be shorter than  $4 \log_2 m'$  nodes with probability  $1 - \frac{1}{m'}$  as follows

$$\begin{aligned} \Pr[\exists P, |P| > 4 \log_2 m'] &\leq m' \cdot \Pr[|P| > 4 \log_2 m'] \\ &\leq \frac{1}{m'} \end{aligned} \quad (3.9)$$

Finally, P-CPI calls CPI at each node of the partition tree. Thus, the overall number of CPI invocations is  $O(m' \log m')$ . ■

Note that in practice we usually have that  $m \ll 2^b$ , thus the worst-case number of CPI invocations of  $O(mb)$  (see Eq. (3.1)) is worse than  $O(m \log m)$  CPI invocations in the high-probability case from Theorem 1.

Combining the high-probability upper bound on the number of CPI invocations from Theorem 1 and the worst-case computation complexity of CPI from [68], we get that the computation complexity of P-CPI is  $O(m \cdot \eta b \bar{m} (\bar{m}^2 + k) \cdot \log(\eta m))$ , with

probability  $1 - \frac{1}{\eta m}$ . By the same token, the communication complexity is  $O(m \cdot \bar{m} \eta b \cdot \log(\eta m))$  with the same probability. Importantly,  $\bar{m}$  is a fixed constant known in advance.

### 3.3.4 “Peeling” Process in Invertible Bloom Filters

The IBF decoding process (*peeling*) is of a probabilistic nature and can fail with an arbitrary small probability [87], [115]. This is particularly important in the IBLT-based set reconciliation protocols because the synchronizing parties must agree on the IBLT parameters to ensure that the  $IBLT^*$  obtained by subtracting  $IBLT_B$  from  $IBLT_A$  can be decoded with a sufficiently high probability. Remember that by the definition of IBLT subtraction operation [64],  $IBLT_A$ ,  $IBLT_B$ , and  $IBLT^*$  are of the same size, and what we essentially want to peel are the symmetric differences, not necessarily the entire sets. In other words, we do *not* require  $IBLT_A$  and  $IBLT_B$  to recover their corresponding sets, but rather their difference  $IBLT^*$  to recover the set of symmetric difference, which is no larger than some known  $d$ . Therefore, the problem of IBLT parameterization in the IBLT-based set reconciliation protocols can be formulated as follows.

**Problem statement:** Given an upper bound on the number of symmetric differences  $d$ , the universe from which the set elements are drawn  $\mathcal{U}$ , and an arbitrarily small error probability  $\epsilon$ , find the minimum number of cells  $m$  that the  $IBLT^*$  should have for its peeling process to fail with a probability not larger than  $\epsilon$ .

First results related to the above problem were given by Goodrich and Mitzenmacher [87] who bounded the probability of the peeling failure by  $O(t^{-k+2})$  when we choose  $m > (c_k + \epsilon)t$ , where  $t$  is a threshold number of items inserted in the IBLT,  $k$  is the number of hash functions used by the IBLT, and  $c_k > 1$  is a constant. Their results are based on the similarity between the problem of finding the 2-core of a random hypergraph and peeling an IBLT [116], [117].

However, it turns out that the choice of an optimally small  $c_k$  depends on both  $k$  and  $\epsilon$ , which has motivated the authors of popular practical IBLT implementations to use various heuristics in calculating the optimal IBLT parameters. For instance, Levine [23], [118], [119] defines the “hedge” factor  $h = \frac{m}{d}$  and proposes a brute-force technique that searches for an  $(h, k)$  pair that minimizes the size of the resulting IBLT for the given  $d$  and  $\epsilon$ . For the purpose of their work in [23], Ozisik *et al.* used the technique of Levine and published a database of precomputed optimal  $(h, k)$  pairs for  $d < 1000$  and  $e = \frac{1}{240}$ , which are the parameters suited for their concrete application of *Graphene*, a Bitcoin block propagation protocol.

While running a brute-force search over the space of all possible  $(h, k)$  pairs for the given  $d$  and  $\epsilon$  is certainly impractical as a component of a set reconciliation protocol, precomputing a table of optimal  $(h, k)$  pairs can help in the application scenarios where we can impose a fixed choice of possible error rates and expect only some small subset of possible count of symmetric differences. Otherwise, the best we can do is to select some fixed  $(h, k)$  pairs that succeed with a high probability for a wide range of practical counts of symmetric differences. While  $k$  may not significantly affect the performance of a practical set reconciliation protocol, the hedge factor  $h$  does affect the communication cost. In several practical implementations,  $h$  is set to a value between 1.5 and 2, and somewhere between 4 and 8 hash functions are used [64], [119]. On the one hand, these constraints make a protocol generally applicable with respect to the number of symmetric differences. On the other hand, they may reduce the protocol’s communication efficiency, especially in comparison with the communication-optimal protocols such as CPI. Specifically, the IBLT-based ones may transmit from 50 to 100% more data than their CPI-based counterparts. While this difference may not seem significant from a purely theoretical standpoint (*i.e.*, both are  $O(d)$  in communication), it does show up in practice, as we demonstrate

through experiments in Section 5.3.4.

Another peeling-related constraint of IBLTs that can affect the practical performance of set reconciliation is that even for largely oversized IBLTs (*i.e.*,  $m \gg d$ ) peeling failure rate may still affect some applications. Mizrahi *et al.* [120] have recently demonstrated that a classical IBLT with parameters  $m = 64, k = 4$  representing a set from universe  $|\mathcal{U}| = 381$  and having 6 actually inserted elements fails with probability larger than  $4 \times 10^{-4}$  (*i.e.*, 4 times in 10 thousand tries). Therefore, the same group of authors proposed an alternative IBLT construction called *IBLT with listing guarantees* resembling the technique used in *Bloom filter with a false positive free zone* [121], which allows for deterministic peeling given a bound on  $d$  and the universe from which the set elements are chosen. We believe that this alternative IBLT construction can be particularly useful in applications that require success guarantees and can afford to sacrifice some communication efficiency.

Related to the above problem statement is the work of Kubjas and Skachek [122], where the authors analyzed the success probability of *partial* element extraction using Tanner graphs [123]. Application that do not aim at synchronizing the entire set of symmetric differences but a portion of it can use the construction of Kubjas and Skachek to improve the communication efficiency of the IBLT-based protocol.

### 3.3.5 Set Reconciliation Protocol Symmetry

Note that the IBLT-based set reconciliation is *asymmetrical* in the sense that the most computation is done by Alice, while the protocol metadata (*i.e.*, Bob’s IBLT) is transmitted only in one direction, from Bob to Alice. To the best of our knowledge, we are the first to acknowledge and analyze the problem of symmetry in set reconciliation protocols [42].

In practice, Alice and Bob may run different hardware (*e.g.*, an IoT sensor versus a high-end cellphone), or have significantly different compute capacities allotted

for the sync process (*e.g.*, CPU cycles). As we discuss later in Section 5.3.3, the difference in the compute capabilities of the nodes involved in synchronization may affect the performance of asymmetrical set reconciliation protocols such as IBLT to the extent that we would be better off using a different data sync approach (*i.e.*, there is a different state-of-the-art protocol that achieves notably better performance under the same data parameters). In other words, although IBLT may be a good approach for the symmetrical compute scenario (*i.e.*, Alice and Bob have comparable compute capacities allotted to the sync process), it may be a very bad choice under asymmetrical compute (*e.g.*, when Alice runs another computationally expensive task or wants to allocate only a fraction of its compute capacity to the sync process).

A solution to the compute asymmetry problem would be for the two parties to run an “introductory” round before the set reconciliation protocol in which they would learn which party has the better compute capabilities at the moment. Using this information, the parties can pick the more capable one to play the role of Alice, thus assuring that they get the best possible performance. However, given the standard assumptions of the set reconciliation problem (*e.g.*, no prior context), implementing an introductory sub-protocol could be challenging due to (1) an increased *message complexity*, and (2) *security* concerns. To learn which one is more computationally capable, the two parties either need to compute a known challenge and exchange the duration it took them to complete it, or they need to exchange their platform specifications (*e.g.*, CPU identifiers) and consult a third party for comparison. In both cases, the overall message complexity of the protocol increases for at least one message. When it comes to system security, exchanging the exact platform specification or even fine-grained timing information for a known task may help a malicious actor to detect security-critical properties of the device (*i.e.*, inadvertently facilitate adversarial reconnaissance [124]–[126]).

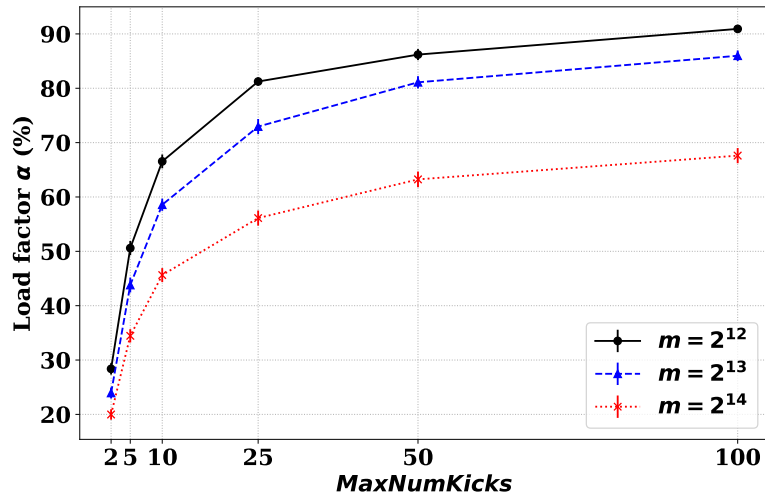
On the other hand, if we drop the assumption of no prior context, and adopt a new assumption that the relative compute power of participants does not change over time, then the asymmetry problem can be addressed in the following way. Starting with a random party playing the role of Alice, in the next protocol iteration the role of Alice swings to the other party. After some threshold number of protocol invocations, Alice can compute the average performance in the two cases and retain the role of Alice, or stand down in favor of the other party. Under the given assumption, this protocol will eventually converge to the optimal choice of Alice and Bob roles.

### 3.3.6 Hash Function Implementation for Cuckoo Hashing

As we pointed out earlier in Section 2.3.2, the cuckoo filter family of probabilistic data structures is designed to permit certain small amount of false *positives*. Precisely, a lookup query must return the correct answer for all inserted elements, but *not* for all elements that are not inserted. The amount of non-inserted elements that the structure is permitted to declare falsely present is bounded by  $\epsilon$ , the false positive rate. Importantly, the Cuckoo filter construction does *not* permit any false *negatives* — actually inserted elements that cannot be found. The central benefit of Cuckoo filters over other similar approximate set membership data structures (*e.g.*, traditional Bloom filters) is that Cuckoo filters achieve better space efficiency for the fixed false positive rate, especially for very small  $\epsilon$  (*e.g.*,  $< 3\%$ ).

The insertion operation, which makes Cuckoo filters surpass the space efficiency of Bloom filters, is more complex to implement than for traditional Bloom filters. The two main challenges in this regard are:

- (1) Maintaining the no-false-negatives property (*i.e.*, the Cuckoo hashing idempotence defined in Section 2.3.2), and
- (2) Achieving a high space efficiency.



**Figure 3.4:** Average load factor  $\alpha$  for cuckoo filters of sizes  $2^{14}$ ,  $2^{15}$ ,  $2^{16}$  ( $b = 4$ ,  $f = 12$ ), over 100 experiments.

In the rest of this section, we tackle these two challenges and find that popular implementations [127]–[129] often make trade-offs between the two — they are either space efficient and incur certain amount of undesirable false negatives, or sacrifice space efficiency to an extent that makes them less space efficient than traditional Bloom filters. It is important to stress that introducing false negatives makes these practical implementations inadequate for the applications that require false negatives-free data structures (*e.g.*, privacy-preserving authentication [130]), while suboptimal use of space practically erases their advantage over the main competitors (*i.e.*, traditional Bloom filters).

### Achieving a High Load Factor

One of the main properties of Bloom filters and its successors is a constant insertion time. In Cuckoo filters, this is achieved through bounding  $MaxNumKicks$  to a constant. While in the Bloom filter-based alternatives, the constant insertion time is rooted in the fact that we use some constant number of independent hash functions

to mark  $k$  buckets, in Cuckoo filters, we recursively attempt to place the element that is being inserted into no more than  $MaxNumKicks$  buckets. At the end, an item will be placed in only one bucket, or fail to insert. The probability of failure is related to the size of the filter and its current load [77], [131]–[133]. Therefore, the goal is to maximize the load factor while minimizing  $MaxNumKicks$ . In Fig. 3-4, we plot the average load factor  $\alpha$  for various  $MaxNumKicks$  and compare filters with different sizes. For each Cuckoo filter configuration, we keep inserting elements until the first failure happens. The ratio between the number of buckets  $m$  and currently inserted elements  $n$  gives the load factor  $\alpha$ . The elements that we insert are uniformly distributed in  $GF(2^{64})$  and we repeat the experiment for each filter configuration 100 times.

The first observation is that for all filter sizes the average load factor increases with the number of maximally permitted cuckoo evictions ( $MaxNumKicks$ ). However, to achieve densely populated filters (*i.e.*, minimize the amount of unused buckets), we need to set  $MaxNumKicks$  to a fairly high value compared to practical Bloom filter implementations that often use only several hash function ( $k$ ). The larger the filter, the higher we need to set  $MaxNumKicks$  to achieve a sufficiently high load factor (*e.g.*, 90%). Therefore, setting  $MaxNumKicks$  to the same constant across all possible filter sizes may lead to sparsely populated cuckoo filters, which bloats the communication cost in the corresponding set reconciliation protocol.

### False Negatives In Cuckoo Filters

The main property of Cuckoo hashing is that alternative buckets can be calculated using only the fingerprint and the current bucket it is being placed into. This property allows the cuckoo filters to relocate the elements from heavily loaded buckets into more sparsely populated ones, and most importantly, minimize the amount of unused space. However, the popular cuckoo filter implementations [127]–[129], at the

moment of writing, only handle the filters whose size (in buckets) is a power of two. We refer to this constraint as the “power of two” requirement. For all the other requested maximum capacities ( $n_{max}$ ), traditional cuckoo filters waste a large amount of space, which further reduces the communication efficiency of the corresponding set reconciliation protocols. In parallel with us [50], the negative ramifications of the power of two requirement have been discussed by Wang *et al.* [134], who proposed alternative Cuckoo construction called Vacuum filter. Following these results, several other authors analyzed the power of two requirement and proposed alternative filter constructions, not necessarily relying on Cuckoo hashing [73], [135]–[143]

In the rest of this section, we argue that removing the power of two requirement from cuckoo filters is not a trivial task and show that the naive approach introduces generally undesirable false *negatives*. Contrary to the false positives, false negatives are not the desired property of approximate set membership data structures. We formulate our claims in the form of the following two proposition.

**Proposition 1** (*naive removal*): Suppose that we remove the power of two requirement while keeping the hashing procedure from the original construction [77], then the *Cuckoo hashing idempotence* is violated.

Consider the Cuckoo filter of non-power-of-two size  $m = 5$ , which uses 7 bits per fingerprint, and makes each bucket 4 fingerprints wide. In such a filter, we choose to insert an element  $x = 759$  and pick the bucket and fingerprint hashes as in Table 3.3. The insertion process then proceeds as follows. First, we calculate the two candidate buckets as in Eq. (2.1) and obtain  $i_1 = 1, i_2 = 0$ . Suppose that  $i_1$  is full and  $i_2$  is not. Therefore, we place  $\phi_x = 119$ , the fingerprint of  $x$ , into bucket 0. Note that after we inserted  $x$  in bucket 0, some elements from bucket 1 may get deleted. Suppose now that we insert some other  $x_o \neq x$  with one alternative bucket being 0 and the other being full. In accordance with the cuckoo hashing principle, we then randomly

<b>f</b>	<b>b</b>	<b>m</b>	<b>x</b>	$H(x)$	$\phi_x = x_f(x)$	$H(\phi_x)$
7	4	5	759	1	119	4

**Table 3.3:** Counterexample used in Proposition 1.

choose the victim to evict from bucket 0, meaning that  $\phi_x$  (the fingerprint of  $x$ ) can get selected for eviction. If so happens, we need to calculate the alternative bucket of  $\phi_x$  starting from bucket 0. That is, we substitute  $i_c = 0$  and  $\phi_x = 119$  in Eq. (2.2) and obtain  $i_a = 4$ . The fact that bucket 4 differs from both  $i_1 = 1$  and  $i_2 = 0$  means that the idempotence of Cuckoo hashing is violated.

On the other hand, we can easily see why idempotence holds for powers of two by substituting  $i_1$  and  $i_2$  from Eq. (2.1) into Eq. (2.2):

$$H(x) \equiv \left( \left( H(x) \bmod m \right) \oplus H(\phi_x) \bmod m \right) \oplus H(\phi_x) \bmod m. \quad (3.10)$$

The above identity holds whenever  $m = 2^k$  for some  $k$ , because any number in modulo  $2^k$  is represented as its  $k - 1$  least significant bits. Therefore, the identity from Eq. (3.10) is transformed to:

$$H_{[k-1..0]}(x) = H_{[k-1..0]}(x) \oplus \underbrace{H_{[k-1..0]}(\phi_x) \oplus H_{[k-1..0]}(\phi_x)}_{\text{Cancels out}}.$$

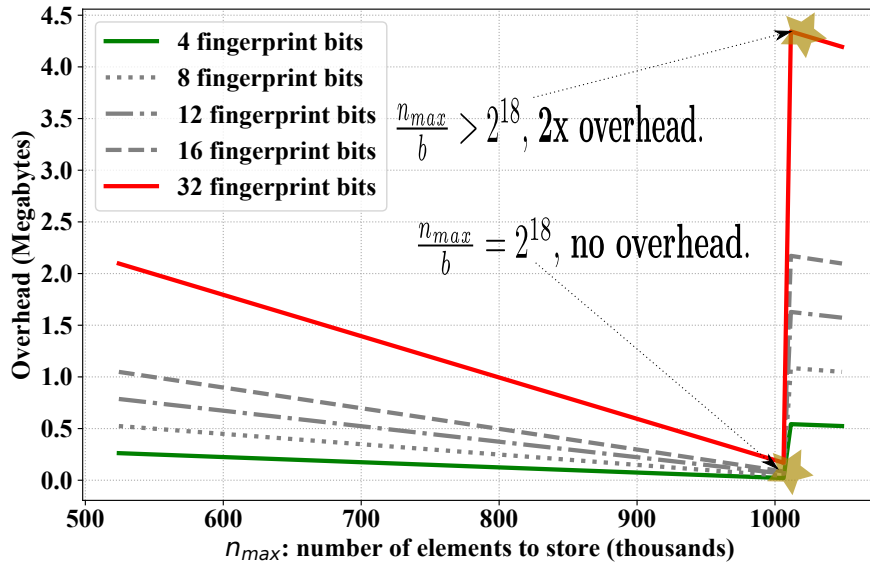
**Proposition 2 (false negatives):** When the idempotence of Cuckoo hashing is violated, then there can be false negatives.

We can use the example from Table 3.3 to demonstrate that an eviction to the wrong bucket can lead to false negatives. Consider the scenario in which  $\phi_x$  from Table 3.3 is evicted in the wrong bucket 4 and there is no  $x'$  for which the following conditions hold:

$$\begin{aligned} \phi_{x'} &= \phi_x, \text{ and} \\ H(x') &= H(x) \pmod{m} \end{aligned} \tag{3.11}$$

That is, there is no item  $x'$  that collides with  $x$  on both fingerprint and the bucket hashes. In this case, the lookup procedure will traverse  $i_1 = 1$  and  $i_2 = 0$  searching for  $\phi_x$ . Since  $\phi_x$  is not in any of  $i_1$  and  $i_2$ , and since  $i_1$  and  $i_2$  do not store any  $\phi_{x'}$  from Eq. (3.11), the lookup procedure for  $x$  will return a negative answer (*i.e.*, item not found). However,  $x$  was successfully inserted, and  $\phi_x$  is still present in the filter, meaning that this answer is a false negative.

To preserve the idempotence of the Cuckoo hashing, the referential implementation of Cuckoo filters [77] relies on the *multiply-shift* hashing scheme of Dietzfelbinger [144], which itself assumes hashing into  $2^k$  bins. However, if we constrain  $m$  to only powers of two, we reduce space efficiency for all the practical applications that require filters of other sizes. That is, instead of an optimally sized filter of size  $\lceil C * n_{max} \rceil$  bits, we get a filter of size  $b * \bar{m} * f$ , where  $\bar{m}$  is the closest power of two greater than or equal to our targeted  $m$ . This discrepancy is shown in Fig. 3-5, where we calculate the overhead as the difference between the actual memory footprint of the filter and the space needed to store  $n_{max}$  items given  $b$  and  $f$ . The x-axis spans linearly from  $2^{19}$  to  $2^{20}$ , excluding both ends. We evaluate five different cuckoo filters each representing items using fingerprints of 4, 8, 12, 16, and 32 bits. We observe that the overhead is at its minimum when  $n_{max} = 1006209$  and at its maximum when  $n_{max} = 1011505$ , for all  $f$ 's. The main reason for  $n_{max} = 1011505$  to be minimum of overhead is its proximity to  $2^{20} * 0.96$ . This number is important because Fan *et al.* [77] have shown that Cuckoo filters with  $b = 4$  achieve their asymptotic maximum of load factor  $\alpha$  at 96%. For that reason, multiple implementations [127]–[129] that bind their  $b$  for all filters to 4 do not allow for larger load factors than 96%. The exact

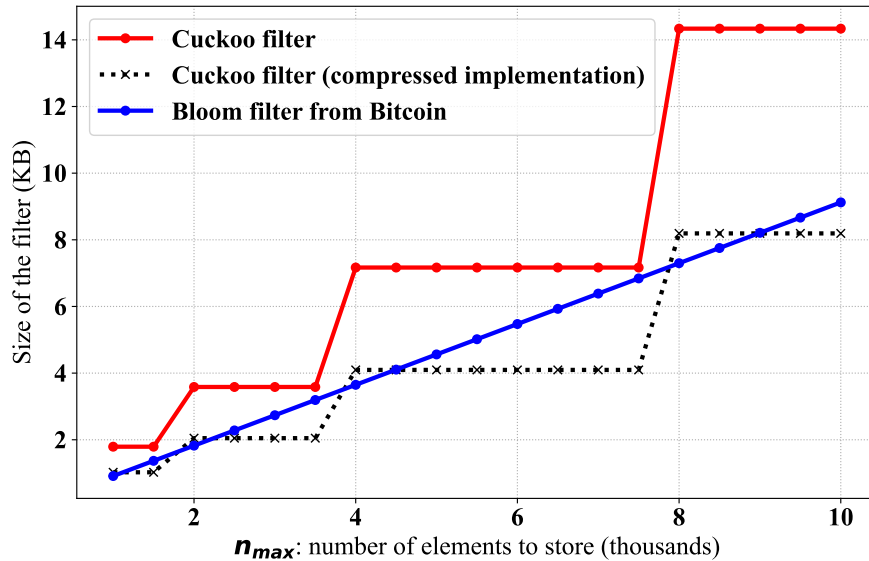


**Figure 3-5:** Space overhead as a function of  $n_{max}$  for  $b = 4$  (referential implementation [77]).

method used here is to double the size of the cuckoo filter when  $n_{max}$  would occupy more than 96% of  $b * m$  slots. As a consequence, we observe the peak of overhead immediately after this 96% load factor boundary.

To emphasize the significance of this space overhead, we compare the cuckoo filter implementations [127]–[129] to the Bloom filter implementation from Bitcoin Core [145]. Fig. 3-6 shows that due to the  $m = 2^k$  constraint in the Cuckoo filter implementations, a space-optimized Bloom filter implementation achieves better space efficiency for many values of maximal number of items  $n_{max}$ . The filter sizes for which cuckoo filters achieve superior space efficiency are exactly those that immediately precede  $m = 2^k$ .

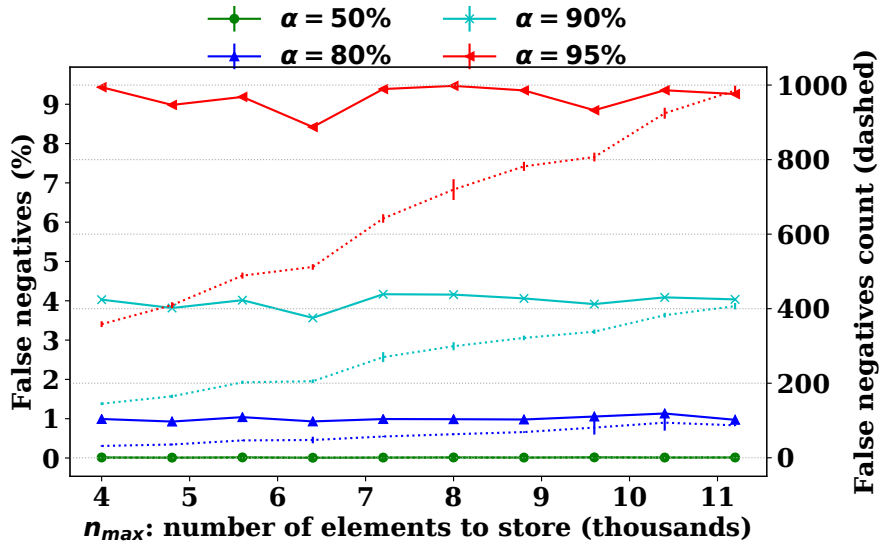
In our comparison, we set both Bloom filters and the “semi-sorting” [127] enabled cuckoo filters for false positive rate  $\epsilon \leq 3\%$ . “Semi-sorting” optimization of Cuckoo filters allows for improving space efficiency by compressing buckets. It is achieved by precomputing every possible combination of bits in a single bucket and storing them



**Figure 3-6:** Space utilization of the referential Cuckoo filter implementation [77] and the Bloom filter implementation from Bitcoin Core [145].

in a separate lookup table. The main part of the filter then stores the index of the lookup table instead of the buckets themselves. For example, when  $f = b = 4$  the compression saves one bit per item. In Fig. 3-6 we show both the total memory used by cuckoo filters (including the lookup table), and the memory used only by the main part of the filter (*compressed implementation*).

In accordance with our Proposition 2, the naive removal of the power of two constraint to reduce the space overhead causes false negatives. In Fig. 3-7, we show the empirical false negatives rates that emerge when we remove the power of two requirement. We construct the filters to target  $\epsilon \leq 3\%$  and thus set  $b = 4$  and  $f = 8$ . We evaluate ten different sizes of the filter starting from 4000 and increasing the size by 200. For each filter we insert as many items (sampled from a uniform distribution) as needed to achieve the given load factor. The successfully inserted items are recorded. When the given load factor is achieved, we execute a lookup query for each one of successfully inserted items. Those items that are not found in



**Figure 3-7:** Empirical false negatives rate for  $b = 4, f = 8$  cuckoo filters, with  $n_{max}$  not a power of two.

the filter, although successfully inserted, are considered as false *negatives*. We repeat each experiment 100 times and report the mean values with 95% confidence intervals.

## Chapter 4

# Network-Wide Synchronization

As opposed to end-to-end synchronization that we discussed in Chapter 3, in the problem of network-wide synchronization we view the entire network of synchronizing nodes as a single entity. Therefore, we are particularly interested in the effect of the network topology and the statistics of the data that is being synchronized to the overall communication cost and time to synchronize the entire network.

As an example application, we consider the problem of network-wide synchronization of blockchain transaction pools (see Section 2.4.1). The great potential of syncing transaction pools for improving block propagation delay has recently been shown through *in-situ* measurements in Bitcoin network [43]. As we discussed earlier in Section 1.1, the more block transactions nodes have in common, the faster the block flows through the network and reaches the majority. In effect, the improvement in block propagation delay has a positive impact on the blockchain confirmation times and its overall security.

While the transaction pool synchronization approaches from the literature either mingle transaction pool synchronization into block propagation protocols [23], [67], or use heuristics with no quantifiable guarantees [43], we take a different approach. We aim at maintaining the transaction pool synchronization outside and independently from the main block propagation channels. In the process, we formalize the network-wide synchronization problem within a graph theoretic framework and introduce ***SREP*** (*Set Reconciliation-Enhanced Propagation*) — a novel distributed

algorithm for transaction pool synchronization with quantifiable guarantees. We analyze *SREP*'s performance in various realistic network topologies and show that it converges on any connected graph in the number of steps that is bounded by the *diameter* of the network. We confirm our analytical findings through extensive simulations, which includes the comparison with *MempoolSync*, a recent approach from the literature. Our simulations show that *SREP* incurs reasonable overall bandwidth overhead and, unlike *MempoolSync*, scales gracefully with the number of the participants in the network. To the best of our knowledge, our simulation technique is unique in its ability to incorporate the real-world transaction pool data and simulate transaction pool synchronization over the networks of realistic sizes (*i.e.*, tens of thousands of nodes).

#### 4.1 *SREP* Algorithm

The core of *SREP* is a concept that we denote as ***primal sync*** — an end-to-end set reconciliation protocol with communication complexity *linear* in the number of symmetric differences. We have seen in Chapter 3 that there are several concrete end-to-end set reconciliation protocols with this property. For instance, CPI [68], BCH [69], and various IBLT-based [64], [65], [120] approaches can be used as the primal syncs in *SREP*. The main idea behind *SREP* is to invoke the primal syncs for each neighbor in parallel and incorporate the newly arrived elements at the end of each round. The primal syncs invoked by *SREP* operate on the transaction pools viewed as a set of globally unique transaction hashes [146]. In this regard, *SREP* is similar to the approaches from the literature such as Graphene [23], Erelay [72], Gauze [67], and *MempoolSync* [43].

One way to support many parallel invocations of primal syncs at a node is to create transaction pool *replicas* for each neighbor of the node. We can then safely

run primal syncs in parallel using the corresponding replicas to avoid write collisions. Upon the completion of all parallel tasks, we can reuse our primal sync to sync the replicas locally. In that regard, we distinguish the *network* synchronization, when two adjacent nodes sync their transaction pools, and *local sync*, when multiple replicas sync locally at the same node. Importantly, only the network sync commits to the overall communication cost.

We summarize *SREP* in Algorithm 2 using  $S_n$  to denote the transaction pool at node  $n$ ,  $d_{in}$  to denote the differences between  $S_i$  and  $S_n$  that reside in  $S_i$ , and **Sync** to denote a primal sync. As an illustration, in Fig. 4-1, we depict one iteration of *SREP*'s main loop (line 2), assuming that each node  $n$  initially holds only one transaction whose hash is also  $n$ .

---

**Algorithm 2:** *SREP* Algorithm.

---

**Input:** Network  $G = (V, E)$  as adjacency list.

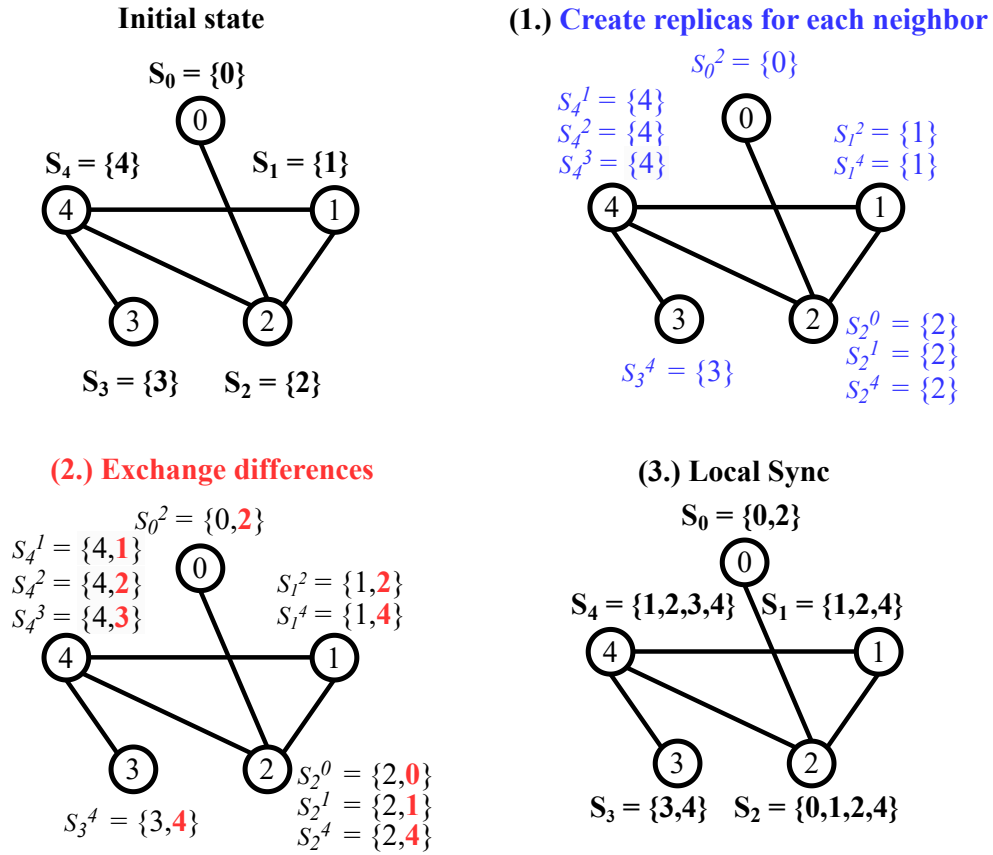
```

1 At each node  $n \in \{0, |V| - 1\}$ 
2   Loop
3     for  $i$  in  $G[n]$  do // Neighbors of  $n$ 
4        $S_n^i \leftarrow S_n$  ; // Replicate data set
5       Do in parallel
6         // Network sync
7          $d_{in} \leftarrow \mathbf{Sync} ( S_n^i, S_i )$  ;
8          $S_n^i \leftarrow S_n^i \cup d_{in}$  ;
9       for  $i$  in  $G[n]$  do
10        // Local sync
11         $S_n^i \setminus S_n \leftarrow \mathbf{Sync} ( S_n, S_n^i )$  ;
12         $S_n \leftarrow S_n \cup (S_n^i \setminus S_n)$  ;
```

---

### Avoiding Replication

*SREP* from Algorithm 2 has a significant memory overhead caused by transaction pool replication for each neighbor. However, certain primal syncs allow us to implement *SREP* without replication, thus mitigating this memory overhead. In particular, multiple set reconciliation algorithms mentioned in Chapter 3 use data set *sketches*



**Figure 4-1:** One iteration of *SREP* on a tractably small network  $G = (V, E)$ . Each node initially contains only one element equal to the node's unique identifier. The replicas at node  $n$  correspond to its neighbors and we denote them as  $S_n^i$ , for  $i \in G[n]$ . Transaction pools at each node are denoted as  $S_n$  for  $n \in V$ .

to perform synchronization and modify the underlying data sets only at the end of the protocol.

For instance, CPI reads from the set only once, at the beginning of the protocol, and writes to it only once at the end of the protocol. Suppose that we choose CPI as the primal sync in *SREP*. Then we can construct the characteristic polynomial of  $S_n$  as the very first step in each iteration (after line 2 in Algorithm 2). Instead of using the neighbor replicas, we can now use the same characteristic polynomial in all neighbor threads. As no thread will modify the polynomial, the procedure is thread-safe and the threads can now write directly to the underlying set. Although the write operation will need to acquire the lock corresponding to the transaction pool, the order in which the threads acquire the lock does not matter, because the set union operation is commutative and associative. As we now avoid replication, the local synchronization step can be safely eliminated altogether. This version of *SREP* is depicted in Algorithm 3, where we use  $\mathcal{X}_S$  to denote the characteristic polynomial of set  $S$ .

---

**Algorithm 3:** *SREP* with CPI as the primal sync. No local transaction pool replication.

---

```

Input: Network  $G = (V, E)$  as adjacency list.
1 At each node  $n \in \{0, |V| - 1\}$ 
2   Loop
3      $\mathcal{X}_{S_n} \leftarrow \text{CharacteristicPolyOf} ( S_n ) ;$ 
4      $T \leftarrow [] ;$  // List of threads
5     for  $i \leftarrow 0$  to  $G[n].size$  do
6       spawn thread  $T_i ;$ 
7        $T.append ( T_i ) ;$ 
8       Do in thread  $T_i$ 
9         // Network synchronization
10         $d_{in} \leftarrow \text{CPISync} ( \mathcal{X}_{S_n}, S_i ) ;$ 
11         $S_n \leftarrow S_n \cup d_{in} ;$ 
12    for  $i \leftarrow 0$  to  $G[n].size$  do
13       $T[i].join( ) ;$ 

```

---

---

$G = (V, E)$	Network of $ E $ edges and $ V $ nodes
$S_n$	Transaction pool at node $n \in \{0.. V  - 1\}$
$d_{ij} = S_i \setminus S_j$	Differences between $i$ and $j$ that reside in $i$
$\overline{deg}$	Average node degree
$t_n$	Time node $n$ spends to synchronize with all its neighbors once
$T_{x\%}$	Time until $x\%$ of $G$ is synchronized ( <b><math>x\%</math> of nodes keep equal sets</b> )
$\Sigma_{x\%}$	Number of primal sync invocations to achieve $x\%$ sync
$C_{x\%}$	Overall communication cost to achieve $x\%$ sync

---

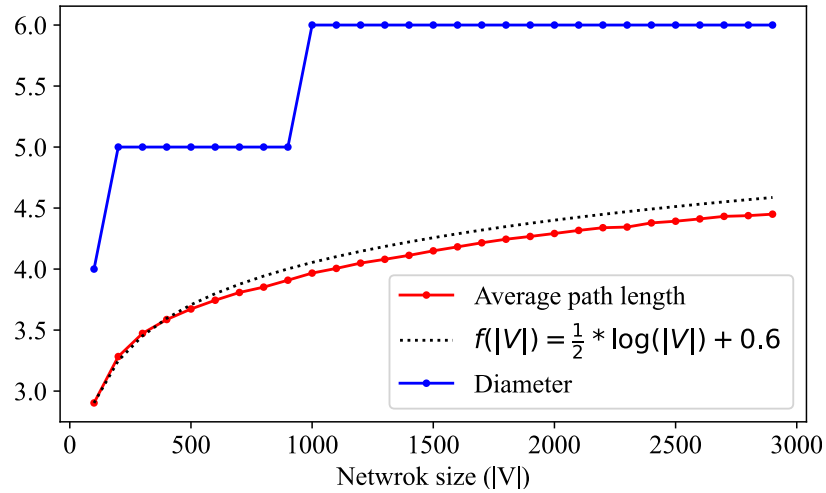
**Table 4.1:** Summary of notation.

Note that this implementation improvement does not change the functional properties of *SREP*. That is, each thread still operates on its own version of the sketch and will update its sketch only at the beginning of the subsequent iteration. Hence, a difference that arrives in iteration  $i$  via some neighbor thread will only get acknowledged by other threads in iteration  $i + 1$ . For that reason, we use the simple notion of “replicas” in the subsequent analysis.

## 4.2 *SREP* Performance Analysis

Several aspects affect the performance of *SREP*, including the network topology and the statistics of transaction pools. To aid our analysis, we first define an explicit network model, and then analyze *SREP* in a step-by-step fashion. In each stage of our analysis, we describe a *SREP* variant with the corresponding set of *simplifying assumptions* and analyze its performance. By successively relaxing these assumptions, we arrive at the final version of *SREP*. Table 4.1 summarizes notation used throughout this work.

**Definition 1:** We use  $T_{x\%}$ ,  $\Sigma_{x\%}$ , and  $C_{x\%}$  to denote time, total number of primal sync invocations, and total communication cost until  $x\%$  of transaction pools in the network are equal. When  $x = 100$ , we say that *full network* synchronization is



**Figure 4-2:** The small world property in random graphs generated through Watts-Strogatz model with  $k = \overline{deg} = 19$  and rewiring probability  $p = 0.24$ .

achieved — the ultimate goal of *SREP*.

#### 4.2.1 Network Model

Watts-Strogatz [84] random graphs allow us to describe a wide range of realistic blockchain network topologies reasonably well [82], [83], [147], [148]. A typical set of parameters to Watts-Strogatz model are the number of nodes in the network  $|V|$ , average node degree  $\overline{deg}$ , and rewiring probability  $p$  [84].

For instance, each Bitcoin node selects 8 random neighbors upon joining the network [149]–[151], which has been shown to yield an unstructured random graph [152]. We can capture this in the Watts-Strogatz model by setting  $\overline{deg} = 8$  and  $p = 1$  (see Section 2.4.3 for how Watts-Strogatz model can generate the random graph). Ethereum’s neighbor selection mechanism, on the other hand, relies on a Kademlia distributed hash table (DHT) [153], and yields a network with more structure [82]. Notwithstanding this, multiple recent measurement results have independently confirmed that the generated network exhibits the “small world” property and fits the Watts-Strogatz model [82], [83], [147]. That is, the average shortest path between

any two nodes can be reasonably approximated by  $O(\log|V|)$  [154], and the diameter of the network is small (see Fig. 4.2).

Besides the graph topology, our network model also captures the states of transaction pools across the network. In particular, we define the *pool assignment*  $A$  as a collection of sets  $S_0..S_{|V|-1}$  where set  $S_i$  represents the transaction pool at node  $i$ . We model the statistical properties of  $A$  through the following *pool parameters*:

$\mathcal{S}$ : *sizes distribution*. A discrete random variable describing the sizes of transaction pools  $S_i$  for  $i \in \{0..|V| - 1\}$ ,

$s$ : *sizes vector*. A  $|V|$ -size vector where elements are drawn from  $\mathcal{S}$ ,

$\mathcal{P}$ : *differences distribution*. A discrete random variable describing the sizes of mutual differences between the pairs of transaction pools (*i.e.*,  $|S_i \oplus S_j|$ ),

$M$ : *mutual differences matrix*. A  $|V| \times |V|$  upper triangular matrix of mutual differences. For the given topology  $G = (V, E)$ , the elements of the matrix are defined as:

$$m_{ij} = \begin{cases} |S_i \oplus S_j| & \text{when } (i, j) \in E \text{ and } i < j, \\ 0 & \text{otherwise.} \end{cases}$$

Non-zero elements are drawn from  $\mathcal{P}$ .

$D$ : *differences partition matrix*. A  $|V| \times |V|$  square matrix with zero diagonal where  $d_{ij} = |S_i \setminus S_j|$ . As we show in Appendix A.3, for the given mutual differences matrix  $M$ , there are  $\prod_{i < j} (m_{ij} + 1)$  difference partition matrices.

$\mathcal{U}$ : *universe*. A discrete random variable from which we draw transaction IDs. We choose  $\mathcal{U}\{0, u\}$  be a uniform random variable for some  $u \geq |V|$ .

### 4.2.2 Elementary *SREP* (*E-SREP*)

The starting point for our build up of *SREP* is called *elementary SREP* (Algorithm 4).

We summarize its simplifying assumptions as follows:

- (A<sub>1</sub>) All nodes have global view of the network.
- (A<sub>2</sub>) Initially, the transaction pools at each node contain only one element (transaction) that is unique across all network nodes (*e.g.*, index of the node). Strictly speaking, we set the pool parameters as:  $\mathcal{S} = 1$ ,  $\mathcal{P} = 2$ , and  $u \gg |V|$ .
- (A<sub>3</sub>) No new transactions arrive to the network after the initialization.
- (A<sub>4</sub>) In one iteration of elementary *SREP* (line 1), nodes take turns to perform their synchronization duties such that no two nodes invoke primal sync at the same time. For instance, nodes with smaller indices go first. An iteration ends when all nodes have invoked synchronization once for all their neighbors.
- (A<sub>5</sub>) Nodes synchronize with their neighbors sequentially. For instance, the neighbors with smaller indices get synchronized first (line 3).
- (A<sub>6</sub>) All synchronizations are two-way (lines 7 and 8), meaning that the differences are exchanged in both directions.
- (A<sub>7</sub>) All synchronizations take equally long.

In the context of *E-SREP*, the following special case is particularly significant for the analysis.

**Lemma 1:** For *E-SREP* over a complete graph  $G = (V, E)$ , the communication cost to sync the entire network is

$$C_{100\%}(G) = |V| \cdot (|V| - 1).$$

---

**Algorithm 4:** Elementary *SREP*.

---

**Input:** Network  $G = (V, E)$  as adjacency list.

```

1 while network is not fully synchronized do
2   for  $n \leftarrow 0$  to  $\{0..|V| - 1\}$  do
3      $neighbors \leftarrow \text{sort} ( G[n] )$  ;
4     for  $i$  in  $neighbors$  do
5        $d_{in} \leftarrow \text{Sync} ( S_n, S_i )$  ;
6        $d_{ni} \leftarrow \text{Sync} ( S_i, S_n )$  ;
7        $S_n \leftarrow S_n \cup d_{in}$  ;
8        $S_i \leftarrow S_i \cup d_{ni}$  ;
```

---

### 4.2.3 Elementary Parallel *SREP* (*EP-SREP*)

The main aim of the *elementary parallel SREP* is to relax  $(A_1)$ ,  $(A_4)$  and  $(A_5)$ . Instead of invoking synchronization in order, *EP-SREP* invokes synchronization for all neighbors at once (*i.e.*, Algorithm 2). In addition to that, we also relax  $(A_7)$ . The synchronization between nodes  $u$  and  $v$  now takes time *equal* to the number of their mutual differences (*i.e.*,  $|d_{uv} \cup d_{vu}|$ ). As discussed earlier in Section 3.1.3, this is a reasonable assumption to make (*e.g.*, CPI has such a property).

**Theorem 2:** In *EP-SREP* and for any connected network  $G = (V, E)$ , we have the following bounds on the overall communication cost until the network is fully synchronized:

$$|V| \cdot (|V| - 1) \leq C_{100\%} < |V| \cdot (|V|^2 - 1).$$

*Proof:* The lower bound is obtained similarly as in Lemma 1. The least amount of communication to achieve full synchronization is equivalent to each node sending its element to all the other nodes directly. On the other hand, we get the upper bound by observing that there cannot be more than  $|V|^2 \cdot (|V| - 1)$  *redundant* element transmissions on top of the lower bound. Redundant transmissions happen when a node receives an element via multiple replicas in the same iteration (see Appendix A.1). To count all redundant transmissions, we observe that, in each iteration, each node either receives some new elements or does not receive any. In the latter case, obviously, no

redundant transmissions happen. Otherwise, if there are some new elements received, the following holds: (1) there will be no more than  $|V|$  new elements arriving at the node across all iterations, as there is only that much elements in the network, and (2) for each element, there cannot be more than  $|V| - 1$  redundant transmissions, as there cannot be more than that much replicas at any node. Thus, there cannot be more than  $|V|^2 \cdot (|V| - 1)$  redundant transmissions at all nodes in all iterations. ■

As in Watts-Strogatz networks we have  $\overline{deg}$  replicas at each node on average, the same counting argument from above applies in the following form.

**Corollary 1:** For *EP-SREP* in Watts-Strogatz networks:

$$C_{100\%} < |V| \cdot (|V| \cdot \overline{deg} + |V| - 1).$$

On the other hand, to infer the upper bound on the time that *EP-SREP* needs to complete a full sync ( $T_{100\%}$ ), we rely on following definition.

**Definition 2:**  $I_{x\%}(G)$  is the maximal number of *EP-SREP* iterations (line 2 in Algorithm 2) at any node to achieve  $x\%$  network synchronization.

**Theorem 3:** In *EP-SREP* and for any connected network  $G = (V, E)$ , with the shortest path between nodes  $u$  and  $v$  denoted as  $dist(u, v)$ , the maximum number of iterations required for a full network synchronization is equal to the diameter of the network:

$$I_{100\%}(G) = \max_{u, v \in V} dist(u, v).$$

*Proof:* By the definition of full synchronization, all elements need to reach every other node. Without a loss of generality, suppose that we follow the propagation of some element  $i \in V$  during the execution of *EP-SREP*. Since the graph is connected, in each iteration of *EP-SREP*,  $i$  will progress exactly one step further through the network. The number of iterations required to synchronize the entire network is then equivalent to the maximum distance between any two nodes in the network (*i.e.*,

diameter).

■

**Lemma 2:** In *EP-SREP* over complete graphs  $G = (E, V)$ :

$$I_{100\%}(G) = 1 \text{ and } C_{100\%} = |V| \cdot (|V| - 1).$$

The former holds as the diameter of complete graphs is 1. The latter is a consequence of the former; as no element traverses more than one edge, there cannot be any redundant transmissions.

**Corollary 2:** For *EP-SREP* and Watts-Strogatz networks, the maximal number of iterations at any node to synchronize the entire network ( $I_{100\%}$ ) is logarithmic in the size of the network.

Counting the number of nodes that have heard about an element  $n \in V$  in iteration  $i$  of *EP-SREP* over a Watts-Strogatz network, we get the following sum:

$$1 + \overline{deg} + \overline{deg}^2 + \dots + \overline{deg}^i.$$

By equating it to  $|V|$ , we can express  $i$ , the number of iterations until all nodes have heard of  $n$ , as a logarithmic function of  $|V|$  [154]. Practically speaking, *EP-SREP* will complete in logarithmically small number of iterations ( $\approx 4 \log_{\overline{deg}}(10)$ ) for the blockchain networks of realistic sizes (*e.g.*, Blockchain and Ethereum [150], [151]).

**Theorem 4:** In general graphs  $G = (V, E)$ , the following holds for *EP-SREP*:

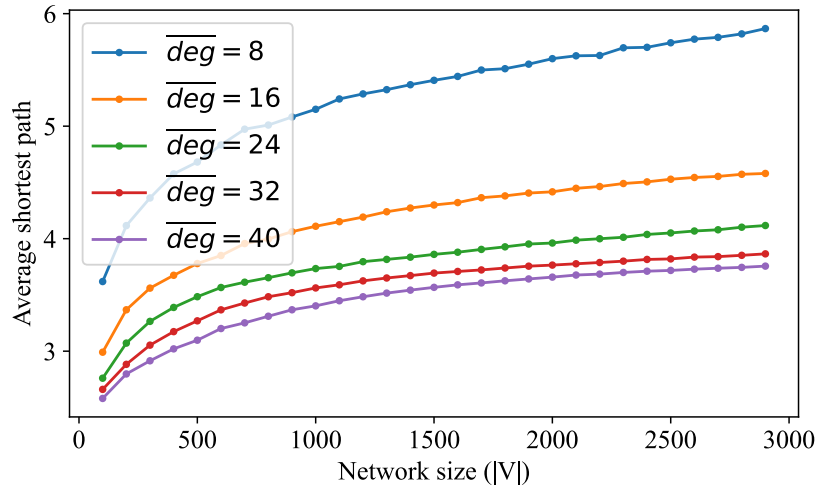
$$T_{100\%} \leq I_{100\%}(G) \cdot \max_{i \in V} t_i < I_{100\%}(G) \cdot |V|,$$

$$\Sigma_{100\%} \leq I_{100\%}(G) \cdot |E|.$$

*Proof:* Since synchronizations happen in parallel, the overall elapsed time is proportional to the number of iterations. Any sync invocation at any node will take strictly less than  $|V|$ , as no two data sets can differ in more than  $|V| - 1$  elements

(each data set keeps exactly one element at the beginning). Since in each iteration nodes sync with all their neighbors and each sync is two-way by  $(A_6)$ , there will be no more than  $|E|$  syncs in each iteration. ■

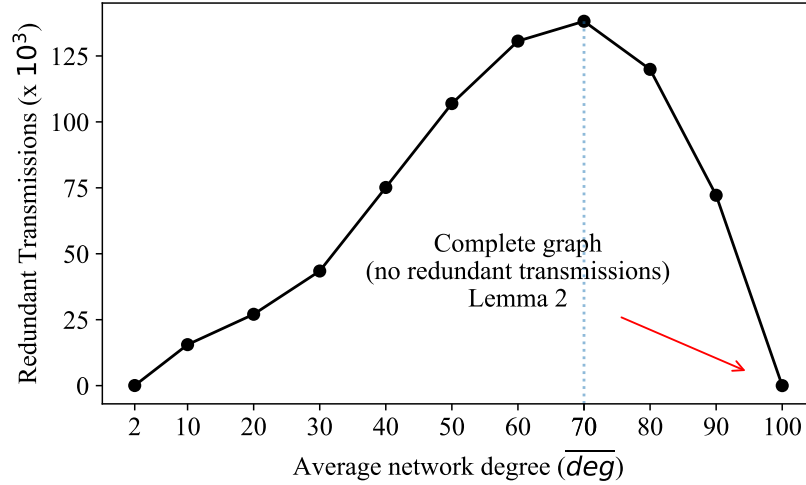
### The $\overline{deg}$ Dilemma



**Figure 4-3:** Average pair-wise path lengths in Watts-Strogatz model for various  $k$  and rewiring probability  $p = 0.24$ .

Due to the counting argument from Theorem 2, the upper bound on overall communication cost is *not* tight; there must be at least some elements that will *not* generate redundant transmissions in any connected network (see Appendix A.2). On top of that, the topology of the network plays a complex role in generating redundant transmissions. Intuitively speaking, the impact of  $\overline{deg}$  in Watts-Strogatz networks is twofold, and conflicting:

- (1) The larger  $\overline{deg}$ , the larger the average number of replicas per node, which may cause redundant transmissions, and
- (2) The larger  $\overline{deg}$ , the shorter the average pair-wise shortest path among the nodes in the network (see Fig. 4-3), which makes each element traverse less inter-



**Figure 4-4:** Amount of redundant t network of 100 nodes ( $p = 0.24$ ).

mediate nodes to reach the entire network, thus reducing the probability of redundant transmissions.

We plot this non-monotonic effect that  $\overline{deg}$  has on the amount of redundant transmissions in Fig. 4-4 for a tractably small network. Up to a point, the first effect (replicas count) prevails and drives the overall communication cost up. After that point, the second effect (path shortening) prevails and drives the overall communication cost down all the way to the point when the network becomes a complete graph and there is no redundant transmissions at all.

#### 4.2.4 Multi-element *SREP*

The final stage in building *SREP* is *multi-element SREP*. We build it by relaxing ( $A_2$ ) — transaction pools can now initially contain multiple elements. In terms of our network model, this means that our  $\mathcal{S}$  (sizes distribution) and  $\mathcal{P}$  (differences distribution) are no more constant. Thus, *SREP* is a *generalization* of *EP-SREP*.

**Definition 3:** Function  $f : (G, A) \mapsto \mathbb{Z}$  maps a pair of a topology  $G$  and a pool assignment  $A$  to a non-negative integer via first constructing the corresponding

mutual differences matrix  $M$ , then computing its sum  $\sum m_{ij}$ .

**Definition 4:** Function  $g : (G, A) \mapsto (G, A_{(next)})$  maps a pair of a topology  $G$  and a pool assignment  $A$  to the same topology  $G$  and a transformed pool assignment  $A_{(next)}$ . We define the transaction pools in the transformed pools assignment  $A_{(next)}$  as:

$$S_{(next)i} = S_i \cup \left( \bigcup_{j \in G[i]} S_j \right).$$

We use  $\bigcup_{j \in G[i]} S_j$  to denote the union of all sets  $S_j$  corresponding to the neighbors of node  $i$  in the previous iteration.

**Definition 5:** For some function  $h$ , we write  $h^{(n)}(x)$  to denote the composition of function  $h$  with itself  $n$  times, starting with argument  $x$ :

$$h^{(n)}(x) = \underbrace{h \circ h \cdots h}_{n}(x).$$

**Definition 6:**  $A_{(n)}$  is the assignment resulting from  $n$  compositions of  $g$  with itself starting with the initial pool assignment that we denote as  $A = A_{(0)}$ .

**Lemma 3:** For a network model  $(G, A)$  where  $G$  is a connected graph and  $A$  the initial pool assignment, the number of *SREP* iterations to achieve the full network synchronization  $I_{100\%}(G, A)$  is given as a solution to the following equation:

$$f(g^{(I_{100\%}(G, A))}(G, A)) = 0.$$

Note that by Definition 4,  $g$  exactly corresponds to one iteration of *SREP*. That is, the transformed pool assignment  $A_{(next)}$  reflects the state of the transaction pools after an iteration of *SREP* at all nodes in the network. Composing  $g$  with itself  $n$  times corresponds to repeating an iteration of *SREP* at all nodes  $n$  times. By a similar argument as in Theorem 3, all elements will reach all nodes after some number of iterations. Since this implies that no two sets have any differences,  $M$  will be an

all-zeros matrix. That is,  $(f \circ g^{(n)})(G, A)$  has at least one zero. Thus, the number of times we need to compose  $g$  with itself until  $f(G, A_{(n)}) = 0$  gives us the maximal number of *SREP* iterations to achieve full network synchronization

**Theorem 5:** For a connected graph  $G = (V, E)$  and an initial pool assignment  $A$ , the number of *SREP* iterations to achieve the full network synchronization is bounded by the diameter of the network:

$$I_{100\%}(G, A) \leq \max_{u,v \in V} \text{dist}(u, v).$$

*Proof:* As *SREP* is a generalization of *EP-SREP*, the argument here is similar to that of Theorem 3. To achieve the full network synchronization, elements need to traverse at most the diameter of  $G$ . As opposed to *EP-SREP*, in *SREP* each element may initially appear at more than one node, dictated by the differences distribution  $\mathcal{P}$ . Thus the diameter is an upper bound on *SREP* iterations. ■

**Lemma 4:** For a connected graph  $G = (V, E)$  and initial pool assignment  $A$  with the corresponding mutual differences matrix  $M$ , the communication cost of *SREP* is:

$$\begin{aligned} C_{100}(G, A) &= \sum_{i=0}^{I_{100\%}(G,A)} f(G, A_{(i)}) \\ &< I_{100\%}(G, A) \cdot \max\{f(G, A), \dots, f(G, A_{(I_{100\%}(G,A))})\}. \end{aligned}$$

In  $i$ th iteration of *SREP*, we transmit exactly as much elements as there are in the differences matrix that corresponds to  $A_{(i)}$ . Given  $I_{100\%}(G, A)$  from Lemma 3, we get the overall communication cost of *SREP*.

**Lemma 5:** In *SREP* over a connected network  $G = (V, E)$  with the given initial pool assignment  $A$  and the largest order statistics of differences distribution  $\mathcal{P}$  denoted

as  $\mathcal{P}_{(n)}$ :

$$T_{100\%} \leq I_{100\%}(G, A) \cdot \max_{i \in V} t_i = I_{100\%}(G, A) \cdot \mathcal{P}_{(n)},$$

$$\Sigma_{100\%} \leq I_{100\%}(G, A) \cdot |E|.$$

The argument is similar to that of Theorem 4.

Finally, note that the assumptions in our analysis such as  $(A_3)$  — no new transactions arrive after *SREP* starts, are artificial in that they simplify our analysis, but they do not constrain *SREP* in practice. The properties such as the overall communication cost ( $C_{100\%}$ ) and time ( $T_{100\%}$ ) to sync the entire network relate to the transactions that have arrived before *SREP* begins.

### 4.3 Simulations

To validate our analytical findings about *SREP*, we construct an event-based simulator called *SREPSim* [155] that shares the topology generation procedure with *CBlockSim* of Ma *et al.* [148] and adds the other parameters of our network model described in Section 4.2.1.

In the rest of this section, we first describe a method to parameterize our network model using real-world transaction pool data. Then, we use such parameterized model to validate the main analytical properties of *SREP*. We then compare the overall communication cost of *SREP* with a similar approach from the literature. At the end, we present a *SREPSim* optimization that allows for easy *SREP* communication cost calculation over large-scale networks.

#### 4.3.1 Configuring Network Model Parameters

Unlike the simulation approaches from the literature (*e.g.*, *BlockSim:Faria* [156], *BlockSim:Alharby* [157], and *SimBlock* [158]), our network model can seamlessly in-

tegrate real-world transaction pool data. For instance, the empirical distributions of  $\mathcal{S}$  and  $\mathcal{P}$  can be generated for some small subset of all nodes in the network using the measurement software such as *log-to-file* of Imtiaz *et al.* [159], [160]. This software instruments adjacent Bitcoin nodes and periodically serializes the snapshots of their transaction pools. From these transaction pool snapshots, we can measure transaction pool sizes and their mutual differences to construct the empirical distributions for  $\mathcal{S}$  and  $\mathcal{P}$ .

For the purpose of this work, we have conducted a 3-day long measurement campaign on two time-synchronized Bitcoin nodes in the main network, and requested the transaction pool snapshots each minute. Fig. 4-5 depicts the results that we obtained. Roughly speaking, the set sizes fit the Maxwell distribution reasonably well, while the number of mutual differences fits the Hyperbolic distribution. Next, given the empirical distribution of  $\mathcal{S}$ , we need to configure the rest of our network model’s pool parameters. Ultimately, we need to construct a pools assignment  $A$  that conforms to the differences distribution  $\mathcal{P}$ <sup>1</sup>.

In *SREPSim*, we construct the target transaction pools assignment  $A$  through Procedure 5. In particular, Procedure 5 solves the following problem. Given the network topology  $G = (V, E)$ , sizes distribution  $\mathcal{S}$ , and a parameter  $\psi$ , construct the pools assignment  $A$ . The main observation here is that we can use parameter  $\psi$  to gauge the differences distribution  $\mathcal{P}$  of the resulting pools assignment  $A$ . As shown in Fig. 4-6,  $\psi = 0.35$  makes the resulting differences distribution fit our target experimental distribution from Fig. 4-5 reasonably well. Note that we can simply use the parameter  $\psi$  to increase or decrease the similarity between transaction pools in our simulations. For instance, we can decrease the average similarity among the transaction pools (*i.e.*, increase the number of their mutual differences) by increasing  $\psi^2$ , as shown in Fig. 4-6

---

<sup>1</sup>Direct usage of  $\mathcal{P}$  is also possible but perhaps harder (see Appendix A.3).

<sup>2</sup>See Appendix A.5 for a discussion on Procedure 5.

in blue and green.

---

**Procedure 5:** Network parameterization in SREPSim.

---

**Input:** Network  $G = (V, E)$ .  
**Input:** Sizes distribution  $\mathcal{S}$ .  
**Input:** Parameter  $\psi$ .  
**Output:** Pool assignment  $A$ .

```

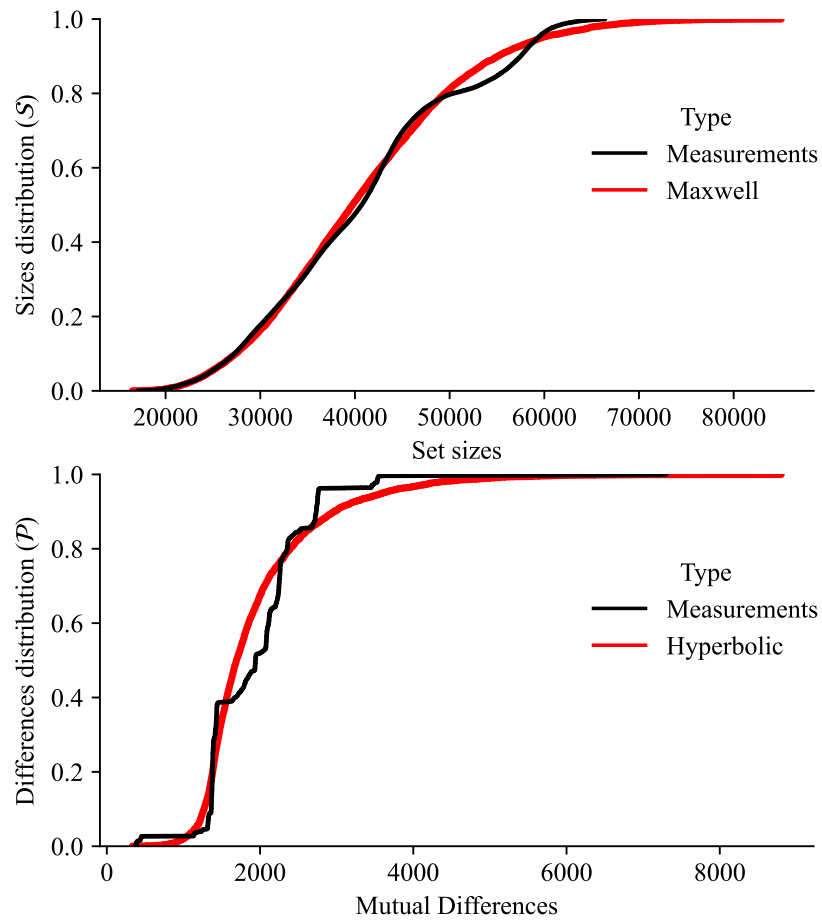
1  $u \leftarrow \lceil \psi \mathbb{E}[\mathcal{S}] \rceil$  ;
2  $\mathcal{U}\{0, u - 1\}$  ; // Uniform distribution
3  $\text{sizes} \leftarrow \text{sample } |V| \text{ elements from } \mathcal{S}$  ;
4  $A \leftarrow []$  ;
5 for  $i \leftarrow 0$  to  $|V| - 1$  do
6    $S_i \leftarrow \text{sample } \text{sizes}[i] \text{ elements from } \mathcal{U}$  ;
7    $A.\text{append}( S_i )$  ;
```

---

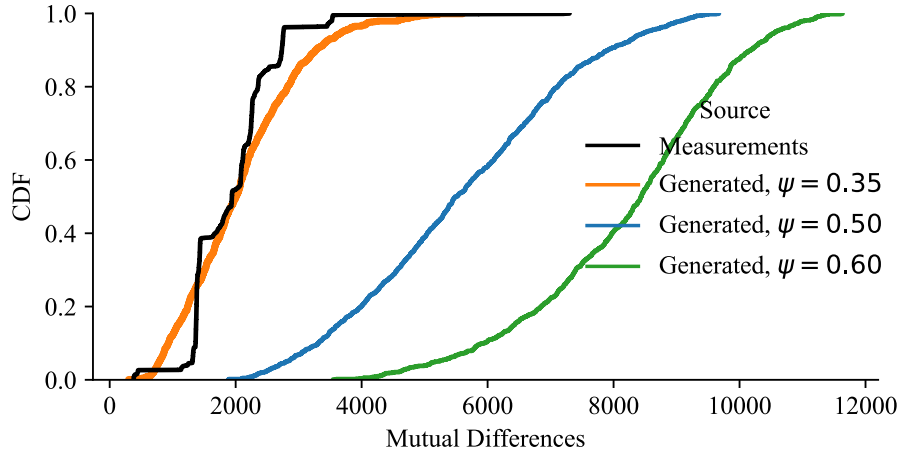
### 4.3.2 *SREP* Properties Validation

The main analytical properties that we want to validate through simulations are *SREP*'s communication cost to achieve full network sync ( $C_{100\%}$ ) and the time required to achieve this state ( $T_{100\%}$ ). In particular, we want to show how these two quantities change as a function of the network topology and the measure of difference among the transaction pools. As discussed earlier in Section 4.2.1, one of the main parameters that affects the topology of the network is the average node degree ( $\overline{deg}$ ).

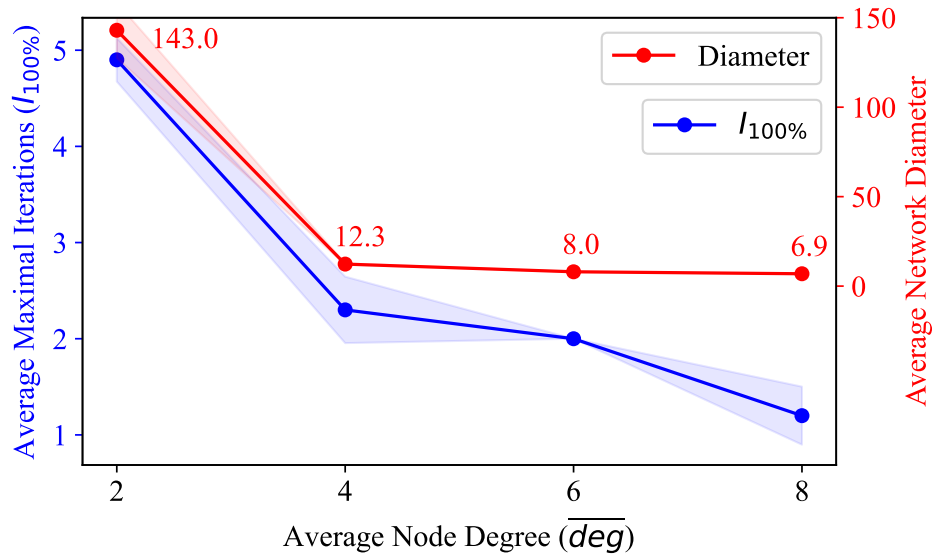
In Fig. 4-7, we plot the relation between  $I_{100\%}$  and the network diameter. In Fig. 4-8, we plot the communication cost and time to full network sync as a function of  $\overline{deg}$ . The main observation is that the overall communication increases with the average node degree as a consequence of using more replicas per node, which increases the number of redundant transmissions (see Fig. 4-4). On the other hand, the time to achieve full network synchronization does not exhibit such a trend. Since primal syncs run in parallel, it is the maximal number of differences among any two nodes in the network that dominates the total time to sync the network (see Lemma 5).



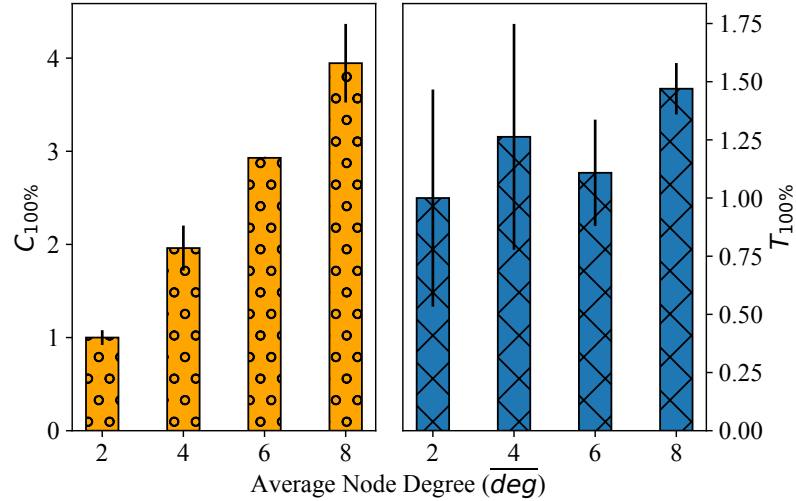
**Figure 4-5:** Empirical distributions of transaction pool sizes  $\mathcal{S}$  for two adjacent Bitcoin nodes (up) and their mutual differences  $\mathcal{P}$  (down). Best distribution fits in red (using Error Sum of Squares).



**Figure 4-6:** Empirical differences distribution for two adjacent Bitcoin nodes versus the differences distribution generated by Procedure 5 for various  $\psi$ . Watts-Strogatz network with 100 nodes ( $\overline{deg} = 19$  and  $p = 0.24$ ).



**Figure 4-7:** Maximal number of *SREP* iterations at any node ( $I_{100\%}$ ) bounded by the network diameter for Watts-Strogatz graphs with 1000 nodes ( $p = 0.24$ ). 95% confidence intervals.

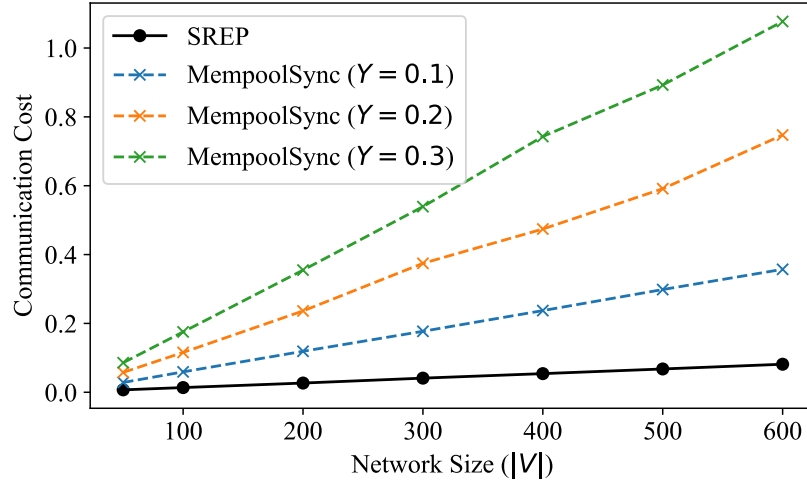


**Figure 4-8:** Relative communication cost ( $C_{100\%}$ ) and time to fully synchronize the network ( $T_{100\%}$ ). Network with 1000 nodes ( $p = 0.24$ ).

### 4.3.3 Comparison with *MempoolSync*

*MempoolSync* of Imtiaz *et al.* is a transaction pool synchronization protocol that can improve the average transaction propagation delay by 50% in the event of churn in the Bitcoin network [43]. Here we describe this protocol and compare its communication efficiency with our newly proposed *SREP* through simulations.

As pointed out in [43], the main reason for slow block propagation times is a large number of missing transactions in the transaction pools of the block-receiving nodes. This effect occurs in the legacy block propagation protocols such as *CompactBlock* [22] and the more recent improvements such as *Graphene* [23], [161]. Thus, the goal of *MempoolSync* is to supply the nodes with potentially missing transactions, and it does so through an *ancestor score*-based heuristics [162]. The protocol uses a small constant `DefTxtoSync` as the default number of transaction hashes that the transmitting node will select from its transaction pool in descending order of ancestor score. The transmitting node will send exactly `DefTxtoSync` selected transaction hashes *unless* one of the following holds:



**Figure 4-9:** Normalized overall communication cost of *SREP* ( $C_{100\%}$ ) and *MempoolSync* as a function of network size. Transaction pool sizes and differences from our Bitcoin measurement campaign (Section 4.3.1).  $DefTXtoSync = 1000$ .  $Y$  is the *MempoolSync* heuristic constant.

- 1) Transmitting node’s transaction pool is much larger than  $DefTXtoSync$  (e.g., 10 times). In this case, the node will send  $Y \times DefTXtoSync$  top rated transactions, where  $Y$  is a constant between 0 and 1, or
- 2) Transmitting node’s transaction pool is smaller than  $DefTXtoSync$ . In this case, the node will send its entire transaction pool. Because  $DefTXtoSync$  is a small constant, this is a quite rare event. It occurs only when the node has just joined the Bitcoin network or has just propagated a large block that triggered a massive transaction pool cleanup [43].

In Fig. 4-9, we compare the overall communication costs of *MempoolSync* and *SREP*. For *SREP*, we plot the communication cost to sync the entire network ( $C_{100\%}$ ). For *MempoolSync*, we plot the communication cost that *MempoolSync* incurs until *SREP* would achieve a full sync.

Note that this kind of comparison gives an advantage to *MempoolSync*. While *SREP*’s  $C_{100\%}$  implies that the network is fully synced, *MempoolSync*’s communication

$\overline{deg}$	$\psi$	Diameter <i>average</i>	$I_{100\%}$ <i>average</i>	$C_{100\%}$ (GB) <i>average</i>
4	0.355	16	2.5	1.214397
	0.5		3.0	3.165879
	0.6		3.1	4.801665
8	0.355	9	1.7	2.428649
	0.5		2.0	6.317304
	0.6		2.0	9.569259
12	0.355	7	1.0	3.642738
	0.5		1.5	9.485572
	0.6		2.0	14.347242
16	0.355	6	1.0	4.876714
	0.5		1.0	12.649385
	0.6		1.0	19.135943
20	0.355	5	1.0	6.065679
	0.5		1.0	15.804836
	0.6		1.0	23.886079
24	0.355	5	1.0	7.294909
	0.5		1.0	18.966694
	0.6		1.0	28.672272
28	0.355	5	1.0	8.465624
	0.5		1.0	22.156316
	0.6		1.0	33.446278

**Table 4.2:** *SREP* over a 10,000 nodes network.  $p = 0.24$ .

cost does not. In fact, *MempoolSync* has no guarantees about the communication (or time) needed to sync the entire network. Note also that *MempoolSync* uses Bitcoin internals to calculate the ancestor score of the transactions and later uses this score to determine which transactions to transmit. As opposed to *MempoolSync*, *SREP* is a general approach that does not rely on any Bitcoin internals and can be seamlessly integrated into other blockchains that keep transaction pools.

#### 4.3.4 Communication Cost in Large-Scale Networks

Event-based simulators such as *SREPSim* may consume prohibitive amounts of memory and take a long time to complete simulations when the simulated network is large [148]. To address this issue, we designed a *SREPSim* module that computes *SREP*'s performance metrics analytically. In particular, we implement the functions from Definitions 3 and 4, and rely on the results from Lemma 4 to compute  $C_{100\%}$  and  $I_{100\%}$ . We describe the *SREPSim*'s analytical module in Procedure 6. Using this module, we can easily compute the desired performance metrics for the networks of realistic sizes (*e.g.*, Bitcoin and Ethereum) [150], [151].

In Table 4.2, we summarize the results for a 10,000 nodes network with various average node degrees ( $\overline{deg}$ ) and the measure of similarity among transaction pools ( $\psi$ ). As we report the communication cost, we assume that the transaction pools represent each transaction as a 32-byte long globally unique hash [146]. All simulations complete in tens of minutes.

### 4.4 Summary

In summary, we developed *SREP*, an independent protocol that assists block propagation in large-scale blockchains. It does so by synchronizing transaction pools of adjacent nodes in the blockchain network using a communication-efficient end-to-end sync protocol (see Section 3). In contrast to the previous approaches from the literature that insert transaction pool synchronization directly into the block propagation protocols, *SREP* operates in a distributed manner *outside* the block propagation channels of the network. As a result, it is easier to formally analyze *SREP*'s performance, and, indeed, we have shown that it completes in time bounded by the network diameter. In “small-world” networks, a reasonable model of popular public blockchains, *SREP* converges in a number of steps logarithmic in network size.

---

**Procedure 6:** SREPSim's analytical module.

---

**Input:** Network  $G = (V, E)$ .  
**Input:** Initial pool assignment  $A$  as  $S_0..S_{|V|-1}$ .  
**Output:** Overall network communication cost  $C_{100\%}$ .  
**Output:** Maximal number of iterations  $I_{100\%}$ .

```

1 function CalculateM(A):
2    $M \leftarrow \text{zeros}(|V| \times |V|)$  ; // Zero matrix
3   for  $i \leftarrow 0$  to  $|V| - 1$  do
4     for  $j \leftarrow i + 1$  to  $|V| - 1$  do
5       if  $i \in G[j]$  then // i neighbor of j
6          $M[i][j] \leftarrow |S_i \oplus S_j|$  ;
7   return  $M$ ;
8  $C_{100\%} \leftarrow 0$  ;
9  $I_{100\%} \leftarrow 0$  ;
10  $M \leftarrow \text{CalculateM}(A)$  ;
11 while  $\sum m_{ij} > 0$  do
12   for  $i \leftarrow 0$  to  $|V| - 1$  do
13      $S'_i \leftarrow S_i$  ; // New assignment
14     for  $j \in G[i]$  do
15        $S'_i \leftarrow S'_i \cup S_j$  ;
16    $C_{100\%} = C_{100\%} + \sum m_{ij}$  ;
17    $I_{100\%} \leftarrow I_{100\%} + 1$  ;
18    $A \leftarrow A'$  ;
19    $M \leftarrow \text{CalculateM}(A)$  ;

```

---

We have also validated our analytical findings against a novel event-based simulator that we have developed. We run the simulator on real-world transaction pool statistics drawn from our own measurement campaign. In our simulations, *SREP* incurs several times less communication overhead than *MempoolSync*, a similar approach from the literature, on the transaction pools statistics from the live Bitcoin network. *SREP* can synchronize the networks of tens of thousands of nodes incurring only tens of gigabytes of overall bandwidth overhead. Our simulation technique introduces a novel approach of integrating the real-world transaction pool data from measurements into simulations. Therefore, we can simulate the blockchains of realistic scale and easily control the statistics of transaction pools in simulations.

## 4.5 Discussion

The problem of synchronizing data sets in the network of many participants is somewhat similar to the problem of *multi-party* set reconciliation (see Section 3.1.4). Although the main benefit of utilizing multi-party set reconciliation for transaction pools synchronization can be further reduction in overall communication cost, it is not clear whether an advantage over pairwise approach of *SREP* can be achieved when the total set intersection  $(\cap_i S_i)$  is relatively small compared to the average pair-wise intersection [89].

Another problem that is somewhat similar to network-wide data synchronization is *k-token dissemination* [163]–[166]. In the base k-token dissemination problem formulated by Topkis [163], we are given a synchronous network  $G = (V, E)$  and some  $k$  pieces of information (tokens) spread across the nodes. The goal is to disseminate all  $k$  tokens to all  $|V|$  participants, with the constraint that no more than one token can traverse each edge in each round. Topkis [163] has shown that the problem can be solved in  $O(|V|)$  time in general static graphs. This result is in accordance with

our bounds from Theorem 5.

A restricted class of  $k$ -token dissemination when each node initially keeps exactly one globally unique token is defined by Kuhn *et al.* [164] and named *all-to-all token dissemination*. The initial conditions of all-to-all token dissemination are equivalent to *EP-SREP*. Kuhn *et al.* proved that the  $\mathcal{O}(|V|)$  time complexity bound holds even in *1-interval connected* dynamic networks, where 1-interval connectivity means that the edges in the network may change in between the rounds, but in each round there exists some unknown connected spanning subgraph. We believe that the time complexity of *k-token dissemination* may yield an upper bound on the number of *SREP* iterations until the entire network is synchronized, and leave the proof for the future work.

## Chapter 5

# *GenSync* Framework

As we explicate in Section 3.2, the state-of-the-art *data sync* (set reconciliation) protocols differ significantly in their core algorithmic principles. This not only hinders the comparative analysis from the theoretical perspective, but also creates several challenges when it comes to implementing a practical data sync framework. Roughly speaking, these challenges stem from the following: (1) unifying and simplifying the parameterization of disparate protocols to offer reasonable defaults to the users, (2) designing generic protocol implementations to make them useful for a wide range of applications (various data sources and system platforms), and (3) designing a lightweight yet versatile benchmarking subsystem.

To tackle these challenges, we designed *GenSync*, a self-contained open-source data reconciliation framework with a well-integrated benchmarking subsystem allowing for methodical cost-benefit analysis among data sync protocols under realistic system conditions, and seamless protocol integration into existing implementations. As outlined in Table 5.1, *GenSync* currently implements both coding theory- and approximate set membership data structures-based protocols with various extensions to support additional capabilities such as multiset and prioritized synchronization (see Section 3.1.4).

For the sake of portability and bolstering *GenSync*'s usage in future applications, we implemented it as a C++ middleware library with a minimal user-facing API [42], [167]. As shown in Fig. 5-1, a *GenSync* user application interacts with the library

Protocol	Capabilities				
	Set	Multiset	Set of sets	Prioritized	Interactive
CPI	✓	✓	✓	✓	✓
IBLT	✓	✓	✓	✗	✗
Cuckoo	✓	✗	✗	✗	✗

**Table 5.1:** Set reconciliation protocols currently supported by *GenSync* and their capabilities.

through the API, which passes control to the *compatibility* layer for protocol parameterization and configuration. The compatibility layer then invokes the required protocol implementation that carries out the sync process and optionally reports fine-grained execution statistics back to the user application. At compile time, *GenSync* users can enable the optional benchmarking layer by choosing one of the two modes of operation: (1) *prototyping*, which enables fine grained performance monitor hooks that allow for integration with the *GenSync*'s benchmarking layer and the reporting subsystem, and (2) *in-situ*, which disables the performance monitor hooks to eliminate the middleware overhead. When compiled in the *in-situ* mode, *GenSync* operates as a standard shared library. Importantly, no application logic interventions are needed on the user's side to alternate between modes of operation, which allows for seamless experimentation and prototyping.

*GenSync* protocol implementations are platform-independent (*i.e.*, do not root their performance in platform-specific capabilities such as special CPU instructions or cache hierarchies), and operate with generic data set representations. The former is particularly significant when a user applies *GenSync* to heterogeneous distributed computing platforms [40], [168] (*e.g.*, distributed ledgers, where nodes need to support a wide range of hardware platforms), while the latter allows for integration with various data sources (*e.g.*, databases, blockchain *memory pools*).

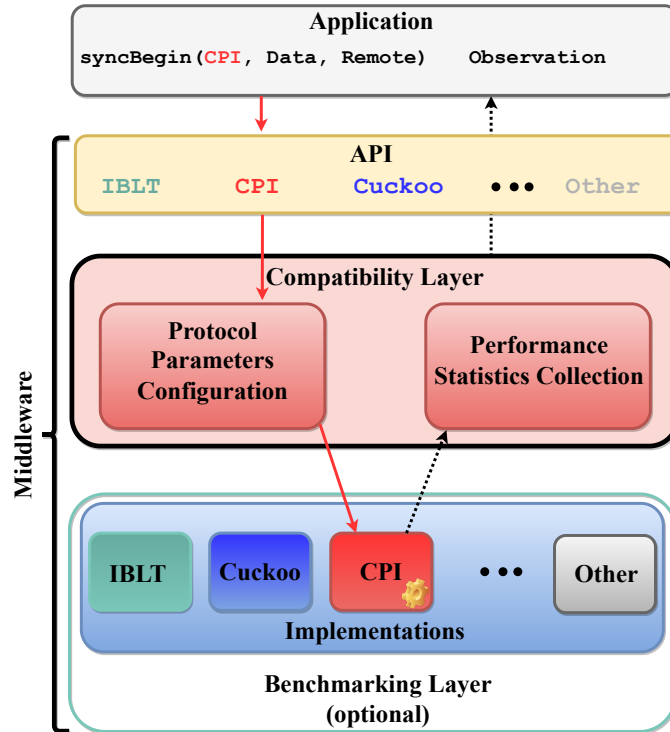


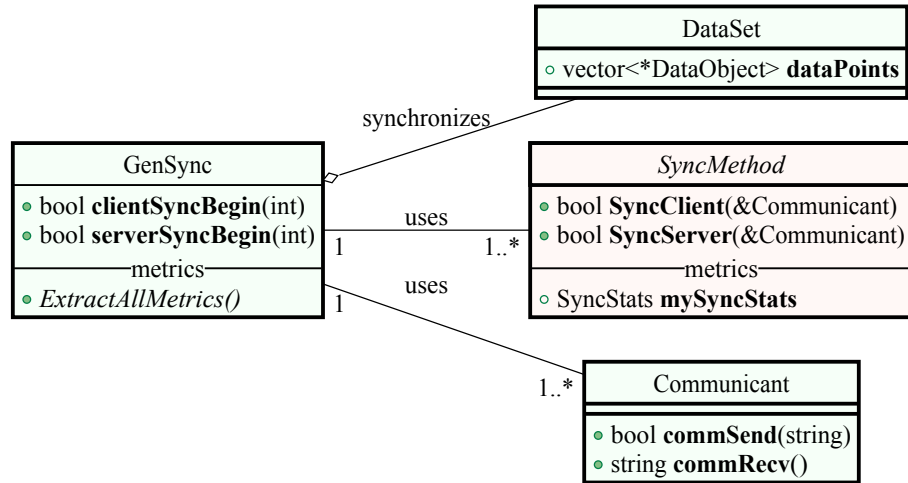
Figure 5.1: *GenSync* middleware structure.

## 5.1 Core Abstractions

*GenSync*'s strength lies in its versatile hierarchy of abstractions. Next, we describe the role of the four core *GenSync* abstractions whose relation is depicted in Fig. 5.2.<sup>1</sup>

**DataObject:** The light-weight data point abstraction is called `DataObject`, and its main purpose is to improve compatibility between data sources by imposing a common representation of data points. Regardless of the data source, *GenSync* requires records to be converted to `DataObjects`. One way to achieve this conversion is through hashing, where the resulting hashes serve as unique identifiers. A good distribution of these hashes can be achieved by uniform hashing over a sufficiently large space [169]. However, certain applications already keep globally unique identifiers for each data point in the system and fit naturally to *GenSync*. For instance, blockchains use

<sup>1</sup>See Appendix B for a comprehensive view.



**Figure 5.2:** Relation between four core *GenSync* abstractions.

hashing to assure global uniqueness of transaction identifiers, which can be reused in *GenSync* to eliminate the need for re-hashing when `DataObject` of a transaction is created. Internally, *GenSync* relies on Shoup’s *Number Theory Library* (NTL) [170] to efficiently represent variable length signed integers for long hashes.

**Communicant:** The communication channel between two *GenSync* users (nodes) is represented using the `Communicant` abstraction. Nodes can sync among themselves through various transport- or even physical-layer protocols using the corresponding `Communicant`. The current version of *GenSync* includes two `Communicant` implementations: (1) TCP socket (which we use in our experiments in Section 5.3), and (2) C++ string for data sets that are disconnected in space, time, or reside on the same node. *GenSync* users can extend the framework to support other network protocols such as UDP or SCTP — Stream Control Transmission Protocol [171] by implementing corresponding `Communicants`.

**SyncMethod:** The minimal substructure for all data sync protocols in *GenSync* is `SyncMethod`, which is designed to carry out the following operations:

- (1) agree on reconciliation parameters among the participants,

- (2) invoke the desired protocol implementation, and
- (3) update underlying data sets upon a successful completion of the sync process.

All sync protocols currently implemented by *GenSync* inherit their structure from `SyncMethod`, and the same should be done by researchers and practitioners that want to extend *GenSync* with (1) new protocols to benchmark against the state-of-the-art, or (2) new platform-specific implementations of existing protocols to increase the sync performance in homogeneous systems. The *GenSync* benchmarking layer relies on the `SyncMethod` abstraction to connect the fine-grained performance hooks that are used upstream in the framework to report the execution performance statistics, which allows for a standardized experimental comparison of sync protocols under realistic system conditions.

***GenSync*:** The topmost abstraction that incorporates all components of the *GenSync* framework is `GenSync`. From the users' perspective, `GenSync` object serves as an entry point to the middleware (see Listing 1). The sync process begins when the node that invoked `SyncServer` receives the connection from the node that invoked `SyncClient`. As discussed in Section 3.2, all non-trivial sync protocols may fail with arbitrarily small probability, which is indicated by the return value of the `SyncMethod`. Given that the failure probability can be configured to an arbitrarily small value, one straightforward way to assure a successful synchronization is to repeat the `SyncMethod` invocation until it succeeds.

***Observation*:** When *GenSync* is used in the prototyping mode and the fine-grained performance hooks are enabled, a sync invocation (through `ServerBegin` or `ClientBegin`) produces an `Observation` object that is returned to the application and can be serialized, which is particularly amenable for analysis. This object contains the summary of performance metrics captured by the middleware including the protocol signature (protocol name and exact parameters used), overall time spent in running

```
#include <GenSync.h>

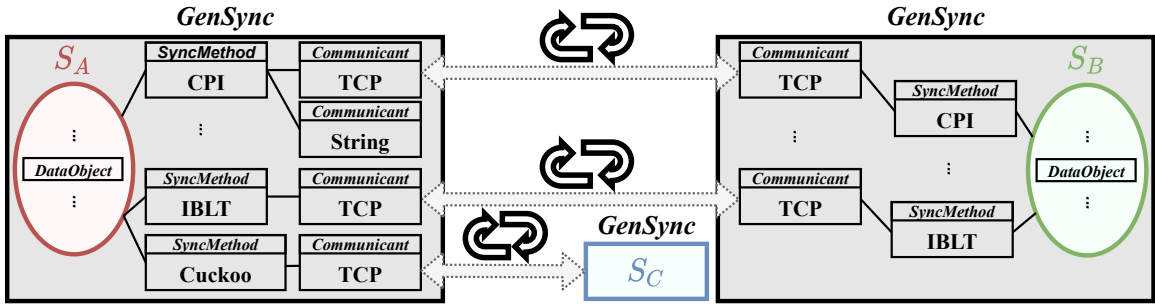
// Point to a remote and pick a protocol
auto builder = GenSync::Builder();
builder.setProtocol(GenSync::CPI);
builder.setCommunicant(GenSync::socket);
builder.setHost("the.peer.remote.addr");

GenSync gs = builder.build();

// Add data
for (auto data_point : data_set)
gs.addElement(hash(data_point));

// Perform sync
if ( gs.syncServer() ) {
    // Get execution statistics
    Observation ob = gs.getObservation();
    ob.communicationTime;
    ob.computationTime;
    ob.bytesTransmitted;
} else {
    // Sync failed
}
```

**Listing 1:** Illustrative usage of *GenSync* from an application.



**Figure 5-3:** *GenSync* usage example. Alice (left) and Bob (right) sync their corresponding sets  $S_A$  and  $S_B$ . Each time Alice wants to sync with Bob, she can choose between CPI-over-TCP and IBLT-over-TCP. When Alice wants to sync with Chuck, a third participant, she always uses Cuckoo-over-TCP.

the sync (from client and server’s perspective), the number of bytes transmitted from the server to the client (and vice versa), real time spent in computing the metadata (*e.g.*, approximate set membership data structure encoding and decoding), real time spent pushing the data through the corresponding *Communicant* (which can indicate congestion on sender and receiver’s side), and success codes that indicate the type of error that caused sync to fail (*e.g.*, “peeling” process failure in IBLTs).

The above-described hierarchy of abstractions makes *GenSync* practical for a variety of diverse use cases. For instance, a *GenSync* node may sequentially initiate data sync with several neighbors, where each of them may use different *Communicant*, *SyncMethod*, or even different protocol. One such usage example is depicted in Fig. 5-3, where a participant (Alice) syncs with two other neighbors (Bob and Chuck) using different *Communicants* and sync protocols depending on some internal decision (*e.g.*, the amount of available bandwidth on the links immediately before the sync or a strict bandwidth budget). We describe another *GenSync* use case later in Section 5.6 where we conduct a case study involving one of the world largest wireless network emulators [172].

## 5.2 *GenSync* Testbed

Experimental evaluation of data sync protocols under realistic system conditions and data parameters requires a versatile testbed that allows for: (1) network parameters adjustment, (2) isolation of the server and client execution environments to reduce interference that can cause sways in real time measurements, and (3) synthetic data generation to support protocol execution under various data parameters (*i.e.*, data size and the number of mutual differences).

To fulfill the above requirements, we build upon several technologies. First, we use *Mininet* [173] with *Open vSwitch* [174] to emulate network performance aspects such as bandwidth, latency, and packet-loss between the *GenSync* nodes. For instance, when there are two participants, we create a single-switch-two-nodes Mininet topology and configure each of the two links separately. This, combined with Linux Traffic Control subsystem [175], allows us to implement complex scenarios such as asymmetrical bandwidth (*i.e.*, when Alice can send more data per unit of time to Bob than Bob can to Alice). Second, we use core Linux kernel virtualization functions such as `cgroups` [176] and Completely Fair Scheduler [177] to gauge the compute capabilities of nodes in *GenSync* testbed. In conjunction with the public CPU benchmark information from *PassMark* [178], we can configure the compute power of the modeled CPU's. More precisely, *GenSync* testbed configures the `cgroup` parameters to assure that the CPU usage of the emulated nodes does not exceed the target fraction of the testbed machine's single core capacity.

The technologies we used to design our *GenSync* testbed impose certain technical limitations to our framework. First, the single core performance of the CPU that executes the testbed (*host machine*) is an upper bound to the performance of the CPU that we can emulate. This is a direct consequence of the `cgroup` and Completely Fair Scheduler design in the Linux kernel, which is primarily designed to solve the problem

of the host machines resource sharing among several users. For instance, it does not emulate the internals of the target CPU such as specialized instructions or cache hierarchies. Second, the network parameters that we can model in *GenSync* testbed are constrained by the capabilities of Mininet emulator. For instance, the packet-loss model implemented in Mininet does not capture complex packet-loss schemes such as those caused by the user mobility in wireless networks. If not otherwise specified, throughout this dissertation, we use an Intel Core i7-7700 CPU with v5.10.10.11 Linux kernel and v2.14.1 Open vSwitch as our host machine.

Given that our testbed is public and well-integrated into *GenSync* middleware, users can independently experiment with the testbed's structure to overcome some of the above limitations. For instance, users can run the testbed on different computing platforms allowing for better emulation of their target CPUs (*e.g.*, mobile phones). Furthermore, *GenSync* testbed contains an additional component called **Runner** that exposes most of the *GenSync*'s experimental capabilities independently of the emulation technologies, which allows users to integrate *GenSync* with an independent emulation infrastructure. The resulting experimental observations would maintain their structure described earlier in Section 5.1, but now the results would come even closer to those that users can expect on their target platforms, contingent on the quality of the independent emulation.

For the most part, the complexity of the *GenSync* testbed is purposefully hidden and the users can utilize it through a simple configuration script where they declare the desired system conditions, while the rest of the experimentation process and reporting is automatized. We give an illustrative example of *GenSync* testbed configuration script in Listing 2.

```

# Protocol identifier
protocol=CPI
# Latency in milliseconds
latency=20
# Bandwidth in Mbps (in two directions)
bandwidth="10/25"
# Packet loss (percentage)
packet_loss=0.01
# Percentage of CPU cycles used for sync
cpu_server=100
cpu_client=20
# Repeat each experiment
repeat=100

```

**Listing 2:** *GenSync*'s testbed (benchmarking layer) configuration script example.

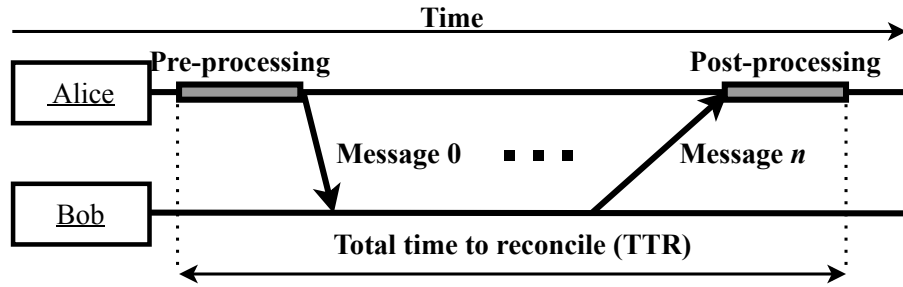
### 5.3 Experimental Performance Comparison

When comparing the practical performance of data sync protocols, we are primarily interested in the following two measures:

- (1) ***Communication cost***, defined as the amount of data transferred over the network link that connects Alice and Bob during the execution of a data sync protocol, and
- (2) ***Total time to reconcile*** (TTR), the total wall-clock time elapsed from the protocol initiation to its completion.

An intrinsic distinction between the two measures is that the former can be estimated analytically, while the latter depends significantly on the system parameters. In other words, communication cost is a function of *data parameters* (*i.e.*, data set cardinality and the amount of mutual differences, as discussed in Section 3.2), while TTR is also a function of system characteristics such as network latency, bandwidth, and compute capacity available at the nodes.

Suppose that Alice and Bob synchronize their local clocks at some time prior



**Figure 5-4:** Total time to reconcile (TTR) as the time elapsed between the protocol initiation and completion.

to data synchronization and that Alice plays the role of the protocol initiator while Bob receives the requests, we define TTR as the absolute difference between the time measured by Alice immediately before she initiates the protocol and the time measured by Bob or Alice when they learn the missing data (*i.e.*, the data points local to the other participant), whichever comes later.

Depending on the exact protocol in use, the participants may learn the missing data in different ways. For instance, in IBLT and CPI Alice can learn the missing data on her own through decoding, while in Cuckoo Bob computes this information and sends it to Alice. As plotted in Fig. 5-4, TTR comprises of real time needed for Alice and Bob to exchange all sync-related messages and perform certain computation during the exchange, which includes pre- and post-processing time. The pre-processing time is usually spent in creating the metadata that is being exchanged in the first message, while the post-processing typically involves handling the specialized data structures to discover the differences (decoding).

Note that our notion of TTR does not include the time needed for the parties to exchange the actual data. This is due to the fact that actual records can vary in length depending on the application or even the internal state of the participating nodes, while data points (`DataObjects`) are constant in size. Suppose that Alice and Bob are two adjacent Bitcoin nodes that sync the sets of their valid unconfirmed trans-

Config. Name	Latency ( <i>ms</i> )	Bandwidth ( <i>Mb/s</i> )	Packet loss (%)
<i>Mobile Broadband</i>			
<i>I</i>	30	18	$10^{-1}$
<i>II</i>	20	35	$10^{-3}$
<i>III(a)</i>	<b>1</b>	20	$10^{-1}$
<i>III(b)</i>	<b>1</b>	<b>0.1</b>	$10^{-1}$
<i>Consumer Internet</i>			
<i>IV</i>	10	60	$10^{-1}$
<i>IV(a)</i>	10	<b>12/16</b> (uplink/downlink)	$10^{-3}$

**Table 5.2:** Network configurations chosen to mimic both mobile broadband and consumer Internet networks. We consider both symmetrical and asymmetrical bandwidth Internet connections.

actions (*mempools*). Since transaction identifiers are generated by double SHA256 hashing [78], they are globally unique and of constant length (32 bytes), which makes them useful in `DataObject` construction. However, the size of the actual transactions that are identified by these hashes varies and can reach the size of the entire block [179].

In the rest of section, we use *GenSync*'s testbed to compare the communication cost and TTR of various sync protocols under realistic system conditions. For that purpose, we choose a collection of practically significant network parameters outlined in Table 5.2. Configurations *I* and *II* roughly emulate different mobile broadband networks and are derived from available measurement research [180], [181]. Configurations *III(a)* and *III(b)* model low-latency-low-bandwidth networks that will be of particular interest later in Section 5.3.4. Configurations *IV* and *IV(a)* model typical home Internet connections and their parameters are drawn from popular low-end residential Internet provider plans. Let us stress that the configuration *IV* emulates a *symmetrical* bandwidth whereas *IV(a)* emulates an *asymmetrical* bandwidth scenario that often occurs in DOCSIS [182] implementations.

Config. Name	Alice's CPU	Bob's CPU
$\mathcal{B}$	Apple A13 Bionic @2.7 GHz	Apple A13 Bionic @2.7 GHz
$\mathcal{C}$	Intel Core i7-7700 @3.6 GHz	Intel Core i7-7700 @3.6 GHz
$\mathcal{CA}$	Intel Core i7-7700 @3.6 GHz	Samsung Exynos 9810 @2.3 GHz
$\mathcal{CB}$	Intel Core i7-7700 @3.6 GHz	Apple A13 Bionic @2.7 GHz

**Table 5.3:** Compute configurations include both symmetrical (*i.e.*,  $\mathcal{A}$ , and  $\mathcal{B}$ ) and asymmetrical (*i.e.*,  $\mathcal{CA}$  and  $\mathcal{CB}$ ) compute scenarios.

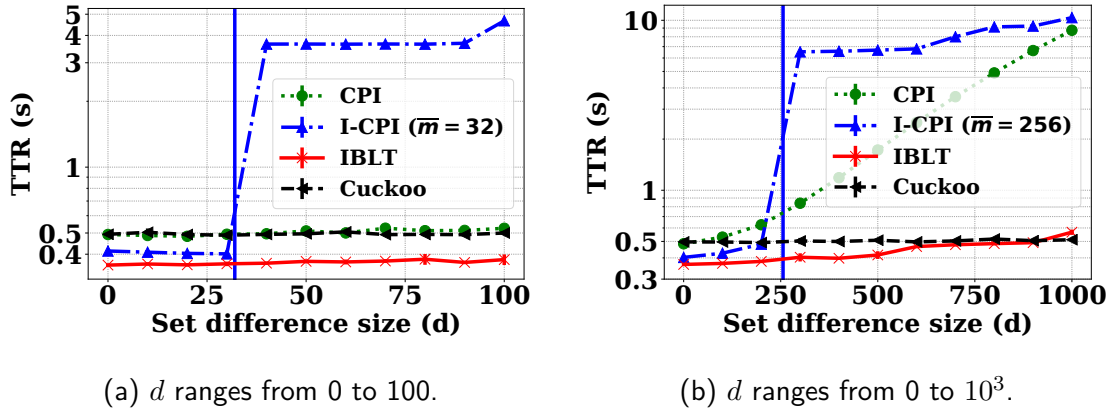
When it comes to modeling compute capabilities of nodes, we use the four compute configurations from Table 5.3. Importantly, we model both symmetrical and asymmetrical compute scenarios. The former refers to data sync process between the nodes of similar compute capacities (modeled by their CPU frequencies) whereas the latter refers to the scenarios when the devices have significantly different compute power. We use single-letter configuration names to denote the symmetrical and two-letter names for asymmetrical compute configurations. To denote a *complete system configuration* (CSC) consisted of a network configuration  $N$  and a compute configuration  $C$ , we use the ordered pair  $(N, C)$ .

### 5.3.1 Impact of Differences Size

Utilizing the *GenSync* testbed described in Section 5.2 and the system configuration parameters outlined earlier in this section, here we present our first set of experiments. We fix the set sizes, network conditions, and compute capabilities of Alice and Bob, while varying the number of mutual differences. As we generate mutual differences, we distribute them uniformly among the participants such that for  $d$  mutual differences each party keeps  $d/2$  of them. Each experiment is repeated 100 times and we report the measured quantities with 95% confidence intervals.

In system configuration  $(I, \mathcal{C})$  and for a small number of mutual differences relative to the sizes of the sets, the IBLT protocol performs the best with respect to the total time to reconcile metric (TTR). The CPI and Cuckoo protocols lag behind for around 40%, and the root causes for this effect differ depending on the protocol. On the one hand, Cuckoo is *non-optimal* in communication (*i.e.*, linear in the set size and not the number of differences, see Section 3.2), which causes a significant drop in the communication cost performance when compared to the other two protocols (see Fig. 5-6), especially when the ratio between the number of mutual differences and the size of the sets is low. On the other hand, CPI is indeed *communication-optimal*, thus the lag in performance should be attributed to its *computational complexity*, which is worse than IBLT's for roughly two orders of magnitude (*i.e.*, cubic in the number of mutual differences compared to the linear of IBLT).

As the number of mutual differences approaches 10% of the set size (*i.e.*,  $d = 10^3$ ), the TTR performance of Cuckoo and IBLT converge, as plotted in Fig. 5-5(b). This is primarily caused by the convergence of the protocols communication costs as the number of mutual differences increases. While IBLT has an  $O(d)$  communication complexity (see Section 3.2), its communication overhead per element is strictly larger than one, which makes it practically less communication efficient than CPI and also comparable in communication with Cuckoo when the number of mutual differences is comparable to the size of the sets. Therefore, as the communication cost of IBLT converges to that of Cuckoo (see Fig. 5-6(b)) so do their TTR performances. As discussed above, the CPI's deterioration in TTR performance with the increase of the number of mutual differences should be attributed to its computational complexity rather than its communication cost. In other words, when the number of mutual differences is sufficiently large, the main bottleneck of CPI is computation, which reflects to its overall TTR performance.

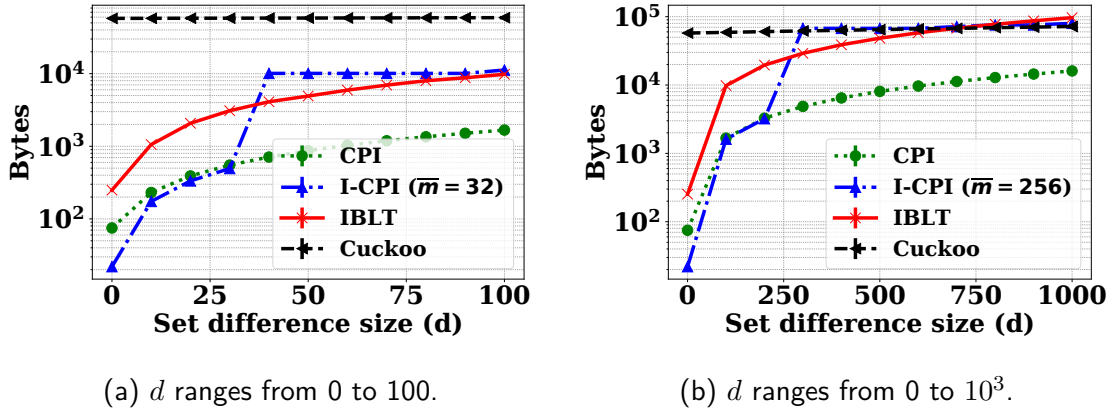


**Figure 5.5:** TTR when  $|S_A| = |S_B| = 10^4$  in the  $(I, C)$  configuration (log scale). The blue vertical line marks the position of  $\bar{m}$  in I-CPI.

Network			Compute	
Bandwidth (Mb/s)	Latency (ms)	Packet loss (%)	Alice's CPU	Bob's CPU
10 / 25 (uplink / downlink)	20	1	Intel Core i7-7700 @ 3.6 GHz	MediaTek MT6735 @ 1.3 GHz

**Table 5.4:** Illustrative system configuration.

Besides the overall computation and communication complexity, another significant factor that determines the practical performance of sync protocols is the *round complexity* (*i.e.*, the number of round trips required for the protocol to complete). This is particularly significant in the interactive protocols such as I-CPI, which uses multiple rounds of communication to eliminate the need for a precise estimate on the number of mutual differences. As plotted in Fig. 5.5, I-CPI drops in TTR performance when it fails to reconcile the sets in the first try and needs to exchange additional messages until it converges to a sufficiently large mutual differences bound. The magnitude of I-CPI's performance drop is primarily a function of network latency.

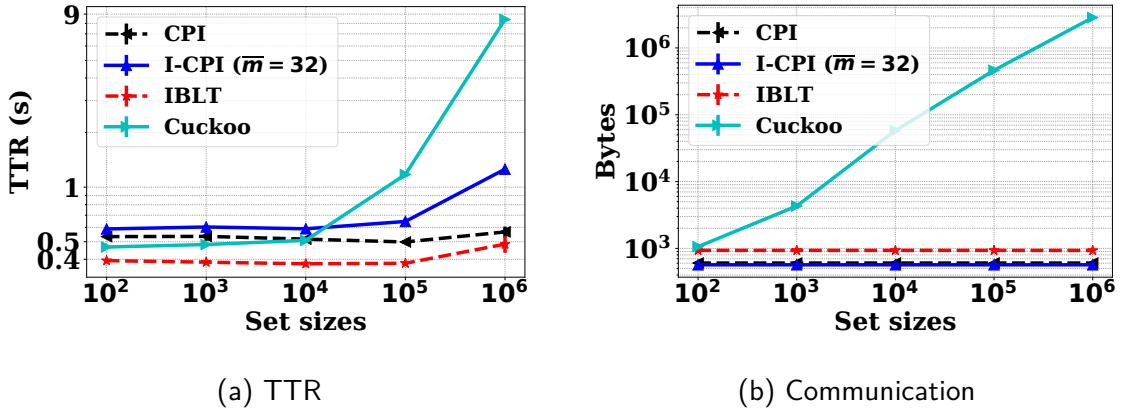


**Figure 5-6:** Communication cost when  $|S_A| = |S_B| = 10^4$  in the  $(I, \mathcal{C})$  configuration (log scale).

### 5.3.2 Impact of Data Cardinality

In our second set of experiments, we study the impact of the set sizes to the performance of the sync protocols from *GenSync*. Therefore, we set the network configuration to  $I$  (Table 5.3) and the compute configuration to  $\mathcal{C}$  (Table 5.2), while keeping the number of mutual differences,  $d$ , constant at 30. The resulting TTR performance and communication costs are plotted in Fig. 5-7.

As depicted in Fig. 5-7(b), for the set sizes ranging from  $10^4$  to  $10^6$ , IBLT dominates with respect to the overall TTR performance while the other protocols lag behind for 25-300%. Although all protocols deteriorate in TTR performance as the size of the sets increase, they do so at significantly different rates. For instance, the TTR performance of Cuckoo remains comparable to that of CPI-based protocols as long as the set sizes are small (roughly  $< 10^4$ ), while deteriorates drastically when the sets are large. On the other hand, the CPI-based protocols exhibit only a small loss in performance with the increase of set sizes, which is consistent with our analysis in Section 3.2. For sets larger than the critical size of roughly  $10^6$  and relatively small amount of mutual differences ( $d = 30$ ), the TTR performance of CPI approaches that of IBLT.

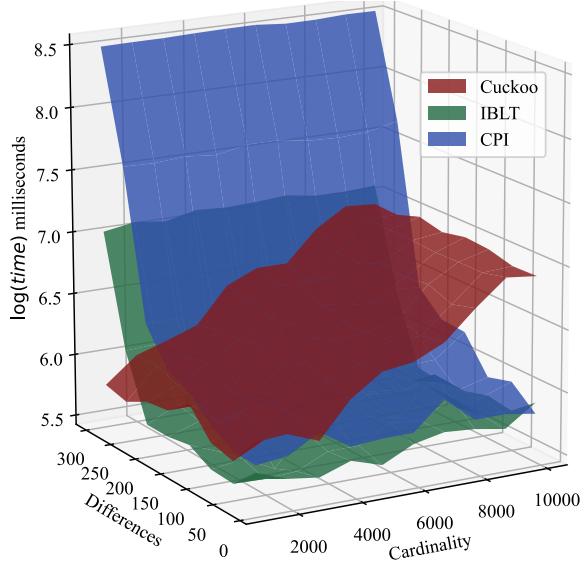


**Figure 5.7:** Difference size  $d = 30$ . Set sizes  $|S_A| = |S_B|$  vary from 100 to  $10^6$  (in  $(I, C)$  configuration). TTR and communication cost shown in log scale.

We summarize our conclusions about the joint effect of the set sizes and the number of mutual differences to the practical performance of the *GenSync* protocols using an illustrative system configuration from Table 5.4. As plotted in Fig. 5.8, there are two trends when it comes to the TTR performance:

- (1) The TTR performance of Cuckoo is largely *invariant* to the number of mutual differences.
- (2) The TTR performances of IBLT and CPI are largely *invariant* to the size of the data sets that are being synced.

In other words, Cuckoo performs relatively well for small sets with any number of mutual differences, while worsens significantly as the size of the sets increase. As depicted in Fig. 5.9, the performance degradation is primarily caused by Cuckoo's overall communication cost that increases in steps for many practical implementations (see Section 3.3.6 for a detailed discussion). On the other hand, IBLT and CPI perform better relative to Cuckoo for very similar sets regardless of their size. As long as we can expect a high similarity among sets (*i.e.*, sets differing in a few elements), the overall performance of IBLT and CPI will dominate Cuckoo.

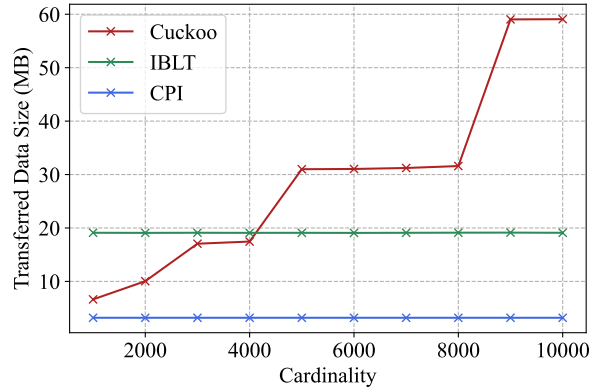


**Figure 5-8:** Logarithm of TRR as a function of set cardinality (size) and the number of differences (d). Illustrative complete system configuration from Table 5.4.

### 5.3.3 Impact of System Parameters

Next, we discuss the series of experiments that demonstrate the impact of system parameters to the relative performance of the *GenSync* protocols. For that purpose, we fix the size of the syncing sets to  $10^4$  and consider a small number of mutual differences relative to the size of the sets (*i.e.*, from 0 to 100). Regarding the compute parameters, we emulate several common sync scenarios: (1) server-to-server sync (*i.e.*, compute configuration  $\mathcal{C}$  from Table 5.3), (2) sync between a low-end smartphone and a server (*i.e.*,  $\mathcal{CA}$ ), and (3) a high-end smartphone and the server (*i.e.*,  $\mathcal{CB}$ ). When it comes to the network parameters, we use  $I$ ,  $II$ ,  $IV$ , and  $IV(a)$  from Table 5.2. Combining these compute and network parameters, we get a collection of complete system configurations (CSC). Our primary goal is to identify the major network and compute parameters that determine the best *GenSync* protocol for each CSC.

Considering the results from Fig. 5-10 and comparing configurations  $(I, \mathcal{C})$  and  $(I, \mathcal{CA})$  with their counterparts  $(II, \mathcal{C})$  and  $(II, \mathcal{CA})$ , respectively, we observe that

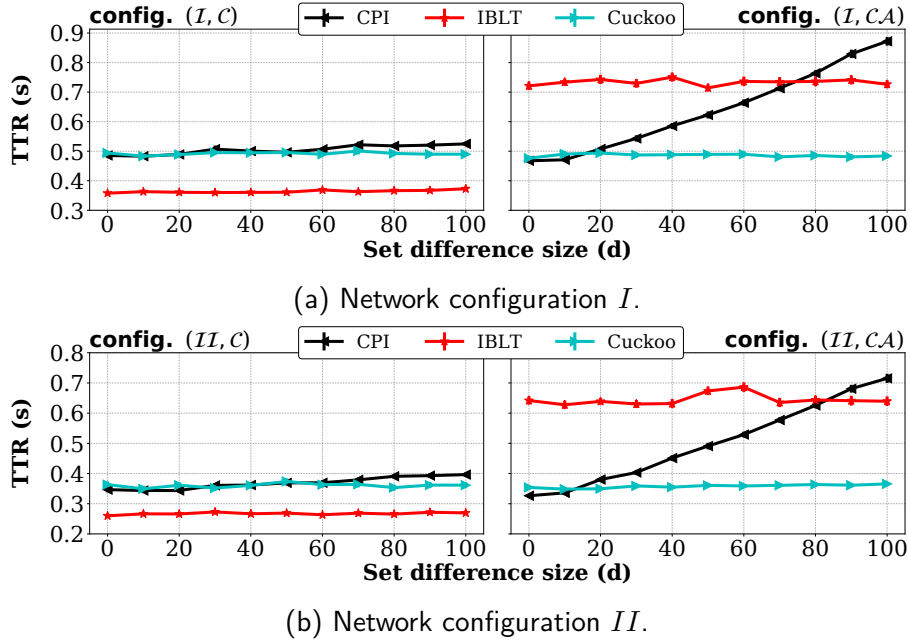


**Figure 5-9:** The step increase in communication cost for Cuckoo as the cardinality increases. The number of mutual differences is constant at 100.

the difference in network performances in configurations *I* and *II* have a significant effect on the performance of the *GenSync* protocols. All three protocols that we consider in Fig. 5-10 improve their average performance under better network bandwidth (*i.e.*, the network configuration *II*). The improvement ranges from 16% for IBLT in compute configuration  $\mathcal{CA}$  to 43% for Cuckoo in compute configuration  $\mathcal{C}$ .

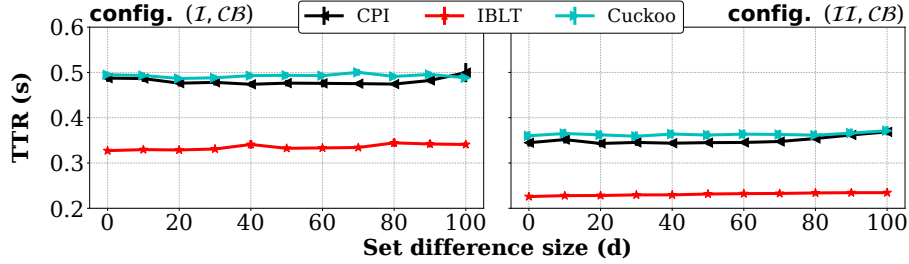
Even more significant is that there is no dominant sync protocol across all examined compute configurations. In the case of the sync between two servers (*i.e.*, *symmetrical* compute configuration  $\mathcal{C}$ ), IBLT has the best TTR performance among the considered protocols (see left side of Fig. 5-10). Yet, in the case of the sync between a server and a low-end smartphone (*i.e.*, a highly *asymmetrical compute* configuration  $\mathcal{CA}$ ), Cuckoo overtakes IBLT for all the considered mutual differences counts and both network conditions (*i.e.*, *I* and *II* from Table 5.2). For very small number of mutual differences (*i.e.*, between 20 and 100), Cuckoo's TTR performance is comparable to CPI's, however, CPI degrades much faster with the increase in differences count.

Cuckoo's dominance in the conditions of *highly* asymmetrical compute (*i.e.*,  $\mathcal{CA}$ ) can be attributed to the following factors: (1) Cuckoo filters decode faster than IBLTs, which makes Cuckoo suitable for less capable compute platforms (*e.g.*, IoT sensors



**Figure 5.10:** The impact of the network performance and available computing power to TTR-performance. Symmetrical compute (left) versus asymmetrical compute (right).

and low-end smartphones), and (2) the size of the sets being small relative to the available bandwidth, which amortizes the Cuckoo protocol’s bandwidth inefficiency. As we discussed previously in Section 3.3.6, the decoding process of Cuckoo filters consists of looking up each represented element by its *fingerprint*, which is typically implemented by a constant number of hash computations (*e.g.*, one for the fingerprint and two more for the two *candidate buckets*). On the other hand, the decoding process of IBLT consists of a computationally more expensive procedure commonly referred to as the “peeling” process. Knowing that in the case of asymmetrical compute the average TTR performance is bounded by the compute power of the less capable node, we conclude that the difference in the decode procedure complexity between Cuckoo filters and IBLTs determines the best protocol. On the other hand, as long as the network bandwidth is not a bottleneck to the overall TTR performance, the relative difference in the encoding size between Cuckoo and IBLT does not have a decisive



**Figure 5-11:** The impact of the network conditions to TTR-performance for the  $\mathcal{CB}$  compute configuration. Network configuration  $I$  (left) versus network configuration  $II$  (right).  $\mathcal{CB}$  has a 3.5 times *smaller* compute power discrepancy than  $\mathcal{CA}$ .

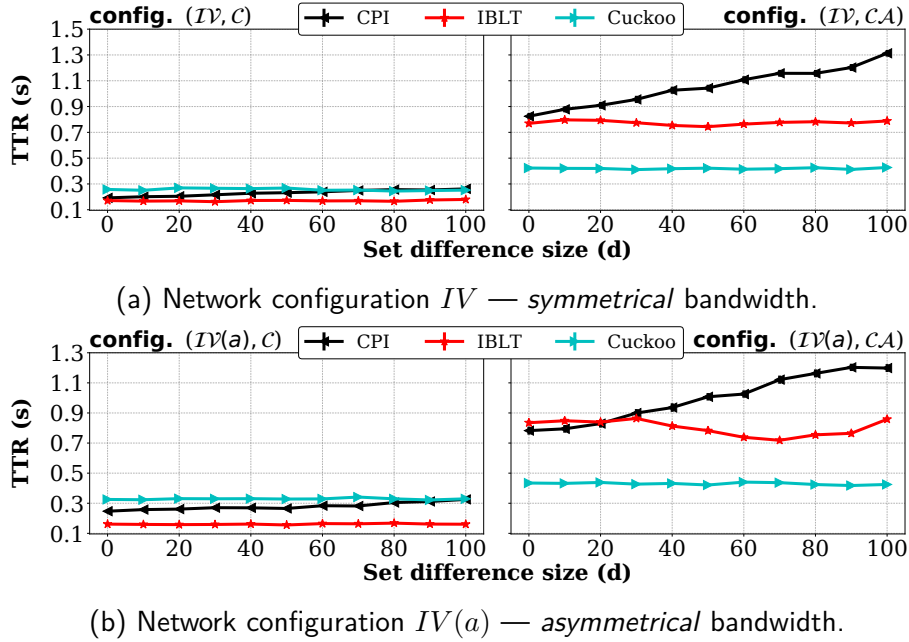
effect.

We want to stress that only *sufficiently large* discrepancies in the compute power at the nodes decide the best protocol. As opposed to the compute configuration  $\mathcal{CA}$ , the configuration  $\mathcal{CB}$  does not alter the dominant protocol (see Fig. 5-11). The reason lies in the fact that the compute discrepancy in the  $\mathcal{CB}$  configuration is 3.5 times smaller than in the case of  $\mathcal{CA}$ .

Unlike the asymmetrical compute power at the nodes, an asymmetrical network bandwidth does not have a significant impact on the best protocol choice, unless the bandwidth in one direction is a bottleneck. We support this conclusion with the experimental results in Fig. 5-12, and discuss the cases when the bandwidth in one direction indeed is a bottleneck to the overall TTR performance in the next section.

### 5.3.4 Impact of Very Low Bandwidth

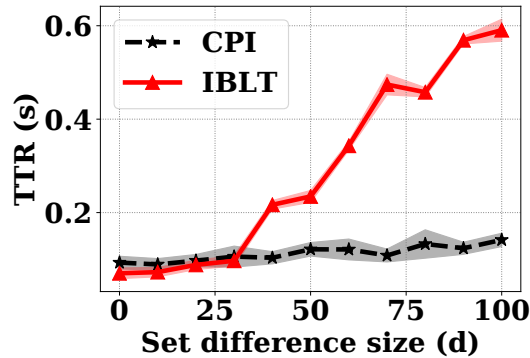
As mentioned in the previous section, an asymmetrical bandwidth between the nodes involved in the sync process does not decide the best protocol (*i.e.*, the best performing protocol among those included in *GenSync*), *unless* the bandwidth in one direction drops under a critical point. In this section, we analyze these exceptional cases when changes in the available bandwidth *do* decide the best protocol. To this end, we construct two low-latency network configurations  $III(a)$  and  $III(b)$  from Ta-



**Figure 5-12:** The impact of asymmetrical *network bandwidth* to TTR-performance. Symmetrical *compute* (left) and asymmetrical *compute* (right).

ble 5.2 that differ only in the amount of the bandwidth that they offer to the sync process, and run our experiments under the constant compute configuration  $C$ , while keeping the set sizes at  $10^4$ . The two network configurations emulate the practical sync scenarios when one of the devices involved in the sync process alternates between allocating the entire available bandwidth to the sync process and imposing a strict bandwidth budget to the sync process to maximize the performance of some other user-experience critical applications (*e.g.*, streaming point clouds in the Augmented Reality (AR) applications [183]).

In the case of the network configuration  $III(a)$ , the TTR performance of IBLT and CPI remain comparable to each other varying between 60 and 140 ms for the differences count between 1 and 100. However, when the amount of bandwidth allocated for the sync process is bounded by  $0.1Mb/s$  (*i.e.*, the network configuration  $III(b)$ ), IBLT's overall TTR performance degrades significantly relative to that of



**Figure 5-13:** TTR for network configuration *III(b)* (the lower the better).  $|S_A| = |S_B| = 10^4$ . 95% confidence intervals shaded. 100 iterations per point.

CPI (see Fig. 5-13). Although IBLT’s performance keeps up with that of CPI for very small numbers of mutual differences ( $d < 30$ ), for the larger differences counts ( $30 \leq d \leq 100$ ), CPI dominates, and at  $d = 100$  reaches a 5 times better performance than IBLT. The CPI’s dominance over IBLT under the system parameters in Fig. 5-13 can be attributed to CPI’s nearly optimal communication complexity [85], which is better than that of IBLT by a constant factor [64], [87]. When the bandwidth is the bottleneck, CPI’s advantage with respect to the communication cost turns into a sizable dominance with respect to the overall TTR performance.

Note that we do not plot the performance of the Cuckoo protocol in Fig. 5-13, as the TTR performance of Cuckoo under extremely low bandwidth conditions cannot keep up with the other two protocols. As discussed earlier in Section 3.2, Cuckoo has a communication cost proportional to the sizes of the sets, which makes it comparably inefficient when the bandwidth is the bottleneck and the count of mutual differences is small relative to the set sizes.

## 5.4 Benefits of Interactive Protocols

As we hinted previously in Section 2, data sync protocols are a widely adopted tool in distinct families of distributed systems ranging from ultra-long battery life wireless sensors to data center storage servers. Due to this diversity in applications that rely on sync protocols, many different *styles of usage* have been developed over time. Especially interesting are the two styles of usage developed to support repeated synchronization: (1) *cold start*, and (2) *incremental*.

In the case of the *cold start* sync, the syncing parties create the sync-related data structures (*e.g.*, efficient set representations) prior to each sync. This is often the case when the application requires consistent data in relatively long time intervals (regular or irregular). The reasoning behind the design decision to recreate the sync-related data structures prior to each sync is often motivated by a relatively small amount of memory (*e.g.*, RAM) on the syncing devices (*e.g.*, IoT). Instead of occupying the scarce memory with the sync-related data structures for long periods of time between two syncs, the application designers often decide to drop the unused data structures and release the memory.

On the other hand, in the case of *incremental* sync, the syncing parties keep the sync-related data structures between subsequent syncs. This is often the case with applications that require relatively high level of data consistency and thus sync very often, or even as often as possible (*i.e.*, as soon as the previous sync has completed). However, not all sync protocols included in *GenSync* support *incremental* sync equally well, primarily because it requires the underlying data structures to be *dynamic*, *i.e.*, to support dynamic set representations.

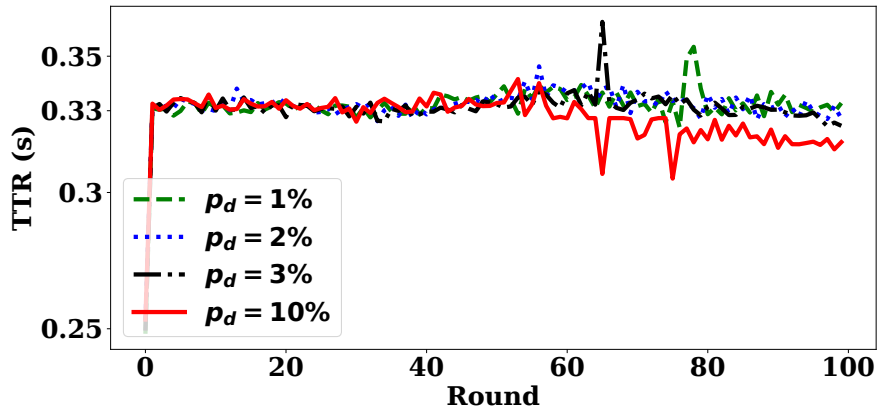
One of the protocols that is well suited for both incremental and cold start sync is CPI. Therefore, we produce a series of experiments for comparing the effectiveness of cold start and incremental syncs using the examples of CPI and its *interactive*

counterpart I-CPI. As we discussed earlier in Section 3.1.3, the main difference between the two is that I-CPI maintains its reconciliation state over time. This state is represented as  $p$ -ary tree that evolves following the set it represents. The initial  $p$ -ary tree is created at the beginning of the protocol and maintained such that each subsequent sync can reuse it.

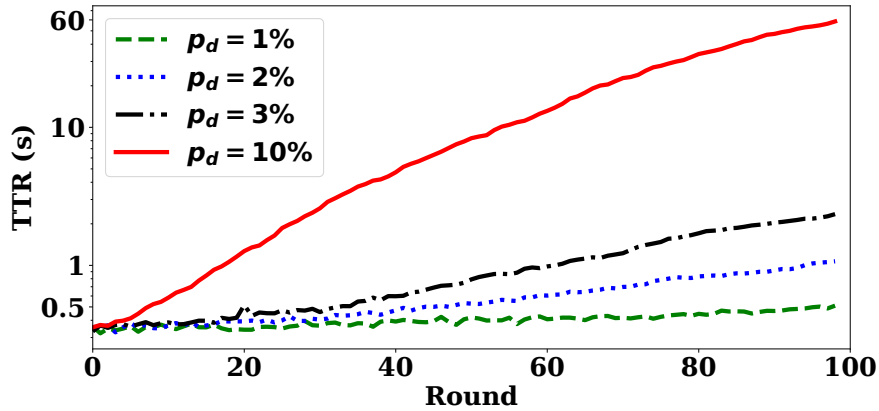
Suppose that we have two syncing parties, Alice and Bob, that constantly receive new data from outside and agree to sync after each  $N = 100$  data points are added on each side, and stop when their set sizes reach  $10^4$  elements (*i.e.*, after 100 rounds of syncing). To aid with our evaluation, we make two simplifying assumptions: (i) we assume that additions to  $S_A$  and  $S_B$  happen in parallel at roughly the same time, and (ii) a data point  $x$  that is added to  $S_A$  is either new to Alice (*i.e.*,  $x \notin S_A$ ), or new to both parties (*i.e.*,  $x \notin S_A \cup S_B$ ). Likewise is the case for each  $y$  that is added to Bob's set (*i.e.*,  $S_B$ ). In other words, common elements are not being added. We define  $p_d$  to be the probability that any pair of data points  $(x, y)$ , where  $x$  is the data point being added to  $S_A$  and  $y$  is being added to  $S_B$ , is new to both parties (*i.e.*,  $(x, y) \notin S_A \cup S_B$ ). In other words,  $p_d$  is the probability that the pair  $(x, y)$  will end up in the set of symmetric differences.

We evaluate the TTR performance of CPI and I-CPI for various values of  $p_d$  while keeping our *GenSync* testbed in configuration  $(II, \mathcal{C})$  for all experiments. As we can see from the results in Fig. 5-14, the TTR performance of I-CPI is roughly constant in all rounds of the experiment and does not vary significantly for various values of  $p_d$ . In other words, as long as the number of new data points that arrive per unit of time is constant, I-CPI exhibits a stable TTR performance regardless of the increasing size of the sets.

On the other hand, the TTR performance of the I-CPI's non-interactive counterpart (CPI) is way worse when it comes to incremental reconciliation. The TTR



(a) I-CPI



(b) CPI (log scale)

**Figure 5-14:** TTR in each round of incremental sync. System configuration  $(II, \mathcal{C})$ .

performance of CPI deteriorates as a function of the sets size and  $p_d$ . This effect is caused by the fact that CPI does not reuse the sync-related data structures and thus cannot benefit from the differences that have already been synced in the previous rounds, which reflects as a cumulative increase in TTR across rounds. Additionally, the more differences arrive between two syncs, the larger is the increase of TTR, which shows as a deterioration in TTR as a function of  $p_d$ .

Another notable disadvantage of CPI over I-CPI is the fact that I-CPI does not

require an accurate initial estimate on the upper bound of symmetric differences ( $d$ ) to succeed (see Section 3.1.3 for a detailed analysis). Although in our experimental setup it may appear that  $p_d$  can be used to derive the upper bound on  $d$ , in practical applications  $p_d$  may not be known to the syncing parties.

Finally, *GenSync* contains more protocols that support incremental sync. However, some of them may suffer from a degradation in sync accuracy when utilized in incremental sync scenarios. For instance, IBLT protocol can also be used in the incremental mode such that Alice and Bob keep adding their newly arrived data into their corresponding IBLTs over time. However, as we discussed in Section 3.1.1, the newly inserted data points may cause IBLT decode failures, should the overall number of the elements in the IBLT exceed the designed threshold. Therefore, one should assure that the IBLTs created at the beginning of an incremental sync have a sufficient capacity to endure for certain amount of time (under certain  $p_d$ ). Alternatively, one can decide to periodically rebuild the IBLTs to amortize the loss in they sync accuracy. As opposed to IBLT, I-CPI naturally supports incremental sync with no losses in the sync accuracy as more data arrives in the underlying sets.

## 5.5 Bitcoin Data Set Evaluation

In this section, we demonstrate the practicality of *GenSync* for comparative analysis between the state-of-the-art data sync protocols under realistic system conditions using the data from the Bitcoin blockchain [78]. We expect that this application scenario can be used as a template for evaluating the practical performance of data sync in other applications of interest.

***Application Scenario:*** The following experiments consider synchronization between two *adjacent* Bitcoin *full* nodes. Meaning that both nodes engage in the process of Bitcoin transaction dissemination and keep transaction pools (see Section 2.4.1). As

previously mentioned, transactions are uniquely identified by their 32-bytes long hash obtained through double `SHA256` hashing [184]. Essentially, transaction pools are the sets of 32-bytes long integers that are uniformly distributed over  $GF(2^{256})$ .

Our application scenario is motivated by the recent experimental results of Imtiaz *et al.* [43] that have demonstrated a potential of transaction pool synchronization for improving Bitcoin’s performance, especially in presence of long node churns (*i.e.*, node offline periods). Specifically, they have shown that the nodes suffering from long churns often miss a relatively large number of transactions disseminated through the network during their off periods. Consequentially, the churning nodes are more likely to fail to reconstruct the newly created blocks once they are back online. Imtiaz *et al.* propose a solution called *MempoolSync* that uses a heuristic approach to synchronize the mempools of adjacent Bitcoin nodes once when they return back online, and before they receive any new blocks. More specifically, a node returning to the network would contact a node that has been online for long time to receive the missing transactions. The authors implemented *MempoolSync* in the Bitcoin software and demonstrated a decrease in the average block propagation time at the nodes that employ *MempoolSync* when compared to those that do not. However, the authors did not attempt to make their solution efficient, and suggested replacing their heuristic scheme with an efficient data sync protocol. Given that *GenSync* is well suited for comparing state-of-the-art data sync protocols, we use it to explore the potentials of different protocols in the context of syncing mempools of two adjacent Bitcoin nodes.

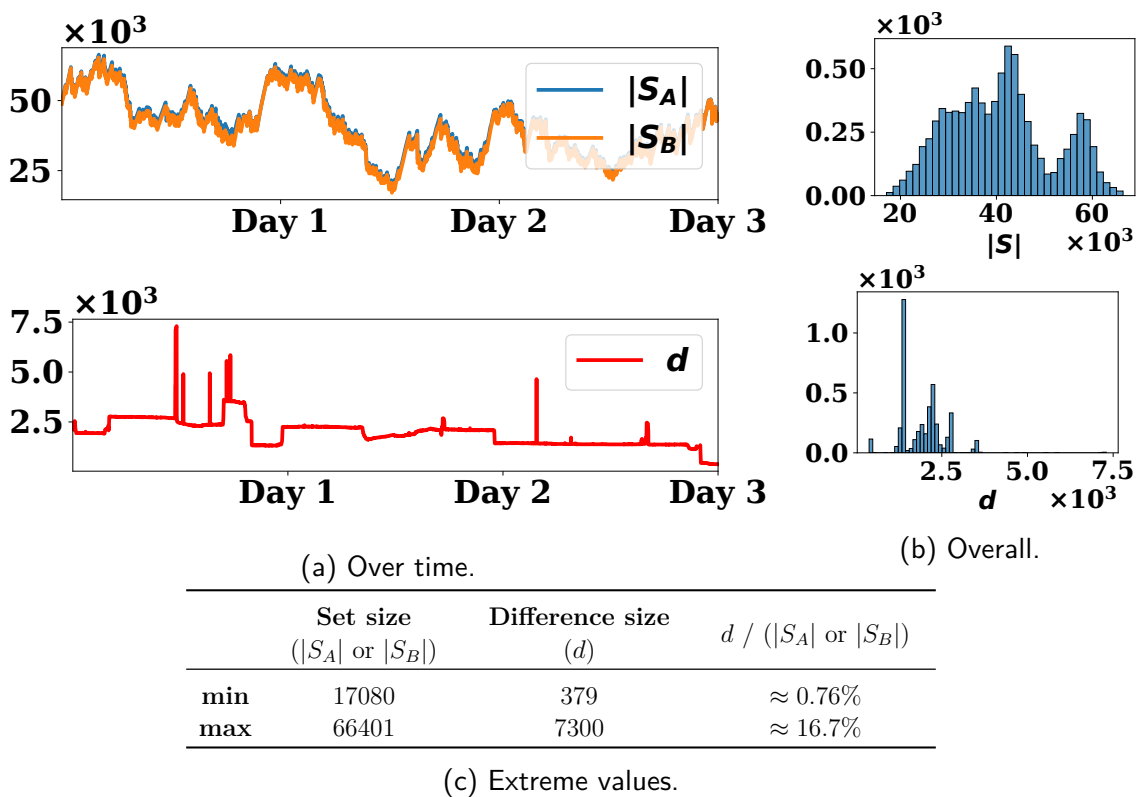
For the purpose of this series of experiments, we set *GenSync* testbed to the symmetrical compute configuration  $(I, \mathcal{C})$  (see Section 5.3, specifically Table 5.2 and Table 5.3). Our decision to model a symmetrical compute scenario is primarily motivated by the fact that it roughly corresponds to a typical Bitcoin setup at the moment of writing. As for the sync style, we do cold start sync, since that allows us to compare

more protocols under similar accuracy assumptions. As we discussed in Section 5.4, some protocols (*e.g.*, IBLT) may suffer from an accuracy degradation in incremental sync. In each experiment, we execute a sync protocol on the pair of sets described below, and report TTR and communication cost with 95% confidence intervals over 100 experiment iterations.

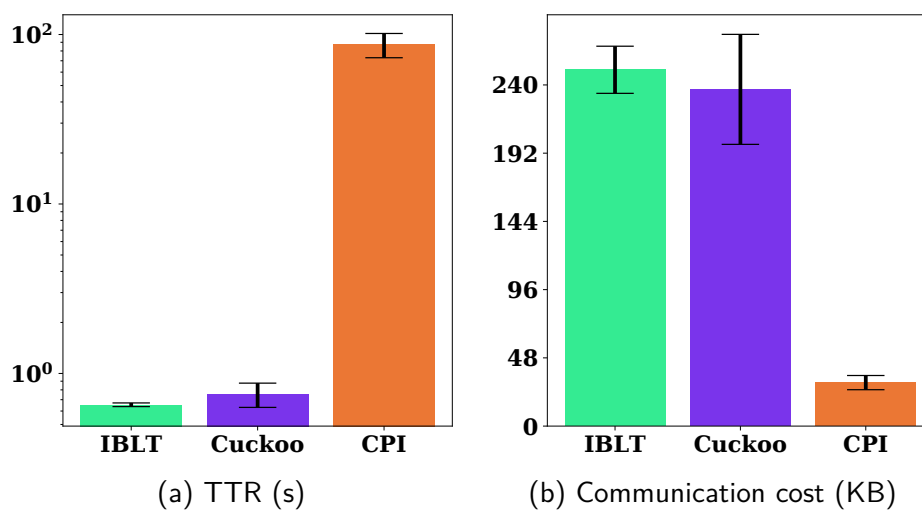
**Data Collection:** We collect the data for our experiments through a three days long measurement campaign on two time-synchronized Bitcoin nodes participating in the main Bitcoin network. The measurement campaign took place in April 2021 and we collected the snapshots of the two transaction pools each minute. Altogether, we collected 4320 pairs of sets ( $S_A$  and  $S_B$ ) where each set represents a collection of identifiers of unconfirmed Bitcoin transactions (*i.e.*, 32-byte long hashes). We plot the summary of our measurement campaign in Fig. 5-15. The mean sizes of the two mempools are 40835 transactions, and the mean count of their mutual differences ( $d$ ) is 1930. In other words, *MempoolSync* deals with large data sets that differ in only about 4.7% of their elements. Notably, the standard deviations of the set sizes and the number of their mutual differences are relatively high: 10654 and 656, respectively.

**Results Analysis:** As depicted in Fig. 5-16(a), the IBLT protocol has the best mean TTR performance among the three protocols that we examined (around 600ms with a very low variance). Applied to syncing two adjacent Bitcoin nodes, these results may suggest a rough estimate on the maximum frequency of mempool synchronization. Roughly speaking, since IBLT achieves full synchronization within a second with a high probability on a typical Bitcoin setup, full Bitcoin nodes may decide to sync their mempools each second, which would lead to a high level of consistency among them.

However, as plotted in Fig. 5-16(b), IBLT causes a substantial amount of bandwidth overhead when compared to other two protocols. Therefore, there is a trade-off



**Figure 5-15:** Distribution of *mempool* sizes ( $|S_A|$  and  $|S_B|$ ) and symmetric difference sizes ( $d$ ).



**Figure 5-16:** *GenSync* protocols applied to *mempool* reconciliation. Configuration ( $I, \mathcal{C}$ ). 95% confidence intervals.

between the maximum frequency of syncing and bandwidth overhead. The larger the maximum frequency of syncing, the more bandwidth overhead the protocol incurs. We want to stress that our results from Fig. 5.16 apply to our system model represented by the *GenSync* testbed configuration  $(I, \mathcal{C})$  and may vary significantly in other practical system conditions, as we previously discussed in Section 5.3.3.

It is also important to emphasize that the Cuckoo protocol exhibits a lower communication cost when compared to IBLT for this data set. Considering only the theoretical bounds from Section 3.2, this may seem surprising, as the communication complexity of the Cuckoo protocol is  $O(S_A + S_B)$ , while that of IBLT is only  $O(d)$ . Theoretical intuition suggests that for huge sets that differ in only a small number of elements (*i.e.*,  $d \ll S_A + S_B$ ), IBLT should exhibit dominant communication performance. However, in accordance with our discussion from Section 5.3.2, as the number of differences approaches 10% of the set size, the communication costs of IBLT and Cuckoo converge. Therefore, for the estimated maximum sync frequency of 1 second, Cuckoo may be a plausible alternative to IBLT, especially when we want to prioritize bandwidth efficiency. Following our publication of *GenSync* [42], Ding *et al.* [67] have designed a block propagation protocol that relies on this competitive advantage of Cuckoo filters against IBLTs, which have been used as a building block of the block propagation protocols that Ding *et al.* compared against [23]. Yet, if we want to maximize the bandwidth efficiency, the CPI protocol should also be considered as a potential alternative to IBLT and Cuckoo. As plotted in Fig. 5.16(b), for the typical data parameters of the Bitcoin blockchain, CPI is far more conservative about the bandwidth usage than the other two examined protocols. On top of that, as we discussed earlier in Section 5.4, I-CPI (the CPI’s interactive counterpart) can be efficiently utilized in *incremental* reconciliation to further improve the TTR performance. However, this optimization comes at a cost of additional rounds of

communication, which may negatively effect TTR in high-latency networks.

## 5.6 Case Study — Synchronization at the Edge

In the systems akin to Distributed Computing Continuum Systems (DCCS) [40] that we touched upon in Section 1.3, there may be natural causes of system parameter variations. For instance, the node mobility in cellular networks may cause variations in network bandwidth [185]–[188], while different QoS-related decisions may result in constraining the amount of the compute available to data sync at certain nodes, contingent on their current load. While some system parameter variations may have only a small impact on the overall system function (*e.g.*, small drops in network bandwidth), some may even cause the service disruptions (*e.g.*, user entering an out-of-coverage area in cellular networks [189]).

In contrast to the the prior sections where we primarily dealt with constant system parameters, here we consider dynamic systems whose system parameters vary within some reasonable bounds. With that in mind, we are primarily interested in assessing the *average* performance of *GenSync* protocols under evolving network parameters caused by user mobility, and to that end, we designed a series of experiments for Colosseum [172], one of the world’s largest wireless network emulators.

In contrast to Mininet [173], [190] emulation, where we set the desired transport-layer performance parameters (*e.g.*, latency and bandwidth), Colosseum [172], [191] allows for more accurate emulation of the network’s physical layer through SDR technology [192], [193]. Colosseum emulator consists of 128 Standard Radio Nodes (SRN) connected to a Massive Digital Channel Emulator (MCHEM) that emulates the wireless channel using FPGAs. Each SRN is equipped with a Dell R730 server and a Ettus Research Universal Software Radio Peripheral (USRP) model X310 that connects to MCHEM. On top of the SRN hardware is a system container and virtual

Scenario Regime	User Speed	Base Station Distance	Scenario Duration
Stationary	0 m/s	20 m	600 s
Pedestrian	5 m/s	20 m	600 s

**Table 5.5:** Emulation parameters for the Colosseum’s Boston cellular scenario [172].

machine manager that supports Linux containers (LXC) [194]. Within each LXC container, a Colosseum user can install the `srsRAN` software [195] to interact with the USRP hardware and other tools that allow for MCHM configuration (*e.g.*, the SCOPE framework [191]).

### 5.6.1 Colosseum Scenarios

Colosseum can emulate various realistic wireless settings that Bonati *et al.* [191] named the “Colosseum scenarios”. For the purpose of our experiments, we are particularly interested in the *cellular scenarios*. A cellular scenario is a configuration of the Colosseum emulator that mimics a real-world cellular network in some area. In a Colosseum cellular scenario, each base station and each user is modeled using a dedicated SRN, which runs the appropriate SDR component from the `srsRAN` software, and the following aspects are captured in the simulation: (1) number of base stations, (2) number of users and which base stations they connect to, (3) user distance from the base station, and (4) user mobility. With respect to users’ distance from the base stations, there are three options to choose from: close (within 20m), medium (within 50m), and far (within 100m). As for the mobility patterns, the users can be stationary or they can move at 3 m/s (slow movements), or 5 m/s (average pedestrian speed).

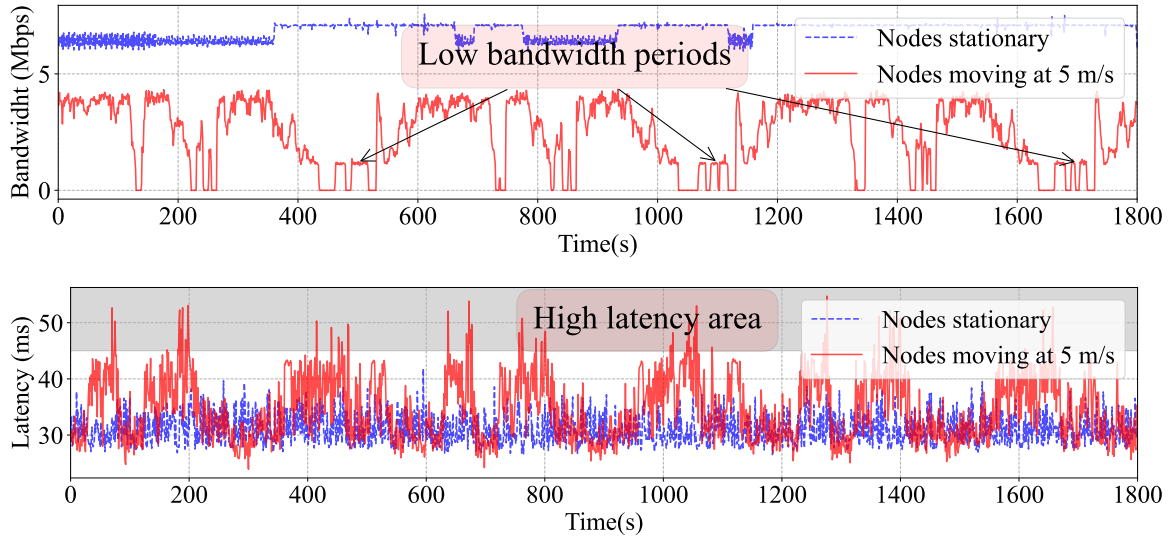
## “Boston” Scenario

The “Boston” scenario is one of the Colosseum’s cellular scenarios that captures the cellular network in the surroundings of the Boston Common public park in downtown Boston, Massachusetts. It consists of 10 base stations and 40 users spread across them with a constant number of users per base station. It supports several regimes including the two from Table 5.5 that we use in our experiments. Importantly, the users are always within 20 meters from their corresponding base stations such that they never switch the base station that they use. Therefore, we expect the link between any two nodes corresponding to the same base station to exhibit a similar performance with respect to the transport-layer measures that we discuss next. One scenario iteration lasts for 600 seconds, after which we restart it with the same parameters.

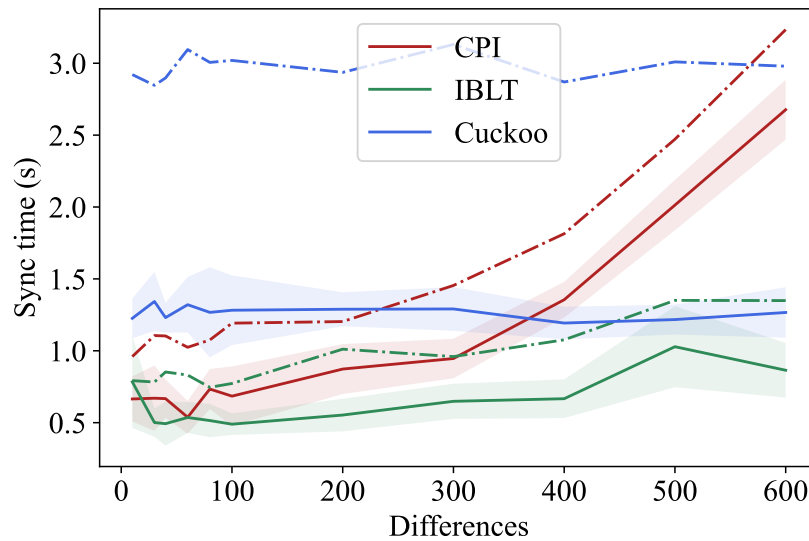
### 5.6.2 Measuring Transport-layer Variations

To obtain the bandwidth and latency traces in the Colosseum’s Boston scenario, we randomly pick two UE nodes connected to the same base station and measure the transport-layer latency and bandwidth between them each second. We plot the resulting traces in Fig. 5-17, where the upper part is the bandwidth and the lower is the latency trace over roughly three repetitions of the scenario. The traces show that user movements result in larger oscillations in available bandwidth and latency. Using the average values of bandwidth and latency during the extreme periods annotated in Fig. 5-17, we define two sets of network conditions against which to evaluate our sync protocols:

- (1) *bad* (bandwidth 1 Mbps, latency 50 ms), and
- (2) *good* (bandwidth 7 Mbps, latency 30 ms).



**Figure 5-17:** Bandwidth (up) and latency (down) traces from Colosseum [172], one of the world’s largest wireless network emulators.

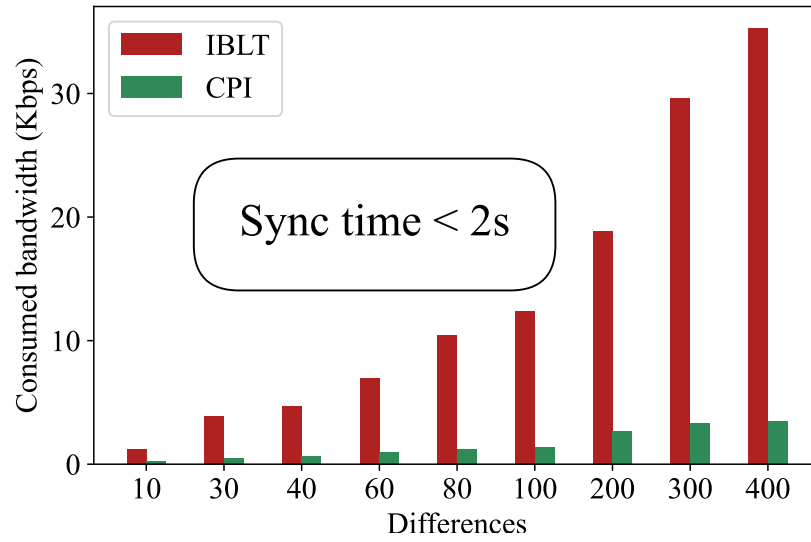


**Figure 5-18:** Sync time for the three main sync protocols in *good* (solid line) and *bad* (dashed line) network conditions. Data cardinality is  $10^8$ . Confidence intervals are shaded.

### 5.6.3 Emulation Results and Takeaways

We are interested in comparing the practical performance of the *GenSync* protocols under the two extreme network configurations defined in the previous section. As for the data parameters, we let Alice and Bob both keep sets of  $10^8$  elements and distribute the mutual differences uniformly among them such that each party contains exactly the half of total differences. We vary the number of total differences between 10 and 600 and measure the total sync time (*i.e.*, TTR). Each experiment is repeated 1000 times and we report TTR with a 95% confidence interval. As shown in Fig. 5-18, IBLT sync performs the best among the compared protocol, beating the other two protocols for both bad and good network conditions. Moreover, it exhibits relatively small performance degradation in bad network conditions. The reason for this effect lies in IBLT’s balance of low computational complexity (linear in the number of differences) and communication cost that, although higher than CPI [85], beats Cuckoo for large data sets. As the average bandwidth in both good and bad network conditions does not drop below a critical point, IBLT also beats CPI.

However, an application-specific protocol that is being constructed using *GenSync* may have additional performance objectives. For instance, it may want to be conservative about the amount of *consumed bandwidth*, while still maintaining some reasonable sync performance (even under bad network conditions). Since the synchronizing device may run several applications concurrently, this kind of bandwidth budgeting could be an important dimension to consider when designing an application-specific sync protocol. Bandwidth savings in the sync protocol could generate substantial performance gains for Quality-of-Experience-critical processes, such as streaming point clouds in augmented reality (AR) applications [183]. In Fig. 5-19 we plot the amounts of bandwidth that IBLT and CPI consume when sync time is constrained to two seconds. In this case, CPI achieves almost nine times better per-



**Figure 5-19:** Bandwidth consumed for the two most bandwidth-efficient sync protocols that still complete under 2 seconds in *bad* network conditions.

formance across all difference counts. The reason for CPI's dominance over IBLT in this scenario is CPI's nearly optional communication cost, while IBLT adds a multiplicative constant to that cost.

## Chapter 6

# Conclusion and Future Work

In this dissertation, we studied the problem of data synchronization in heterogeneous decentralized systems. We showed that the performance of data synchronization can be improved by utilizing the knowledge of the underlying system and the statistics of the data that is being synchronized. Based on these findings, we propose novel tools and protocols.

Our main contributions are as follows. First, we studied the analytical properties of the state-of-the-art end-to-end set reconciliation protocols, and in doing so we: (i) performed a comprehensive comparison among the protocols with respect to computational, communication, and message complexity, (ii) discussed the practical ramifications of common theoretical assumptions, and (iii) investigated the impact of implementation decisions to the real-time performance of state-of-the-art set reconciliation protocols. Next, we proposed a decentralized solution to network-wide synchronization based on *communication-efficient* end-to-end set reconciliation, and to that end we (i) introduced a distributed algorithm called *SREP* that performs network-wide synchronization of transaction pools in large-scale blockchains, (ii) analyzed *SREP*'s performance in general graphs and networks that resemble real-world public blockchains such as Ethereum, and (iii) showed that *SREP* outperforms similar approaches from the literature. Finally, we introduced *GenSync*, a middleware that includes a variety of state-of-the-art end-to-end set reconciliation protocols and (i) allows for an easy integration into existing applications, (ii) can estimate the real-

time performance of data synchronization protocols given the performance model of the underlying system and the statistics of data, and (iii) seamlessly integrates with emulation platforms to assess the synchronization performance in complex network scenarios.

In the rest of this section, we summarize our contributions, and then outline the future research directions in the area of data synchronization for heterogeneous decentralized systems.

## 6.1 Contributions Summary

***Comparative analysis of analytical properties.*** In Chapter 3, we compared several set reconciliation protocols from the two families: *approximate set membership data structures*-based, and *coding theory*-inspired. From the former family we included IBF and Cuckoo filter-based variants, while from the later we included CPI and BCH codes-inspired approaches. With respect to the communication complexity, coding-inspired approaches dominate the competition, especially CPI with its nearly optimal communication for a known upper bound on the number of symmetric differences. When it comes to computational complexity, the IBLT-based approach has the best asymptotic performance. The message complexity of the compared protocols often depends on the method that is used to estimate the initial upper bound on the number of symmetric differences, except for the Cuckoo filter-based protocol that does not require such a bound. We identified and discussed the tradeoff between communication and message complexity when the upper bound on the number of mutual differences is not known in advance. Furthermore, we discussed the techniques for reducing computational complexity of the CPI approach at the price of increasing the message complexity. We concluded that there is no universally dominant protocol that beats the competition with respect to computational, communication and

message complexity at the same time.

***Network-wide synchronization.*** In Chapter 4, we tackled the problem of network wide synchronization through the example of synchronizing transaction pools in large-scale blockchains. As opposed to the recent improvements to popular public blockchains (*e.g.*, Graphene in Bitcoin and Bitcoin Cash) that integrate transaction pool synchronization into block propagation, we proposed maintaining the transaction pool synchronization process separately and independently from the block propagation channels. To that end, we proposed *SREP*, a novel transaction pool synchronization algorithm with quantifiable guarantees, which is based on communication-efficient end-to-end data synchronization protocols from Chapter 3.

We analyzed the performance of *SREP* in a comprehensive network model based on Watts-Strogatz random graphs. Besides the topological properties of the network, our model captures the statistics of transaction pools (*i.e.*, the sizes of transaction pools and the number of pair-wise mutual differences). Within our network model, we proved that *SREP* converges on any connected network in the number of iterations bounded by the network diameter. In the case of the “small-world” networks such as Ethereum, the number of *SREP* iterations to synchronize the entire network is logarithmic in the number of participants. On the other hand, the communication overhead is a function of network topology and transaction pool statistics.

Based on our network model, we developed *SREPSim*, a simulation technique that makes use of the transaction pool data from *in-situ* measurements, and can simulate transaction pool synchronization in large-scale blockchains on a single machine in tens of minutes. We ran a 3-day long campaign on the live Bitcoin network and used the collected data to parameterize *SREPSim*. We showed through simulations that *SREP* converges in only several iterations in “small-world” networks with 10,000 nodes and incurs significantly lower bandwidth overhead than *MempoolSync*.

***GenSync middleware.*** In Chapter 5, we described *GenSync* which, to the best of our knowledge, is the first middleware package that incorporates both approximate set membership data structure-based and coding theory-inspired solutions. It is implemented as a self contained C++ library and allows for a seamless integration into existing implementations through the minimal API. One of the unique features of *GenSync* is its integrated benchmarking layer that users can easily parameterize to mimic the performance of their target system with respect to network characteristics such as bandwidth, latency, and package loss, as well as the computational capabilities of the nodes involved in synchronization.

Using the *GenSync* benchmarking layer, we set up a series of experiments to characterize the performance of the state-of-the-art protocols in various practical system conditions. We found that IBLT-based protocols have the best real-time performance when the nodes have symmetrical compute power, the network link has sufficient bandwidth relative to the sizes of the sets, and the number of mutual differences is low (*e.g.*,  $\leq 10\%$  of the sets). Cuckoo-based approaches exhibit the best real-time performance in asymmetrical compute scenarios (*e.g.*, low-end mobile device synchronizing with a high-end server), as long as the network link has a sufficient bandwidth relative to the set sizes. When the link has a low latency but a low bandwidth relative to the sets sizes (*e.g.*, hundred of *Kb/s* bandwidth and tens of thousands of elements), and the number of mutual differences is small relative to the set sizes ( $\leq 10\%$ ), then CPI approaches dominate the competition. Notably, a bad protocol choice for the given system conditions can cause up to a five-fold hit in performance.

When it comes to incremental synchronization (*i.e.*, nodes sync periodically as new elements arrive on both sides), one is better off using interactive protocols than their non-interactive counterparts. In the case of the CPI protocol, we showed that the

interactive version of the protocol has several orders of magnitude better performance than its non-interactive counterpart.

Finally, we integrated *GenSync* with Colosseum, an SDR-based wireless network emulator to estimate the performance of the state-of-the-art protocols under different mobility patterns in cellular networks. To that end, we employed Colosseum’s “Boston” scenario in two modes: when the users are stationary with respect to their corresponding base stations, and when they move at pedestrian speed (5 m/s). We found that IBLT performs the best for general usage in both the stationary and pedestrian speed movement modes. However, for the applications that allocate only a fraction of bandwidth for data synchronization but still maintain the real-time synchronization performance on a reasonable level (*e.g.*, under 2 seconds), CPI outperforms IBLT up to nine times in terms of the consumed bandwidth.

## 6.2 Future Directions

***Improved end-to-end synchronization.*** As we demonstrated through experiments in Section 5.3.4, the relative communication inefficiency of IBLT-based synchronization protocols when compared to the CPI-based ones has significant ramifications in practice, especially in bandwidth-constrained systems. However, the recent IBLT alternative constructions such as the one of Lázaro and Matuz [65] allow for better communication performance in IBLT-based data synchronization protocols and introduce novel methods for estimating the number of mutual differences. As a future research, we propose implementing the IBLT construction of Lázaro and Matuz in *GenSync* and comparing its performance with other included protocols. Especially interesting are the bandwidth-constrained and high-latency networks. The former due to the promises of smaller IBLT sizes for the same number of encoded elements, and the latter due to the multi-message nature of the proposed differences

count estimation procedure.

Another promising future direction in this regard is implementing IBLT-based data synchronization with success guarantees. As we discussed in Section 3.3.4, current IBLT constructions may fail to decode (peel) with some arbitrarily small probability, even with a known number of mutual differences. A failure in the peeling process causes additional rounds of communication in the protocol. The recent IBLT construction of Mizrahi *et al.* [120] allows for constructing IBLTs with peeling success guarantees. First, we suggest an analytical comparison of this new construction with the existing ones, especially with respect to the communication and computational complexity. Second, we propose implementing the construction of Mizrahi *et al.* in *GenSync* framework and comparing the real-time performance of the resulting set reconciliation protocol with the other candidates from *GenSync*, especially CPI.

**Reactive GenSync.** In repeated end-to-end data synchronization (Section 5.4) in dynamic systems, we may expect significant variations in system parameters over time (*e.g.*, latency increase due to congestion). In this regards, we propose extending *GenSync* middleware with a reactive adaptation layer. In particular, we propose implementing a control module in between the *compatibility* and *implementations* layers (see Fig. 5-1) that switches the set reconciliation protocol contingent on the performance of the previous synchronization rounds. For instance, an increase in latency can be detected from the round trip times of initial messages (*i.e.*, parameters configuration). Using this insight combined with the knowledge of the relative practical performance among the protocols that we discussed in Chapter 5, the control module can replace the current protocol with a better one between rounds.

**SREP in dynamic networks.** In Chapter 4, we analyzed the performance of *SREP* in static networks, *i.e.*, the set of edges and nodes in the graph do not change over time. As future work, we propose analyzing *SREP*'s performance in

edge- and node-dynamic networks [196]. In particular, we propose analyzing *SREP*'s time to synchronize the network as a function of the network dynamics (*e.g.*, *t-interval connectivity* [164]).

## Appendix A

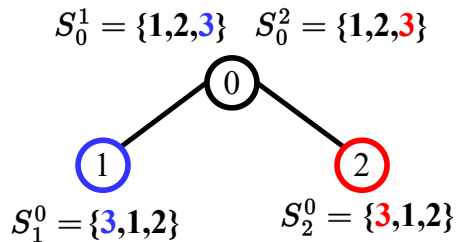
### *SREP*

#### A.1 Redundant Transmissions

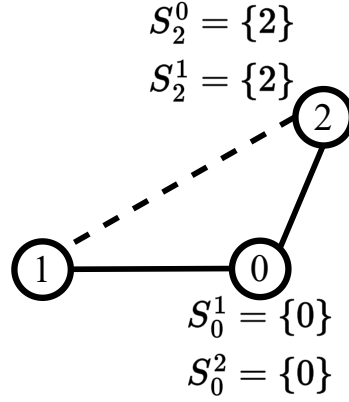
In elementary parallel *SREP*, an element may get transmitted to the node from multiple neighbors (via multiple replicas). One such example is depicted in Fig. A.1. Note that all but one such transmission is *redundant*. In other words, once the node receives the element through some replica, it will incorporate that element into its data set at the end of the round. In the subsequent round, all replicas will get recreated from the data set and will contain the transmitted element regardless of whether the replica itself received the element in the previous round or not.

#### A.2 On The Counting Argument From Theorem 2

To establish an upper bound, the counting argument from Theorem 2 counts all elements at all nodes as redundant transmissions (*i.e.*,  $|V|$  elements arriving via  $|V|-1$



**Figure A.1:** Element 3 gets transmitted from nodes 1 and 2, which makes one of these transmissions redundant.



**Figure A.2:** Three-node sub-network of every connected network with  $|V| \geq 3$  and the initial states of the local replicas at nodes 0 and 2.

replicas at  $|V|$  nodes). Here we present a simple argument to prove that such an estimate is a pessimistic one and that elementary parallel *SREP* produces strictly less than  $|V|^2 \cdot (|V| - 1)$  redundant transmissions.

For any connected network  $G = (V, E)$  with more than three nodes there is a sub-network of exactly three connected nodes with either two or three edges (see Fig. A.2). The element 1 will reach  $S_2^0$  at the beginning of either *second* round (when there are three edges) or the *third* round via node 0 (when there are only two edges). In any case, element 1 arrives at node 0 at the beginning of the *second* round via replica  $S_0^1$ . Starting with the *second* round, in all the subsequent rounds, element 1 will already be present in node 0's transaction pool, thus there will not be the transmission of element 1 to  $S_0^2$  from  $S_2^0$ , provided that there is no false positives of reconciliation in the chosen primal sync. On the other hand, for connected sub-networks of only two nodes, Lemma 1 applies. Therefore, there is strictly less than  $|V|^2 \cdot (|V| - 1)$  redundant transmissions in any connected network.

### A.3 The “Triangle Problem”

The triangle problem arises when we are given the mutual differences matrix  $M$  and need to construct a pools assignment  $A$  that satisfies it. In fact, for some  $M$ s there are no satisfying  $A$ s. Here we show one such example.

Suppose that we are given a network topology  $G = (V, E)$  where  $V = \{0, 1, 2\}$  and  $E = \{(0, 1), (0, 2), (1, 2)\}$  (a triangle), and a mutual differences matrix

$$M = \begin{bmatrix} 0 & 1 & 1 \\ & 0 & 1 \\ & & 0 \end{bmatrix}. \quad (\text{A.1})$$

Each  $m_{ij}$ , for  $i < j$  describes the number of mutual differences between  $S_i$  and  $S_j$ , and can be partitioned in  $\binom{m_{ij}+1}{m_{ij}}$  different ways (the number of weak 2-compositions of  $m_{ij}$ ). Thus, there are

$$\prod_{i < j} (m_{ij} + 1)$$

$D$  matrices corresponding to an  $M$  matrix. We say that a  $D$  matrix is *feasible* when there is a set assignment  $A$  that produces  $D$ .

For instance, the  $M$  from Eq. (A.1) has 8 corresponding  $D$ s and none are feasible:

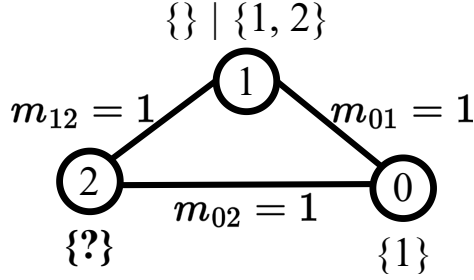
$$\begin{aligned}
D_0 &= \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, D_1 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \\
D_2 &= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, D_3 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \\
D_4 &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, D_5 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \\
D_6 &= \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, D_7 = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.
\end{aligned}$$

Without a loss of generality, we can assume the following construction of  $A$  for the given  $M$  to show that no  $D$ s are feasible.

Let  $S_0 = \{1\}$ , then there are two cases for  $S_1$  (see Fig. A.3). It must either be an empty set or must contain 1 and an additional element different than 1 (*e.g.*, 2). Otherwise, we would violate  $m_{01}$ . This yields the following cases for  $S_2$ :

- (1)  $S_2 = \{1\}$  and  $S_1 = \{\}$ , then  $m_{02} = 0$  which violates  $m_{02} = 1$ .
- (2)  $S_2 = \{2\}$  and  $S_1 = \{\}$ , then  $m_{02} = 2 \neq 1$ .
- (3)  $S_2 = \{3\}$  and  $S_1 = \{\}$ , for some  $3 \neq 2 \neq 1$ , then  $m_{02} = 2 \neq 1$ .
- (4)  $S_2 = \{1\}$  and  $S_1 = \{1, 2\}$ , then  $m_{02} = 0 \neq 1$ .
- (5)  $S_2 = \{2\}$  and  $S_1 = \{1, 2\}$ , then  $m_{02} = 2 \neq 1$ .

Therefore, we cannot simultaneously satisfy  $m_{01}$ ,  $m_{12}$ , and  $m_{02}$ .



**Figure A.3:** Constructing an  $A$  for the given  $M$  may fail.

In conclusion, constructing pools assignments  $A$  directly from a differences matrix  $M$  may not succeed for an arbitrary  $M$ . For that reason, we opt for Procedure 5 in our *SREP* network model.

#### A.4 On Rejecting $(M, s)$ Pairs

For *certain*  $(M, s)$  pairs, we can tell that  $M$  will not have any feasible  $D$ s without attempting to construct these  $D$ s.

**Lemma 6:** Given a network  $G = (V, E)$  and a pair  $(M, s)$ ,  $M$  has no feasible  $D$ s if there are  $(i, j) \in E$  such that:

$$\begin{aligned}
 m_{ij} &< |s_i - s_j|, \text{ or} \\
 m_{ij} &> s_i + s_j.
 \end{aligned}
 \tag{A.2}$$

This simply follows from general properties of sets. For any two sets  $S_i$  and  $S_j$ , the following holds:

$$\begin{aligned}
 |S_i \cup S_j| &\leq |S_i| + |S_j|, \text{ and} \\
 |S_i \setminus S_j| + |S_j \setminus S_i| &\geq ||S_i| - |S_j||.
 \end{aligned}$$

However, note that Lemma 6 does not imply anything about the  $M$ s for which the condition from Eq. (A.2) does *not* hold. That is, the following is a necessary but

not sufficient condition to have a feasible  $D$ :

$$\begin{aligned} m_{ij} &\geq |s_i - s_j|, \text{ and} \\ m_{ij} &\leq s_i + s_j. \end{aligned} \tag{A.3}$$

For instance, on a triangle graph, the following  $(M, s)$  pair satisfies Eq. (A.3) but  $M$  has no feasible  $D$ s:

$$M = \begin{bmatrix} 0 & 0 & 2 \\ & 0 & 1 \\ & & 0 \end{bmatrix} \text{ and } s = [2, 2, 1].$$

## A.5 On Constructing Pools Assignment from Set Sizes

Procedure 5 generates a pools assignment  $A$  given the sizes distribution  $\mathcal{S}$ , network topology  $G = (V, E)$ , and the parameter  $\psi$ . The main goal of the procedure is to produce an  $A$  such that its mutual differences matrix  $M$  is distributed similar to  $\mathcal{P}$ . As we discussed in Section 4.1, the performance of the primitive sync approaches that we use in *SREP* is a function of the number of mutual differences and not the sizes of the sets. Thus, our Procedure 5 prioritizes differences distribution  $\mathcal{P}$  over the sizes distribution  $\mathcal{S}$  in the following way.

Given the *target* sizes distribution  $\mathcal{S}$  and the appropriate value of  $\psi$ , Procedure 5 produces an  $A$  with the differences distribution similar to  $\mathcal{P}$ , but an *actual* sizes distribution  $\mathcal{S}^*$  that does not necessarily follow  $\mathcal{S}$ . This is because Procedure 5 incurs certain amount of collisions while sampling from  $\mathcal{U}$ . The smaller the parameter  $\psi$ , the larger the probability of collision on an  $e \in \{0..u\}$  (line 6 in Procedure 5). On the one hand, these collisions increase the similarity between sets in  $A$  by including the same element in multiple sets. On the other, sampling the same element multiple times for the same set makes the set smaller (as a set cannot contain duplicates by definition). The former class of collisions increases the expected value of mutual differences (mean

of  $\mathcal{P}$ ). The latter class of collisions decreases the expected value of set sizes (mean of  $\mathcal{S}$ ).

## Appendix B

# *GenSync* Abstractions Detailed

In Section 5.1, we describe the core abstractions of the *GenSync* framework and their function. In Fig. B-1, we include the methods that are omitted from the description in Section 5.1.

*GenSync* class contains the methods that the benchmarking layer uses to extract the fine-grained synchronization metrics (e.g., `getRcvBytes`). Since one *GenSync* object may include one or more *SyncMethods*, the metrics methods accept the index of the desired *SyncMethod* for which the fine-grained statistics should be returned. As we discussed in Chapter 3, the synchronization protocols often need to exchange synchronization parameters prior to starting the main part of the synchronization protocol. Therefore, *SyncMethod* contains `Send` and `RecvSyncParameters`. As *GenSync* is implemented in the client-server fashion, *Communicant* class contains `commListen()` that the server uses to initiate the service, `commClose()` that it uses to close the service, and `commConnect()` that the client uses to connect to the server.

Full documentation of the *GenSync* framework is available online at <https://github.com/nislab/gensync>, at the moment of writing.

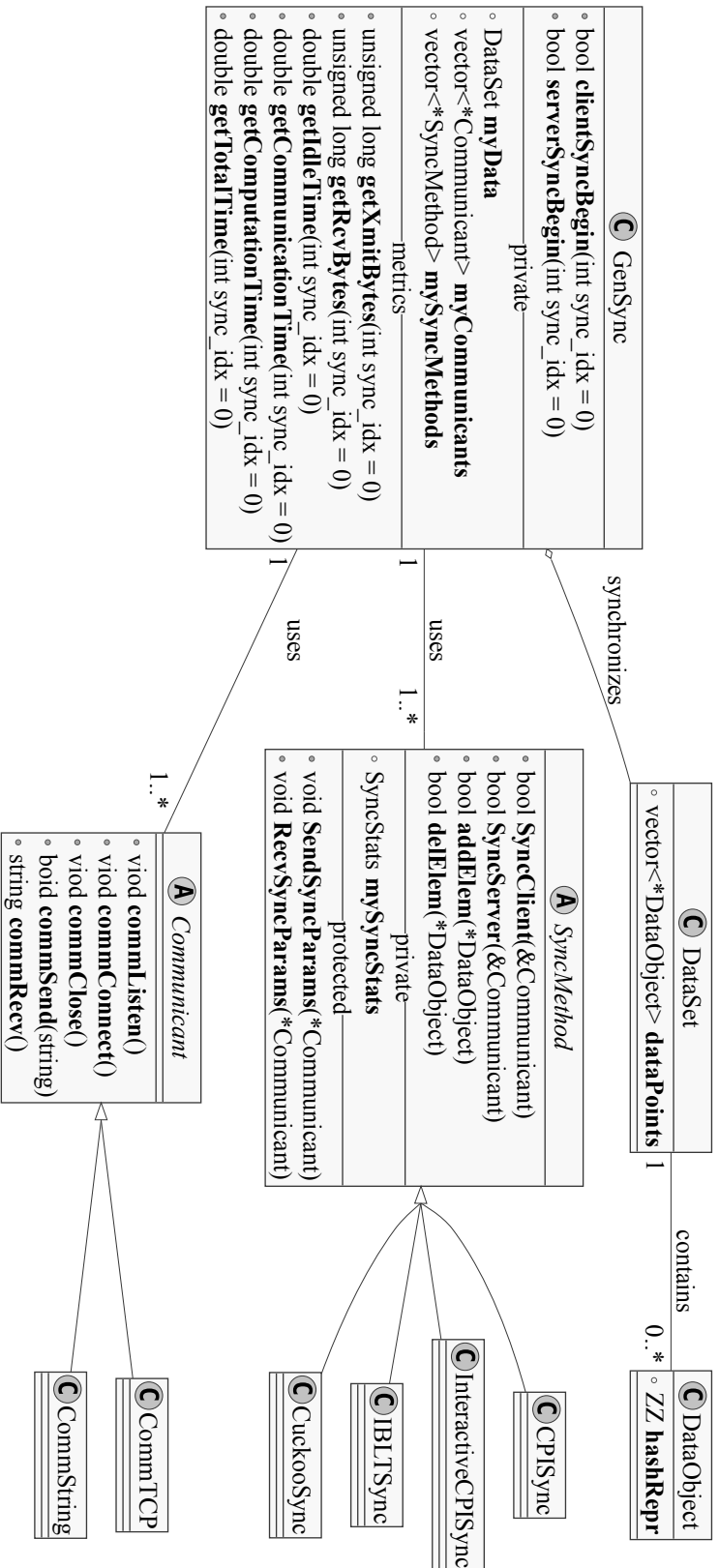


Figure B.1: Simplified UML diagram of the four core *GenSync* abstractions.

## Bibliography

- [1] A. A. Monrat, O. Schelén, and K. Andersson, “A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities,” *IEEE Access*, vol. 7, pp. 117 134–117 151, 2019. DOI: 10.1109/ACCESS.2019.2936094.
- [2] C. Chen, L. Zhang, Y. Li, T. Liao, S. Zhao, Z. Zheng, H. Huang, and J. Wu, “When Digital Economy Meets Web3.0: Applications and Challenges,” *IEEE Open Journal of the Computer Society*, vol. 3, pp. 233–245, 2022. DOI: 10.1109/OJCS.2022.3217565.
- [3] L. Gudgeon, D. Perez, D. Harz, B. Livshits, and A. Gervais, “The Decentralized Financial Crisis,” in *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2020, pp. 1–15. DOI: 10.1109/CVCBT50464.2020.00005.
- [4] W. Li, Z. Su, R. Li, K. Zhang, and Y. Wang, “Blockchain-Based Data Security for Artificial Intelligence Applications in 6G Networks,” *IEEE Network*, vol. 34, no. 6, pp. 31–37, 2020. DOI: 10.1109/MNET.021.1900629.
- [5] T. Hewa, G. Gür, A. Kalla, M. Ylianttila, A. Bracken, and M. Liyanage, “The Role of Blockchain in 6G: Challenges, Opportunities and Research Directions,” in *2020 2nd 6G Wireless Summit (6G SUMMIT)*, 2020, pp. 1–5. DOI: 10.1109/6GSUMMIT49458.2020.9083784.
- [6] A. Cannavò and F. Lamberti, “How Blockchain, Virtual Reality, and Augmented Reality are Converging, and Why,” *IEEE Consumer Electronics Magazine*, vol. 10, no. 5, pp. 6–13, 2021. DOI: 10.1109/MCE.2020.3025753.
- [7] L. Cao, “Decentralized AI: Edge Intelligence and Smart Blockchain, Metaverse, Web3, and DeSci,” *IEEE Intelligent Systems*, vol. 37, no. 3, pp. 6–19, 2022. DOI: 10.1109/MIS.2022.3181504.
- [8] T. M. Fernández-Caramés and P. Fraga-Lamas, “A Review on the Use of Blockchain for the Internet of Things,” *IEEE Access*, vol. 6, pp. 32 979–33 001, 2018. DOI: 10.1109/ACCESS.2018.2842685.
- [9] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen, “A Survey on Metaverse: Fundamentals, Security, and Privacy,” *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 319–352, 2023. DOI: 10.1109/COMST.2022.3202047.
- [10] Bank of Canada, *Central bank digital currency (CBDC)*, <https://www.bankofcanada.ca/research/digital-currencies-and-fintech/projects/central-bank-digital-currency/>, (Accessed 2023-02-02).

- [11] McKinsey & Company, *Web3 beyond the hype*, <https://www.mckinsey.com/industries/financial-services/our-insights/web3-beyond-the-hype>, (Accessed 2023-02-02).
- [12] P. Gigis, M. Calder, L. Manassakis, G. Nomikos, V. Kotronis, X. Dimitropoulos, E. Katz-Bassett, and G. Smaragdakis, "Seven years in the life of hypergiants' off-nets," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 516–533, ISBN: 9781450383837. DOI: 10.1145/3452296.3472928. [Online]. Available: <https://doi.org/10.1145/3452296.3472928>.
- [13] P. Danzi, A. E. Kalor, C. Stefanovic, and P. Popovski, "Analysis of the Communication Traffic for Blockchain Synchronization of IoT Devices," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8422485.
- [14] I. Makhdoom, I. Zhou, M. Abolhasan, J. Lipman, and W. Ni, "Privysharing: A blockchain-based framework for privacy-preserving and secure data sharing in smart cities," *Computers & Security*, vol. 88, p. 101653, 2020, ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2019.101653>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016740481930197X>.
- [15] Z. Lu, Q. Wang, G. Qu, and Z. Liu, "BARS: A Blockchain-Based Anonymous Reputation System for Trust Management in VANETs," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, pp. 98–103. DOI: 10.1109/TrustCom/BigDataSE.2018.00025.
- [16] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, "Solutions to Scalability of Blockchain: A Survey," *IEEE Access*, vol. 8, pp. 16440–16455, 2020. DOI: 10.1109/ACCESS.2020.2967218.
- [17] J. Xu, C. Wang, and X. Jia, "A survey of blockchain consensus protocols," *ACM Computing Surveys*, 2023, Just Accepted, ISSN: 0360-0300. DOI: 10.1145/3579845. [Online]. Available: <https://doi.org/10.1145/3579845>.
- [18] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 906–917, ISBN: 9781450316514. DOI: 10.1145/2382196.2382292. [Online]. Available: <https://doi.org/10.1145/2382196.2382292>.

- [19] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, “On the security and performance of proof of work blockchains,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 3–16, ISBN: 9781450341394. DOI: 10.1145/2976749.2978341. [Online]. Available: <https://doi.org/10.1145/2976749.2978341>.
- [20] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts (SoK),” in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186, ISBN: 978-3-662-54455-6.
- [21] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” *Communications of the ACM*, vol. 61, no. 7, pp. 95–102, 2018, ISSN: 0001-0782. DOI: 10.1145/3212998. [Online]. Available: <https://doi.org/10.1145/3212998>.
- [22] Matt Corallo, *Compact block relay bitcoin standard*, <https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki>, (Accessed 2022-12-02), 2016.
- [23] A. P. Ozisik, G. Andresen, B. N. Levine, D. Tapp, G. Bissias, and S. Katkuri, “Graphene: efficient interactive set reconciliation applied to blockchain propagation,” in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 303–317.
- [24] B. Kemme and G. Alonso, “Database replication: A tale of research across communities,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, pp. 5–12, 2010, ISSN: 2150-8097. DOI: 10.14778/1920841.1920847. [Online]. Available: <https://doi.org/10.14778/1920841.1920847>.
- [25] J. Sahoo, M. A. Salahuddin, R. Glitho, H. Elbiaze, and W. Ajib, “A Survey on Replica Server Placement Algorithms for Content Delivery Networks,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 1002–1026, 2017. DOI: 10.1109/COMST.2016.2626384.
- [26] L. Jin, S. Hao, H. Wang, and C. Cotton, “Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services,” in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 362–373. DOI: 10.1109/DSN.2018.00046.
- [27] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, “Renaissance: A Self-Stabilizing Distributed SDN Control Plane,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 233–243. DOI: 10.1109/ICDCS.2018.00032.

- [28] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, “Towards Network-Level Efficiency for Cloud Storage Services,” in *Proceedings of the Internet Measurement Conference (IMC)*, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 115–128, ISBN: 9781450332132. DOI: 10.1145/2663716.2663747.
- [29] H. Xiao, Z. Li, E. Zhai, T. Xu, Y. Li, Y. Liu, Q. Zhang, and Y. Liu, “Towards Web-based Delta Synchronization for Cloud Storage Services,” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA: USENIX Association, Feb. 2018, pp. 155–168, ISBN: 978-1-931971-42-3.
- [30] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, “QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services,” in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’15, Paris, France: Association for Computing Machinery, 2015, pp. 592–603, ISBN: 9781450336192. DOI: 10.1145/2789168.2790094. [Online]. Available: <https://doi.org/10.1145/2789168.2790094>.
- [31] MongoDB, *Atlas Device Sync*, <https://www.mongodb.com/atlas/app-services/device-sync>, (Accessed 2023-02-02).
- [32] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, “Local-first software: You own your data, in spite of the cloud,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178, ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. [Online]. Available: <https://doi.org/10.1145/3359591.3359737>.
- [33] T. Wang, J. Zhou, A. Liu, M. Z. A. Bhuiyan, G. Wang, and W. Jia, “Fog-based computing and storage offloading for data synchronization in IoT,” *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4272–4282, 2018.
- [34] T. R. Bennett, N. Gans, and R. Jafari, “Data-driven synchronization for internet-of-things systems,” *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 3, 2017, ISSN: 1539-9087. DOI: 10.1145/2983627. [Online]. Available: <https://doi.org/10.1145/2983627>.
- [35] T. Qu, S. P. Lei, Z. Z. Wang, D. X. Nie, X. Chen, and G. Q. Huang, “IoT-based real-time production logistics synchronization system under smart cloud manufacturing,” *The International Journal of Advanced Manufacturing Technology*, vol. 84, no. 1, pp. 147–164, 2016, ISSN: 1433-3015. DOI: 10.1007/s00170-015-7220-1. [Online]. Available: <https://doi.org/10.1007/s00170-015-7220-1>.

- [36] Z. Lin and L. Zhang, “Data synchronization algorithm for IoT gateway and platform,” in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, 2016, pp. 114–119. DOI: 10.1109/CompComm.2016.7924676.
- [37] S. Dang, O. Amin, B. Shihada, and M.-S. Alouini, “What should 6G be?” *Nature Electronics*, vol. 3, no. 1, pp. 20–29, 2020, ISSN: 2520-1131. DOI: 10.1038/s41928-019-0355-6. [Online]. Available: <https://doi.org/10.1038/s41928-019-0355-6>.
- [38] Y. Liu, X. Yuan, Z. Xiong, J. Kang, X. Wang, and D. Niyato, “Federated learning for 6G communications: Challenges, methods, and future directions,” *China Communications*, vol. 17, no. 9, pp. 105–118, 2020. DOI: 10.23919/JCC.2020.09.009.
- [39] X. Zhou, W. Liang, J. She, Z. Yan, and K. I.-K. Wang, “Two-Layer Federated Learning With Heterogeneous Model Aggregation for 6G Supported Internet of Vehicles,” *IEEE Transactions on Vehicular Technology*, vol. 70, no. 6, pp. 5308–5317, 2021. DOI: 10.1109/TVT.2021.3077893.
- [40] S. Dustdar, V. Casamajor Pujol, and P. K. Donta, “On distributed computing continuum systems,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2022. DOI: 10.1109/TKDE.2022.3142856.
- [41] D. Kimovski, R. Mathá, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, “Cloud, Fog, or Edge: Where to Compute?” *IEEE Internet Computing*, vol. 25, no. 4, pp. 30–36, 2021. DOI: 10.1109/MIC.2021.3050613.
- [42] N. Boškov, A. Trachtenberg, and D. Starobinski, “GenSync: A New Framework for Benchmarking and Optimizing Reconciliation of Data,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4408–4423, 2022. DOI: 10.1109/TNSM.2022.3164369.
- [43] M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, “Churn in the Bitcoin Network,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1598–1615, 2021. DOI: 10.1109/TNSM.2021.3050428.
- [44] Y. Gao, J. Shi, X. Wang, Q. Tan, C. Zhao, and Z. Yin, “Topology Measurement and Analysis on Ethereum P2P Network,” in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–7. DOI: 10.1109/ISCC47284.2019.8969695.
- [45] T. Wang, C. Zhao, Q. Yang, S. Zhang, and S. C. Liew, “Ethna: Analyzing the Underlying Peer-to-Peer Network of Ethereum Blockchain,” *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 3, pp. 2131–2146, 2021. DOI: 10.1109/TNSE.2021.3078181.

- [46] N. Boškov, S. Simsek, A. Trachtenberg, and D. Starobinski, “SREP: Out-Of-Band Sync of Transaction Pools for Large-Scale Blockchains,” in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023 accepted. [Online]. Available: <https://arxiv.org/abs/2303.16809>.
- [47] N. Boškov, A. Trachtenberg, and D. Starobinski, “Enabling Cost-Benefit Analysis of Data Sync Protocols,” *Computer*, 2023 accepted. DOI: 10.1109/MC.2023.3251195. [Online]. Available: <https://arxiv.org/abs/2303.17530>.
- [48] L. Samuels, N. Boskov, A. F. Oliveira, E. Sun, D. Starobinski, A. Trachtenberg, M. Varia, M. Monga, R. Canetti, A. Devaiah, *et al.*, “Automated Exposure Notification for COVID-19,” *Journal of Young Investigators*, 2022. [Online]. Available: <https://www.jyi.org/2022-december/2022/12/19/automated-exposure-notification-for-covid-19>.
- [49] N. Boskov, N. Radami, T. Tiwari, and A. Trachtenberg, “Union buster: A cross-container covert-channel exploiting union mounting,” in *Cyber Security, Cryptology, and Machine Learning*, S. Dolev, J. Katz, and A. Meisels, Eds., Cham: Springer International Publishing, 2022, pp. 300–317, ISBN: 978-3-031-07689-3.
- [50] N. Boskov, A. Trachtenberg, and D. Starobinski, “Birdwatching: False Negatives In Cuckoo Filters,” in *Proceedings of the Student Workshop*, ser. CoNEXT’20, Barcelona, Spain: Association for Computing Machinery, 2020, pp. 13–14, ISBN: 9781450381833. DOI: 10.1145/3426746.3434063. [Online]. Available: <https://doi.org/10.1145/3426746.3434063>.
- [51] P. Yellu, N. Boskov, M. A. Kinsky, and Q. Yu, “Security Threats in Approximate Computing Systems,” in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’19, Tysons Corner, VA, USA: Association for Computing Machinery, 2019, pp. 387–392, ISBN: 9781450362528. DOI: 10.1145/3299874.3319453. [Online]. Available: <https://doi.org/10.1145/3299874.3319453>.
- [52] M. A. Kinsky and N. Boskov, “Secure Computing Systems Design Through Formal Micro-Contracts,” in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’19, Tysons Corner, VA, USA: Association for Computing Machinery, 2019, pp. 537–542, ISBN: 9781450362528. DOI: 10.1145/3299874.3319447. [Online]. Available: <https://doi.org/10.1145/3299874.3319447>.
- [53] N. Boskov, M. Isakov, and M. A. Kinsky, “CodeTrolley: Hardware-Assisted Control Flow Obfuscation,” *Boston Area Architecture Workshop*, vol. abs/1903.00841, 2019. arXiv: 1903.00841. [Online]. Available: <http://arxiv.org/abs/1903.00841>.

- [54] Y. Minsky, A. Trachtenberg, and R. Zippel, “Set reconciliation with nearly optimal communication complexity,” *Proceedings. 2001 IEEE International Symposium on Information Theory (IEEE Cat. No.01CH37252)*, pp. 232–, 2001.
- [55] A. Tridgell and P. Mackerras, “The rsync algorithm,” *Joint Computer Science Technical Report Series, TR-CS-96-05, The Australian National University*, 1996. [Online]. Available: <https://www.andrew.cmu.edu/course/15-749/READINGS/required/cas/tridgell196.pdf>.
- [56] W. Xia, C. Wei, Z. Li, X. Wang, and X. Zou, “NetSync: A Network Adaptive and Deduplication-Inspired Delta Synchronization Approach for Cloud Storage Services,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2554–2570, 2022. DOI: 10.1109/TPDS.2022.3145025.
- [57] B. Song and A. Trachtenberg, “Scalable String Reconciliation by Recursive Content-Dependent Shingling,” in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2019, pp. 623–630. DOI: 10.1109/ALLERTON.2019.8919901.
- [58] M. Skjogstad and T. Maseng, “Low complexity set reconciliation using bloom filters,” in *Proceedings of the 7th ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, ser. FOMC ’11, doi: <https://doi.org/10.1145/1998476.1998483>, San Jose, California: Association for Computing Machinery, 2011, pp. 33–41, ISBN: 9781450307796. DOI: 10.1145/1998476.1998483.
- [59] X. Tian, D. Zhang, K. Xie, C. Hu, M. Wang, and J. Deng, “Exact set reconciliation based on bloom filters,” in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, IEEE, vol. 3, 2011, pp. 2001–2009.
- [60] D. Guo and M. Li, “Set reconciliation via counting bloom filters,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 10, pp. 2367–2380, 2012.
- [61] L. Luo, D. Guo, O. Rottenstreich, R. T. Ma, and X. Luo, “Set Reconciliation with Cuckoo Filters,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2465–2468.
- [62] S. Li, L. Luo, D. Guo, and Y. Zhao, “Multiset synchronization with counting cuckoo filters,” in *Wireless Algorithms, Systems, and Applications*, D. Yu, F. Dressler, and J. Yu, Eds., Cham: Springer International Publishing, 2020, pp. 231–243, ISBN: 978-3-030-59016-1.
- [63] L. Luo, D. Guo, Y. Zhao, O. Rottenstreich, R. T. Ma, and X. Luo, “MCF-syn: A Multi-Party Set Reconciliation Protocol With the Marked Cuckoo Filter,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2705–2718, 2021.

- [64] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, “What’s the difference? Efficient set reconciliation without prior context,” *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 218–229, 2011.
- [65] F. Lázaro and B. Matuz, *A Rate-Compatible Solution to the Set Reconciliation Problem*, 2022. DOI: 10.48550/ARXIV.2211.05472. [Online]. Available: <https://arxiv.org/abs/2211.05472>.
- [66] S. Lee, H. Byun, and H. Lim, “Set Reconciliation Using Ternary and Invertible Bloom Filters,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–14, 2023. DOI: 10.1109/TKDE.2023.3237009.
- [67] X. Ding, L. Zhao, L. Luo, J. Xie, D. Guo, and J. Li, “Gauze: Enabling Communication-Friendly Block Synchronization with Cuckoo Filter,” *Frontiers of Computer Science*, vol. 17, no. 3, p. 173403, 2022, ISSN: 2095-2236. DOI: 10.1007/s11704-022-1685-5. [Online]. Available: <https://doi.org/10.1007/s11704-022-1685-5>.
- [68] Y. Minsky and A. Trachtenberg, “Practical set reconciliation,” in *40th Annual Allerton Conference on Communication, Control, and Computing*, Citeseer, vol. 248, 2002.
- [69] Y. Dodis, L. Reyzin, and A. Smith, “Fuzzy extractors: How to generate strong keys from biometrics and other noisy data,” in *International conference on the theory and applications of cryptographic techniques*, Springer, 2004, pp. 523–540.
- [70] L. Gong, Z. Liu, L. Liu, J. Xu, M. Ogihara, and T. Yang, “Space- and Computationally-Efficient Set Reconciliation via Parity Bitmap Sketch (PBS),” *arXiv preprint arXiv:2007.14569*, 2020.
- [71] J. Jin, W. Si, D. Starobinski, and A. Trachtenberg, “Prioritized data synchronization for disruption tolerant networks,” in *MILCOM 2012-2012 IEEE Military Communications Conference*, IEEE, 2012, pp. 1–8.
- [72] G. Naumenko, G. Maxwell, P. Wuille, A. Fedorova, and I. Beschastnikh, “Erlay: Efficient transaction relay for bitcoin,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 817–831.
- [73] A. D. Breslow and N. S. Jayasena, “Morton filters: fast, compressed sparse cuckoo filters,” *The VLDB Journal*, vol. 29, no. 2, pp. 731–754, 2020, ISSN: 0949-877X. DOI: 10.1007/s00778-019-00561-0. [Online]. Available: <https://doi.org/10.1007/s00778-019-00561-0>.
- [74] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970, ISSN: 0001-0782. DOI: 10.1145/362686.362692. [Online]. Available: <https://doi.org/10.1145/362686.362692>.

- [75] D. Eppstein and M. T. Goodrich, “Space-Efficient Straggler Identification in Round-Trip Data Streams Via Newton’s Identities and Invertible Bloom Filters,” in *Algorithms and Data Structures*, F. Dehne, J.-R. Sack, and N. Zeh, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 637–648, ISBN: 978-3-540-73951-7.
- [76] M. T. Goodrich and M. Mitzenmacher, “Invertible bloom lookup tables,” in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2011, pp. 792–799. DOI: 10.1109/Allerton.2011.6120248.
- [77] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.
- [78] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” May 2009. [Online]. Available: <http://www.bitcoin.org/bitcoin.pdf>.
- [79] R. C. Merkle, “A Digital Signature Based on a Conventional Encryption Function,” in *Advances in Cryptology — CRYPTO ’87*, C. Pomerance, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378, ISBN: 978-3-540-48184-3.
- [80] Y. Han, C. Li, P. Li, M. Wu, D. Zhou, and F. Long, “Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 238–252, ISBN: 9781450381376. DOI: 10.1145/3419111.3421283. [Online]. Available: <https://doi.org/10.1145/3419111.3421283>.
- [81] Blockchain.com, *Average Block Size in Bitcoin*, <https://www.blockchain.com/explorer/charts/avg-block-size>, (Accessed 2023-02-02).
- [82] Y. Gao, J. Shi, X. Wang, Q. Tan, C. Zhao, and Z. Yin, “Topology Measurement and Analysis on Ethereum P2P Network,” in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–7. DOI: 10.1109/ISCC47284.2019.8969695.
- [83] T. Wang, C. Zhao, Q. Yang, S. Zhang, and S. C. Liew, “Ethna: Analyzing the Underlying Peer-to-Peer Network of Ethereum Blockchain,” *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 3, pp. 2131–2146, 2021. DOI: 10.1109/TNSE.2021.3078181.
- [84] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks,” *Nature*, vol. 393, no. 6684, pp. 440–442, 1998, ISSN: 1476-4687. DOI: 10.1038/30918. [Online]. Available: <https://doi.org/10.1038/30918>.

- [85] Y. Minsky, A. Trachtenberg, and R. Zippel, “Set Reconciliation with Nearly Optimal Communication Complexity,” *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, 2003.
- [86] A. Boral and M. Mitzenmacher, “Multi-party set reconciliation using characteristic polynomials,” in *2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, IEEE, 2014, pp. 1182–1187.
- [87] M. T. Goodrich and M. Mitzenmacher, “Invertible bloom lookup tables,” in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, IEEE, 2011, pp. 792–799.
- [88] M. Mitzenmacher and T. Morgan, “Reconciling graphs and sets of sets,” in *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2018, pp. 33–47.
- [89] M. Mitzenmacher and R. Pagh, “Simple multi-party set reconciliation,” *Distributed Computing*, vol. 31, no. 6, pp. 441–453, 2018, ISSN: 1432-0452. DOI: 10.1007/s00446-017-0316-0. [Online]. Available: <https://doi.org/10.1007/s00446-017-0316-0>.
- [90] S. Agarwal, V. Chauhan, and A. Trachtenberg, “Bandwidth Efficient String Reconciliation Using Puzzles,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1217–1225, 2006. DOI: 10.1109/TPDS.2006.148.
- [91] M. Mitzenmacher and T. Morgan, “Reconciling Graphs and Sets of Sets,” *CoRR*, vol. abs/1707.05867, 2017. arXiv: 1707.05867. [Online]. Available: <http://arxiv.org/abs/1707.05867>.
- [92] N. Kruber, “Approximate distributed set reconciliation with defined accuracy,” PhD thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, 2020. DOI: <http://dx.doi.org/10.18452/21294>.
- [93] M. Mitzenmacher and T. Morgan, “Robust set reconciliation via locality sensitive hashing,” in *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS ’19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 164–181, ISBN: 9781450362276. DOI: 10.1145/3294052.3319690. [Online]. Available: <https://doi.org/10.1145/3294052.3319690>.
- [94] A. Balasubramanian, B. Levine, and A. Venkataramani, “DTN Routing as a Resource Allocation Problem,” *SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 373–384, 2007, ISSN: 0146-4833. DOI: 10.1145/1282427.1282422. [Online]. Available: <https://doi.org/10.1145/1282427.1282422>.

- [95] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.
- [96] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, *et al.*, “Windows azure storage: a highly available cloud storage service with strong consistency,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 143–157.
- [97] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and Logic*, 5th ed. Cambridge University Press, 2007. DOI: 10.1017/CB09780511804076.
- [98] R. Bose and D. Ray-Chaudhuri, “On a class of error correcting binary group codes,” *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960, ISSN: 0019-9958. DOI: [https://doi.org/10.1016/S0019-9958\(60\)90287-4](https://doi.org/10.1016/S0019-9958(60)90287-4). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0019995860902874>.
- [99] E. Gorog, “Some New Classes of Cyclic Codes Used for Burst-Error Correction,” *IBM Journal of Research and Development*, vol. 7, no. 2, pp. 102–111, 1963. DOI: 10.1147/rd.72.0102.
- [100] W. W. Peterson, W. Peterson, E. J. Weldon, and E. J. Weldon, *Error-correcting codes*. MIT press, 1972.
- [101] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, “An approximate L/sup 1/-difference algorithm for massive data streams,” in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999, pp. 501–511. DOI: 10.1109/SFFCS.1999.814623.
- [102] G. Cormode and S. Muthukrishnan, “What’s new: finding significant differences in network data streams,” *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219–1232, 2005. DOI: 10.1109/TNET.2005.860096.
- [103] G. Cormode, S. Muthukrishnan, and I. Rozenbaum, “Summarizing and mining inverse distributions on data streams via dynamic inverse sampling,” in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB ’05, Trondheim, Norway: VLDB Endowment, 2005, pp. 25–36, ISBN: 1595931546.
- [104] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik, “Reversible Sketches: Enabling Monitoring and Analysis Over High-Speed Data Streams,” *IEEE/ACM Transactions on Networking*, vol. 15, no. 5, pp. 1059–1072, 2007. DOI: 10.1109/TNET.2007.896150.

- [105] P. Indyk and R. Motwani, “Approximate nearest neighbors: Towards removing the curse of dimensionality,” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’98, Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 604–613, ISBN: 0897919629. DOI: 10.1145/276698.276876. [Online]. Available: <https://doi.org/10.1145/276698.276876>.
- [106] A. Broder, “On the resemblance and containment of documents,” in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.
- [107] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, “Min-wise independent permutations,” *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000, ISSN: 0022-0000. DOI: <https://doi.org/10.1006/jcss.1999.1690>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0022000099916902>.
- [108] P. Flajolet and G. N. Martin, “Probabilistic counting algorithms for data base applications,” *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.
- [109] H. Lim, J. Lee, H. Byun, and C. Yim, “Ternary Bloom Filter Replacing Counting Bloom Filter,” *IEEE Communications Letters*, vol. 21, no. 2, pp. 278–281, 2017. DOI: 10.1109/LCOMM.2016.2624286.
- [110] F. Lázaro and B. Matuz, “Irregular invertible bloom look-up tables,” *CoRR*, vol. abs/2107.02573, 2021. arXiv: 2107.02573. [Online]. Available: <https://arxiv.org/abs/2107.02573>.
- [111] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, “A digital fountain approach to reliable distribution of bulk data,” in *Proceedings of the ACM SIGCOMM ’98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’98, Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 56–67, ISBN: 1581130031. DOI: 10.1145/285237.285258. [Online]. Available: <https://doi.org/10.1145/285237.285258>.
- [112] F. Cohen, “A cryptographic checksum for integrity protection,” *Computers & Security*, vol. 6, no. 6, pp. 505–510, 1987, ISSN: 0167-4048. DOI: [https://doi.org/10.1016/0167-4048\(87\)90031-9](https://doi.org/10.1016/0167-4048(87)90031-9). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0167404887900319>.
- [113] C. Partridge, J. Hughes, and J. Stone, “Performance of checksums and crcs over real data,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’95, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1995, pp. 68–76, ISBN: 0897917111. DOI: 10.1145/217382.217413. [Online]. Available: <https://doi.org/10.1145/217382.217413>.

- [114] D. P. Dubhashi and A. Panconesi, *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.
- [115] D. Eppstein and M. T. Goodrich, “Straggler identification in round-trip data streams via Newton’s identities and invertible Bloom filters,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 2, pp. 297–306, 2010.
- [116] M. Molloy, “The pure literal rule threshold and cores in random hypergraphs,” in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’04, New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2004, pp. 672–681, ISBN: 089871558X.
- [117] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, “Tight thresholds for cuckoo hashing via xorsat,” in *Automata, Languages and Programming*, S. Abramsky, C. Gavaille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 213–225, ISBN: 978-3-642-14165-2.
- [118] B. N. Levine, *IBLT-optimization*, <https://github.com/umass-forensics/IBLT-optimization>, (Accessed 2023-02-02).
- [119] G. Andresen, *IBLT\_Cplusplus*, [https://github.com/gavinandresen/IBLT\\_Cplusplus](https://github.com/gavinandresen/IBLT_Cplusplus), (Accessed 2023-02-02).
- [120] A. Mizrahi, D. Bar-Lev, E. Yaakobi, and O. Rottenstreich, *Invertible bloom lookup tables with listing guarantees*, 2022. DOI: 10.48550/ARXIV.2212.13812. [Online]. Available: <https://arxiv.org/abs/2212.13812>.
- [121] S. Z. Kiss, . Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, “Bloom Filter With a False Positive Free Zone,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 2334–2349, 2021. DOI: 10.1109/TNSM.2021.3059075.
- [122] I. Kubjas and V. Skachek, “Partial extraction from invertible bloom filters,” in *IEEE International Symposium on Information Theory, ISIT 2022, Espoo, Finland, June 26 - July 1, 2022*, IEEE, 2022, pp. 2415–2420. DOI: 10.1109/ISIT50566.2022.9834808. [Online]. Available: <https://doi.org/10.1109/ISIT50566.2022.9834808>.
- [123] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981. DOI: 10.1109/TIT.1981.1056404.
- [124] S. A. Shaikh, H. Chivers, P. Nobles, J. A. Clark, and H. Chen, “Network reconnaissance,” *Network Security*, vol. 2008, no. 11, pp. 12–16, 2008, ISSN: 1353-4858. DOI: [https://doi.org/10.1016/S1353-4858\(08\)70129-6](https://doi.org/10.1016/S1353-4858(08)70129-6). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1353485808701296>.

- [125] R. White, G. Caiazza, C. Jiang, X. Ou, Z. Yang, A. Cortesi, and H. Christensen, "Network Reconnaissance and Vulnerability Excavation of Secure DDS Systems," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019, pp. 57–66. DOI: 10.1109/EuroSPW.2019.00013.
- [126] S. Gvozdenovic, J. K. Becker, J. Mikulskis, and D. Starobinski, "Multi-Protocol IoT Network Reconnaissance," in *2022 IEEE Conference on Communications and Network Security (CNS)*, 2022, pp. 118–126. DOI: 10.1109/CNS56114.2022.9947261.
- [127] E. C. at Carnegie Mellon, *Cuckoo filter*, <https://github.com/efficient/cuckoofilter>, (Accessed 2023-02-02), 2019.
- [128] I. Sharif, *Cuckoo filter in golang*, <https://github.com/irfansharif/cfilter>, (Accessed 2023-02-02), 2017.
- [129] S. Lotfy, *Cuckoo filter in golang*, <https://github.com/seiflotfy/cuckoofilter>, (Accessed 2023-02-02), 2020.
- [130] J. Cui, J. Zhang, H. Zhong, and Y. Xu, "SPACF: A Secure Privacy-Preserving Authentication Scheme for VANET With Cuckoo Filter," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 11, pp. 10 283–10 295, 2017. DOI: 10.1109/TVT.2017.2718101.
- [131] L. Devroye and P. Morin, "Cuckoo hashing: Further analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215–219, 2003, ISSN: 0020-0190. DOI: [https://doi.org/10.1016/S0020-0190\(02\)00500-8](https://doi.org/10.1016/S0020-0190(02)00500-8). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0020019002005008>.
- [132] M. Drmota and R. Kutzelnigg, "A precise analysis of cuckoo hashing," *ACM Transactions on Algorithms*, vol. 8, no. 2, 2012, ISSN: 1549-6325. DOI: 10.1145/2151171.2151174. [Online]. Available: <https://doi.org/10.1145/2151171.2151174>.
- [133] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, I. Dinur, K. Jansen, J. Naor, and J. Rolim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 490–503, ISBN: 978-3-642-03685-9.
- [134] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 197–210, 2019, ISSN: 2150-8097. DOI: 10.14778/3364324.3364333. [Online]. Available: <https://doi.org/10.14778/3364324.3364333>.

- [135] L. Luo, D. Guo, O. Rottenstreich, R. T. Ma, X. Luo, and B. Ren, “The Consistent Cuckoo Filter,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 712–720. DOI: 10.1109/INFOCOM.2019.8737454.
- [136] F. Wang, H. Chen, L. Liao, F. Zhang, and H. Jin, “The Power of Better Choice: Reducing Relocations in Cuckoo Filter,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 358–367. DOI: 10.1109/ICDCS.2019.00043.
- [137] H. Lang, T. Neumann, A. Kemper, and P. Boncz, “Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput,” *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 502–515, 2019, ISSN: 2150-8097. DOI: 10.14778/3303753.3303757. [Online]. Available: <https://doi.org/10.14778/3303753.3303757>.
- [138] T. M. Graf and D. Lemire, “Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters,” *ACM Journal of Experimental Algorithmics*, vol. 25, 2020, ISSN: 1084-6654. DOI: 10.1145/3376122. [Online]. Available: <https://doi.org/10.1145/3376122>.
- [139] K. Huang and T. Yang, “Additive and Subtractive Cuckoo Filters,” in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, 2020, pp. 1–10. DOI: 10.1109/IWQoS49365.2020.9213014.
- [140] N. Dayan and M. Twitto, “Chucky: A succinct cuckoo filter for lsm-tree,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21, Virtual Event, China: Association for Computing Machinery, 2021, pp. 365–378, ISBN: 9781450383431. DOI: 10.1145/3448016.3457273. [Online]. Available: <https://doi.org/10.1145/3448016.3457273>.
- [141] P. Fu, L. Luo, S. Li, D. Guo, G. Cheng, and Y. Zhou, “The Vertical Cuckoo Filters: A Family of Insertion-friendly Sketches for Online Applications,” in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 57–67. DOI: 10.1109/ICDCS51616.2021.00015.
- [142] K. Huang and T. Yang, “Tagged Cuckoo Filters,” in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021, pp. 1–10. DOI: 10.1109/ICCCN52240.2021.9522301.
- [143] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson, “Vector quotient filters: Overcoming the time/space trade-off in filter design,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21, Virtual Event, China: Association for Computing Machinery, 2021, pp. 1386–1399, ISBN: 9781450383431. DOI: 10.1145/3448016.3452841. [Online]. Available: <https://doi.org/10.1145/3448016.3452841>.

- [144] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, “A reliable randomized algorithm for the closest-pair problem,” *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997, ISSN: 0196-6774. DOI: 10.1006/jagm.1997.0873. [Online]. Available: <http://dx.doi.org/10.1006/jagm.1997.0873>.
- [145] B. Developers, *Bitcoin Bloom Filter Implementation*, <https://github.com/bitcoin/bitcoin/blob/be2e748f378fc9ed40593a723dd18f2528705956/src/common/bloom.h#L1>, Accessed: 2/2/2023.
- [146] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí, “Analysis of the Bitcoin UTXO Set,” in *Financial Cryptography and Data Security*, A. Zohar, I. Eyal, V. Teague, J. Clark, A. Bracciali, F. Pintore, and M. Sala, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 78–91, ISBN: 978-3-662-58820-8.
- [147] L. Kiffer, A. Salman, D. Levin, A. Mislove, and C. Nita-Rotaru, “Under the Hood of the Ethereum Gossip Protocol,” in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 437–456, ISBN: 978-3-662-64330-3. DOI: 10.1007/978-3-662-64331-0\_23. [Online]. Available: [https://doi.org/10.1007/978-3-662-64331-0\\_23](https://doi.org/10.1007/978-3-662-64331-0_23).
- [148] X. Ma, H. Wu, D. Xu, and K. Wolter, “CBlockSim: A Modular High-Performance Blockchain Simulator,” in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2022, pp. 1–5. DOI: 10.1109/ICBC54727.2022.9805504.
- [149] Bitcoin developers, *Bitcoin referential implementation*, <https://github.com/bitcoin/bitcoin>, (Accessed 2022-12-02), 2022.
- [150] S. Delgado-Segura, S. Bakshi, C. Pérez-Solà, J. Litton, A. Pachulski, A. Miller, and B. Bhattacharjee, “TxProbe: Discovering Bitcoin’s Network Topology Using Orphan Transactions,” in *Financial Cryptography and Data Security*, I. Goldberg and T. Moore, Eds., Cham: Springer International Publishing, 2019, pp. 550–566, ISBN: 978-3-030-32101-7.
- [151] M. Grundmann, M. Baumstark, and H. Hartenstein, “On the Peer Degree Distribution of the Bitcoin P2P Network,” in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2022, pp. 1–5. DOI: 10.1109/ICBC54727.2022.9805511.
- [152] Y. Shahsavari, K. Zhang, and C. Talhi, “A Theoretical Model for Block Propagation Analysis in Bitcoin Network,” *IEEE Transactions on Engineering Management*, vol. 69, no. 4, pp. 1459–1476, 2022. DOI: 10.1109/TEM.2020.2989170.

- [153] P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65, ISBN: 978-3-540-45748-0.
- [154] F. Chung and L. Lu, “The diameter of sparse random graphs,” *Advances in Applied Mathematics*, vol. 26, no. 4, pp. 257–279, 2001.
- [155] N. Boškov, *SREPSim*, <http://www.github.com/nislab/SREPSim>, (Accessed 2023-02-02).
- [156] C. Faria and M. Correia, “BlockSim: Blockchain Simulator,” in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 439–446. DOI: 10.1109/Blockchain.2019.00067.
- [157] M. Alharby and A. van Moorsel, “BlockSim: An Extensible Simulation Tool for Blockchain Systems,” *Frontiers in Blockchain*, vol. 3, 2020, ISSN: 2624-7852. DOI: 10.3389/fbloc.2020.00028. [Online]. Available: <https://www.frontiersin.org/articles/10.3389/fbloc.2020.00028>.
- [158] R. Banno and K. Shudo, “Simulating a Blockchain Network with SimBlock,” in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 3–4. DOI: 10.1109/BL0C.2019.8751431.
- [159] M. A. Imtiaz, D. Starobinski, and A. Trachtenberg, “Characterizing Orphan Transactions in the Bitcoin Network,” in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2020, pp. 1–9. DOI: 10.1109/ICBC48266.2020.9169449.
- [160] —, “Investigating Orphan Transactions in the Bitcoin Network,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1718–1731, 2021. DOI: 10.1109/TNSM.2021.3056949.
- [161] —, “Empirical Comparison of Block Relay Protocols,” *IEEE Transactions on Network and Service Management*, pp. 1–1, 2022. DOI: 10.1109/TNSM.2022.3195976.
- [162] Bitcoin developers, *Ancestor Score Sorting*, <https://github.com/bitcoin/bitcoin/blob/master/src/txmempool.h>, (Accessed 2022-12-02), 2022.
- [163] D. Topkis, “Concurrent Broadcast for Information Dissemination,” *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1107–1112, 1985. DOI: 10.1109/TSE.1985.231858.
- [164] F. Kuhn, N. Lynch, and R. Oshman, “Distributed Computation in Dynamic Networks,” in *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, ser. STOC ’10, Cambridge, Massachusetts, USA: Association for Computing Machinery, 2010, pp. 513–522, ISBN: 9781450300506. DOI: 10.1145/1806689.1806760. [Online]. Available: <https://doi.org/10.1145/1806689.1806760>.

- [165] B. Haeupler and D. Karger, “Faster information dissemination in dynamic networks via network coding,” in *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC ’11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 381–390, ISBN: 9781450307192. DOI: 10.1145/1993806.1993885. [Online]. Available: <https://doi.org/10.1145/1993806.1993885>.
- [166] C. Dutta, G. Pandurangan, R. Rajaraman, Z. Sun, and E. Viola, “On the Complexity of Information Spreading in Dynamic Networks,” in *Proceedings of the 2013 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 717–736. DOI: 10.1137/1.9781611973105.52. eprint: <https://epubs.siam.org/doi/pdf/10.1137/1.9781611973105.52>. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1.9781611973105.52>.
- [167] N. Boskov, A. Trachtenberg, D. Starobinski, and contributors, *GenSync Framework*, <http://www.github.com/nislab/gensync>, (Accessed 2023-02-02).
- [168] P. K. Donta and S. Dustdar, “The Promising Role of Representation Learning for Distributed Computing Continuum Systems,” in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 126–132. DOI: 10.1109/SOSE55356.2022.00021.
- [169] A. Pagh and R. Pagh, “Uniform Hashing in Constant Time and Optimal Space,” *SIAM Journal on Computing*, vol. 38, no. 1, pp. 85–96, 2008. DOI: 10.1137/060658400. eprint: <https://doi.org/10.1137/060658400>. [Online]. Available: <https://doi.org/10.1137/060658400>.
- [170] Victor Shoup and others, *NTL: A Library for doing Number Theory*, <https://libnt1.org/>, (Accessed 2023-02-02).
- [171] R. Stewart and C. Metz, “SCTP: new transport protocol for TCP/IP,” *IEEE Internet Computing*, vol. 5, no. 6, pp. 64–69, 2001, doi: <https://doi.org/10.1109/4236.968833>. DOI: 10.1109/4236.968833.
- [172] L. Bonati, P. Johari, M. Polese, S. D’Oro, S. Mohanti, M. Tehrani-Moayyed, D. Villa, S. Shrivastava, C. Tassie, K. Yoder, A. Bagga, P. Patel, V. Petkov, M. Seltser, F. Restuccia, A. Gosain, K. R. Chowdhury, S. Basagni, and T. Melodia, “Colosseum: Large-Scale Wireless Experimentation Through Hardware-in-the-Loop Network Emulation,” in *Proc. of IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2021.
- [173] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 253–264.

- [174] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, “The design and implementation of open vswitch,” in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.
- [175] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P. Schroeder, J. Spaans, and P. Larroy, “Linux advanced routing & traffic control,” in *Ottawa Linux Symposium*, sn, vol. 213, 2002. [Online]. Available: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.620.1459&rep=rep1&type=pdf>.
- [176] Linux Kernel, *Control Groups (cgroups)*, <https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html>, (Accessed 2023-02-02).
- [177] —, *Completely Fair Scheduler*, <https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html>, (Accessed 2023-02-02).
- [178] PassMark Software, *CPU Benchmarks*, <https://www.cpubenchmark.net/>, (Accessed 2023-02-02).
- [179] B. Developers, *Bitcoin referential implementation*, [https://github.com/bitcoin/bitcoin/blob/3c098a8aa0780009c11b66b1a5d488a928629ebf/src/consensus/tx\\_verify.cpp#L166](https://github.com/bitcoin/bitcoin/blob/3c098a8aa0780009c11b66b1a5d488a928629ebf/src/consensus/tx_verify.cpp#L166), Accessed: 2/2/2023.
- [180] N. Becker, A. Rizk, and M. Fidler, “A measurement study on the application-level performance of LTE,” in *2014 IFIP Networking Conference*, IEEE, 2014, pp. 1–9.
- [181] C. Midoglu, L. Wimmer, A. Lutu, Ö. Alay, and C. Griwodz, “MONROE-Nettest: A configurable tool for dissecting speed measurements in mobile broadband networks,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2018, pp. 342–347.
- [182] Cable Labs, *DOCSIS Technology Specifications*, <https://www.cablelabs.com/technologies/docsis-4-0-technology>, (Accessed 2023-02-02).
- [183] L. Wang, C. Li, W. Dai, J. Zou, and H. Xiong, “QoE-Driven and Tile-Based Adaptive Streaming for Point Clouds,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 1930–1934. DOI: 10.1109/ICASSP39728.2021.9414121.
- [184] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. ”O’Reilly Media, Inc.”, 2014.

- [185] J. Yao, S. S. Kanhere, and M. Hassan, “An empirical study of bandwidth predictability in mobile computing,” in *Proceedings of the Third ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, ser. WiNTECH '08, San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 11–18, ISBN: 9781605581873. DOI: 10.1145/1410077.1410081. [Online]. Available: <https://doi.org/10.1145/1410077.1410081>.
- [186] C. Yue, R. Jin, K. Suh, Y. Qin, B. Wang, and W. Wei, “LinkForecast: Cellular Link Bandwidth Prediction in LTE Networks,” *IEEE Transactions on Mobile Computing*, vol. 17, no. 7, pp. 1582–1594, 2018. DOI: 10.1109/TMC.2017.2756937.
- [187] Y. Xiao, M. Krunz, H. Volos, and T. Bando, “Driving in the Fog: Latency Measurement, Modeling, and Optimization of LTE-based Fog Computing for Smart Vehicles,” in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2019, pp. 1–9. DOI: 10.1109/SAHCN.2019.8824922.
- [188] J. Schmid, A. Höss, and B. W. Schuller, “A Survey on Client Throughput Prediction Algorithms in Wired and Wireless Networks,” *ACM Computing Surveys*, vol. 54, no. 9, 2021, ISSN: 0360-0300. DOI: 10.1145/3477204. [Online]. Available: <https://doi.org/10.1145/3477204>.
- [189] T. Sahin and M. Boban, “Radio Resource Allocation for Reliable Out-of-Coverage V2V Communications,” in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, 2018, pp. 1–5. DOI: 10.1109/VTCSpring.2018.8417747.
- [190] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. DOI: 10.1109/JPROC.2014.2371999.
- [191] L. Bonati, S. D’Oro, S. Basagni, and T. Melodia, “Scope: An open and software-defined prototyping platform for nextg systems,” in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '21, Virtual Event, Wisconsin: Association for Computing Machinery, 2021, pp. 415–426, ISBN: 9781450384438. DOI: 10.1145/3458864.3466863. [Online]. Available: <https://doi.org/10.1145/3458864.3466863>.
- [192] F. K. Jondral, “Software-Defined Radio—Basics and Evolution to Cognitive Radio,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2005, no. 3, p. 652784, 2005, ISSN: 1687-1499. DOI: 10.1155/WCN.2005.275. [Online]. Available: <https://doi.org/10.1155/WCN.2005.275>.

- [193] T. Ulversoy, “Software Defined Radio: Challenges and Opportunities,” *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, pp. 531–550, 2010. DOI: 10.1109/SURV.2010.032910.00019.
- [194] Eric Biederman, Daniel Lezcano, Serge Halryn, Stéphane Graber and others, *Linux Containers (LXC)*, <https://linuxcontainers.org>, (Accessed 2023-02-02).
- [195] Software Radio Systems, *srsRAN*, <http://docs.srsran.com>, (Accessed 2023-02-02).
- [196] P. Holme and J. Saramäki, “Temporal networks,” *Physics Reports*, vol. 519, no. 3, pp. 97–125, 2012, Temporal Networks, ISSN: 0370-1573. DOI: <https://doi.org/10.1016/j.physrep.2012.03.001>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0370157312000841>.

# CURRICULUM VITAE

