2019

Scalable string reconciliation by recursive content-dependent shingling

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

SCALABLE STRING RECONCILIATION BY RECURSIVE CONTENT-DEPENDENT SHINGLING

by

BOWEN SONG

B.Eng., University of Victoria, 2017

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2019

Approved by

First Reader

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

Second Reader

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

Third Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

"Science investigates; religion interprets. Science gives man knowledge, which is power; religion gives man wisdom, which is control. Science deals mainly with facts; religion deals mainly with values. The two are not rivals." Martin Luther King Jr.

Acknowledgments

I want to express my greatest gratitude and appreciation towards Prof. Ari Trachtenberg. His perceptive suggestions, thoughtful guidance, and comprehensive mentoring lead to the success of this research study. I never thought I would enjoy life as a researcher and he is the one who made me realize my potential for pursuing a career in research.

I am also thankful for Prof. David Starobinski for his informative suggestions and commentary helping me understand ideas from complex concepts.

I am grateful for Prof. Orran Krieger for examining the practicality of my thesis application, bridging the gap between academia and practice. Especially with industrial proprietary products, useful applications are critical to justify the effectiveness of my solution.

I would like to express my appreciation toward Prof. Osama Alshaykh for his support and wisdom, and my fellow BU students and lab-mates at the Laboratory of Networking and Information System, especially Daniel Wilson, Anish Gupta, Trishita Tiwari, and Anas Imtiaz for sharing their expertise in programming and research.

Last but not least, I am thankful for the emotional and financial support from my loving parents.

SCALABLE STRING RECONCILIATION BY RECURSIVE CONTENT-DEPENDENT SHINGLING

BOWEN SONG

ABSTRACT

We consider the problem of reconciling similar strings in a distributed system. Specifically, we are interested in performing this reconciliation in an efficient manner, minimizing the communication cost. Our problem applies to several types of large-scale distributed networks, file synchronization utilities, and any system that manages the consistency of string encoded ordered data. We present the novel Recursive Content-Dependent Shingling (RCDS) protocol that can handle large strings and has the communication complexity that scales with the edit distance between the reconciling strings. Also, we provide analysis, experimental results, and comparisons to existing synchronization software such as the *rsync* utility with an implementation of our protocol.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 2

1.3 Current State of the Art 3

1.4 Approach 3

1.5 Applications 4

1.5.1 Client Synchronization in Distributed File Systems 4

1.5.2 Cloud-Based Computing 5

1.5.3 Content-Delivery Networks 5

1.5.4 Gossip Protocol 5

1.5.5 Other Use Cases 6

1.6 Contributions 6

1.7 Outline 7

2 Background **8**

2.1 Set Reconciliation 8

2.1.1 Characteristic Polynomial Interpolation 9

2.1.2 Interactive Characteristic Polynomial Interpolation 10

2.1.3 Invertible Bloom Lookup Table 11

2.1.4 Strata Estimator 12

2.1.5 Robust Set Reconciliation 12

2.2 String Reconciliation 15

2.2.1 Rsync 16
2.2.2 String-Recon 17
2.2.3 Uniquely Decodable Shingles 19
2.2.4 Low Latency File Synchronization 20

3 String Reconciliation via Recursive Content-Dependent Shingling 21

3.1 Underlying Data Structures 22
3.1.1 Local Minimum Chunking 22
3.1.2 Hash Partition Tree 23
3.2 Iterated Backtracking 26
3.3 The Main Protocol 27
3.4 Asymptotic Analysis 29
3.4.1 Communication Complexity 29
3.4.2 Time Complexity 30
3.4.3 Space Complexity 32
3.5 Worst-Case Performance 33
3.5.1 Sparse String Edits 33
3.5.2 Partition Probability and Cascading Partition Mismatch . . . 34
3.6 Failure Modes 38
3.6.1 Collision Probability 39
3.6.2 Reconciliation Failures 41

4 Evaluation 43

4.1 Varying Protocol Parameters 44
4.2 Varying String Inputs 52
4.3 Comparison to Existing Work 59
4.3.1 Synchronizing Single Files 59
4.3.2 Synchronizing Folders 62

4.4 Summary of Performance 67

5 Conclusion **68**

5.1 Future Work 69

5.1.1 Parameters and Data Structures 69

5.1.2 Underlying Algorithms 69

5.1.3 Incremental Partition 70

5.1.4 Hybrid Approach 70

Curriculum Vitae **78**

List of Tables

2.1 Robust set reconciliation performance. 15

3.1 An example of a multiset of hash shingles from the partition tree in Figure 3·3. 27

3.2 Set reconciliation fail rate. 42

4.1 The goodness of fit of the linear polynomial curve model for the time cost of every operation arranged from top to bottom in Figure 4·9 listed from left to right. 57

4.2 The goodness of fit of the linear polynomial surface model for planes in Figure 4·11 comparing total communication cost of RCDS and *rsync* reconciling various input strings. 61

4.3 Git repository versions used comparing synchronization performance between RCDS and *rsync* shown in Figures 4·13 and 4·14. 66

List of Figures

2·1 The problem of set reconciliation. 9

2·2 A quadtree visual example. 14

2·3 The rsync protocol visual illustration. 17

3·1 An example of getting content hash values using the rolling hash algorithm with window size $w = 3$ and hash space $s = 16$. 23

3·2 An example of content-dependent chunking using minimum partition distance $h = 2$. 24

3·3 An example of content-dependent partition tree. 25

3·4 An example de Bruijn digraph of the level 1 partitions from the partition tree in Figure 3·3. 26

3·5 Potential cut-point probability based on Lemma 3.5.1. 37

3·6 Upper-bound probability of hash collisions vs. number of recursion levels as a function of maximum number of partitions at each recursion p. 40

4·1 Stack plot of internal operations in RCDS comparing communication cost for different partition tree parameters. The operations in the stack plot include CPI-Based set reconciliation and communication of literal data arranged from top to bottom as regions between the surfaces. The region between total communication and communication of literal data is the communication cost of set reconciliation. The literal data are strings of the unmatched terminal partitions. 46

4·2 Comparing the percentage of unmatched tree partitions vs. the total number of partitions for different partition tree parameters. The percentage of unmatched tree partitions decreases as the partition tree grows which shows the effective usage of partition hierarchy. 47

4·3 Comparing the total number of unmatched tree partitions for different partition tree parameters. The number of unmatched partitions increases as the partition tree grows and exceeds the fixed 1000 edit burst distance. The extra unmatched partitions are from the partition tree hierarchy which is reflected by set reconciliation communication cost rather than the cascading effects of mismatch partitions. This is confirmed by the communication costs plot in Figure 4·1 where the set reconciliation cost increases as the partition tree grows while the amount of literal data transferred decreases. 48

4·4 Stack plot of internal operations in RCDS comparing time cost for different partition tree parameters. The operations in the stack plot include partition tree construction, CPI-Based set reconciliation, and string reconstruction arranged from top to bottom as regions between the surfaces. The top surface corresponds to the total string reconciliation time cost. 50

4·5 Comparing space complexity for different partition tree parameters. The space requirement increases as the partition tree grows in size with respect to the number of maximum partitions and recursion levels. 51

4·6 Communication performance reconciling strings of different sizes and edit burst distances using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The performance is measured by the percentage of the total number of bytes communicated during reconciliation vs. input string size in bytes. Using partition trees with a higher number of partition levels adds more overhead to communication cost when input string size is small but is more scalable to larger input strings. 53

4·7 Percentage of unmatched partitions vs. the total number of partitions reconciling strings of different size and edit burst distance using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The unmatched partition percentage of L5 partition tree is much lower than that of the L4 partition tree when reconciling the same strings. As the input string length increases, the percentage of unmatched partitions stays almost constant while increasing in a steady trend as the edits burst distance grows. The number of unmatched partitions is proportional to edit distance, referred as d in the analysis from Section 3.4. 54

4·8 Total time cost reconciling files of different size and edit burst distance using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The time used to reconcile different strings using L4 and L5 partition trees increase linearly to the input string size while staying almost constant to increasing edit burst distance. Reconciling strings using L5 partition trees costs more time than that of L4 due to the extra level of recursive partition creating more partitions for a given string. 55

4·9 Time cost of every internal operation of RCDS in a stack plot including partition tree construction, CPI-Based set reconciliation, and string reconstruction for reconciling different length strings with 1000 edit burst distance. Each curve is fitted with Linear polynomial curve model with the goodness of fit presented in Table 4.1. The time cost corresponds to the linear time complexity analysis described in Section 3.4. 56

4·10 Space complexity reconciling strings of different size using L=4 and L=5 level partition trees with a maximum of p=4 partitions.An L5 partition tree can handle longer strings than an L4 partition tree by creating more partitions in the extra level and find more common substrings between the two reconciling strings to save communication cost. However, an L5 partition tree also requires more hash values to represent the extra partitions, thereby, costing more space. 58

4·11 Comparing RCDS and *rsync* on the total number of bytes transmitted reconciling different text files. The green surface represents communication cost for the *rsync* utility which grows linearly to input string length whereas the blue surface representing the communication cost of RCDS remains sub-linear to input string length increase. Both RCDS and *rsync* require more communication cost to reconcile strings with more substantial edit burst distance, and RCDS appears to have a larger overhead. 60

4·12 Line of intersection of linearly polynomial fitted planes from Figure 4·11 projected to a 2-D plot of edit burst distance vs. input string length. The blue area defined by the line of intersection and x-axis describes the circumstances where RCDS performs better than *rsync* in terms of communication cost. 61

4·13 Comparing the communication performance of RCDS and *rsync* when synchronizing the entire repositories between the latest and the second latest releases. In most of the cases, the communication cost of RCDS is much less than that of *rsync*. The performance includes all communication cost reconciling edited files between releases. The average percentage of file edits determines the overall performance of RCDS and *rsync*. 64

4·14 Comparing the time performance of RCDS and *rsync* synchronizing the entire repositories between the latest and the second latest release. The time cost is primarily determined by the number of different files between releases. 65

List of Notations

lg	.....	Base 2 Logarithmic
$\mathbb{N}^+$	.....	Positive Natural Numbers
$O()$	.....	Big-O Notation
$\|X\|$	.....	Cardinality or size of the set X

Chapter 1

Introduction

1.1 Motivation

The rise of cloud-based storage systems such as Google Drive, Dropbox, and iCloud extend the local file system into the cloud. These cloud storage services provide extra storage space, backup and synchronization, and file sharing across different devices. The files shared on different machines inspire corporations and allow different users to view and edit the same files of their latest version. Behind the scenes, these cloud storage services utilize Distributed File Systems (DFS) software such as Google File System (GFS) [23], Andrew File System (AFS) [31], and Ceph [59] to support file availability and consistency everywhere.

The cloud-based distributed file systems interact with client computers to synchronize with the latest version whenever available. Generally speaking, most of the information updates on the cloud start from client devices such as personal laptops, smartphones, and tablets which rely on wireless ad hoc network communication that often disconnect or under restricted bandwidth. The disconnected client devices hold cached files that may diverge from their copies in the cloud upon asynchronous local and remote edits. The system complexity naturally grows with files shared and edited by multiple users and devices. The basic synchronizing method, downloading and uploading the entire changed files, has a high communication cost occupying a

large amount of bandwidth for an extended amount of time. Also, file updates in this network can be incremental, which means most of the file could be unchanged. We could exploit the similarities between different file versions and only communicate the file differences to perform update and synchronization. Therefore, an efficient string reconciliation protocol that achieves eventual consistency would serve as a natural foundation for improving such systems.

It is also worth mentioning that the problem of defining authoritative sources during the reconciliation process is application specific and parallel to our problem. In large-scale distributed systems, common solutions to this parallel problem include referencing time from vector clocks and using conflict-resolving data structures such as the Conflict-free Replicated Data Types [51].

1.2 Problem Definition

The problem of string reconciliation considers two physically separated hosts, Alice and Bob, that are attempting to reconcile their local versions of a string using the minimum amount of communication. For example, Alice might have a string "snack," Bob has "snake," and their objective is for Alice to obtain just enough information upon the string similarity to assemble the string "snake" from Bob.

The most basic way of reconciling string difference is for Bob to transfer the entire string "snake" to Alice. We refer to this reconciliation method of transferring a whole string as the *Full Sync*. The *Full Sync* method is a baseline protocol that does not consider similarities between two reconciling strings or allow incremental string update.

1.3 Current State of the Art

The current state-of-the-art protocols that handle string reconciliations include [1, 10, 35, 39, 45, 49, 58] which are effective with small size string inputs. Their computation or communication cost grows in an exponential rate when scaling up the input string length. The ubiquitous *rsync* utility [12] based on fixed-size partitioning and Content-Dependent Chunking protocols such as [47, 62] are capable of handling large string reconciliation, however, suffer a communication cost that scales linearly in input string length.

As a widely accepted software, the *rsync* utility uses a bandwidth efficient algorithm to synchronize files across the network. The rsync algorithm is scalable to handle large file size and is efficient to synchronize files with great edit distance. Since creation, the *rsync* utility has served many occasions that require to maintain file consistencies and differential backups. The current use of *rsync* utility exists in places such as DFS client software that synchronizes files on client hosts with copies in the DFS [31, 38], GitLab to maintain git repositories [25], and inside of Google Cloud to synchronize and migrate data between *storage buckets* [28]. Rsync is also a standard Linux synchronization utility included in every popular Linux distribution serving local and remote file synchronization. We compare the performance of our protocol to that of the *rsync* utility and show the superiority of our protocol under certain circumstances.

1.4 Approach

We approach the problem of string reconciliation by reducing it to a similar problem known as the set reconciliation problem [41] which is explained in Section 2.1. The key of the reduction is to represent the string as a set of elements, of which each pre-

serves enough information to be unambiguously pieced back into the original string. An underlying challenge is to keep the number of symmetric differences between the string-element sets close to that of the strings dissimilarity metric. We use *edit distance* to quantify the differences between two strings including insertion, deletion and replacement. In all cases, our string reconciliation efficiency goal is to minimize the total number of bytes transmitted, the number of rounds of communication, and the amount of computation needed for the entire process.

1.5 Applications

The problem of string reconciliation servers various applications in different large-scale networks, especially for networks where the connections are restrictive in terms of bandwidth, consistency, and availability.

1.5.1 Client Synchronization in Distributed File Systems

A direct application of string reconciliation is to maintain file consistency in distributed systems where incremental file updates is a common operation. According to a survey on cloud storage services [38], the current standard data synchronization traffic in cloud storage systems of major vendors can be significantly reduced by using more economical synchronization protocols. We specifically consider the network traffic between clients and cloud, where the systems try to sustain file consistency with client computers. By using string reconciliation, the file systems can maintain consistency of incremental file edits using the minimum amount of communication.

1.5.2 Cloud-Based Computing

Another application is to help orchestration software to maintain incremental *snapshot* updates. The orchestration software takes care of fault-tolerance for distributed systems by keeping *snapshots* of deployed instances and keeping the data of working instances periodically. In case of an instance failure, the orchestration software can spawn the latest *snapshot* of the instance to continue from the state when it was still working. The *snapshot* service prevents an instance from losing all progress, however, is expensive if not done incrementally [2]. By using string reconciliation, the system can just save the incremental changes to the snapshot as a disjoint component.

1.5.3 Content-Delivery Networks

Our approach towards the string reconciliation problem can also help with the problem of distributing large files across *content-delivery networks*. Transferring large files with individual point-to-point connections incurs the problem of wasteful bandwidth consumption [50]. An improvement is to parallelize download operations [7] by accessing the same file packets from multiple mirror sites. Specifically, an approach from [8] divides the range of file packets into disjoint sets, each of which can be downloaded from a different source. Our approach of reducing a large string into a multiset of substrings can help divide a whole file packet into disjoint sets of packets to be unambiguously pieced back later at the destination.

1.5.4 Gossip Protocol

The problem of string reconciliation is also applicable to various types of gossip protocols that only rely on peer-to-peer communications such as distributing networked data [57], discovering resource [30], and data synchronization [54]. These gossip

protocols determine and reconcile the differences between data stored in physically separated locations in non-centralized systems under minimized communication cost.

1.5.5 Other Use Cases

Other use cases outside the field of computer networking such as symbol sequencing from a given alphabet which is usually applicable to DNA sequencing corresponds to RNA and protein [15]. The problem is also useful for fuzzy extractors which are used in noisy biometric data encryption [14].

1.6 Contributions

We study fixed-rounds protocols for reconciling strings and propose a new scalable protocol that reduces a significant amount of communication cost compared to existing works. Our main contributions are thus:

1. We have designed a new string reconciliation protocol whose communication complexity is sub-linear in string size under certain circumstances. The new protocol is scalable to long strings.

2. We have provided analysis of our protocol including upper-bound failure probability, expected and worst-case communication and computation costs.

3. We have implemented our protocol as synchronization utility available for *MAC OS* and *Linux*. We have also evaluated its performance over a variety of inputs, including a comparison to the current standard *rsync* protocol.

1.7 Outline

The outline of this thesis is as the following. Section 2 includes a definition of the underlying set reconciliation problem, descriptions of existing bandwidth-efficient set reconciliation protocols, and a survey of related work in string reconciliation. Section 3 presents and analyzes our novel string reconciliation protocol. Section 4 presents the performance of the new protocol against different parameters, inputs, and existing work. Finally, Section 5 states our conclusions and directions of future work.

Chapter 2

Background

2.1 Set Reconciliation

The problem of set reconciliation serves as the foundation of our string reconciliation approach. The performance of our approach is dependent on that of the underlying set reconciliation algorithm. We define the problem of set reconciliation as shown in Figure 2·1. Alice and Bob are hosts with sets of data S_A and S_B respectively. The goal is to determine the union of the two sets while transferring their symmetric differences, elements C and D in our figure, to achieve data consistency between Alice and Bob. The challenge is to minimize the total amount of communication regarding the total amount of bits exchanged between the two reconciling hosts.

In the following section, we will introduce some bandwidth-efficient set reconciliation protocols:

- Section 2.1.1 – *CPISync* [41] is a deterministic though computationally challenging protocol.
- Section 2.1.2 – *Interactive CPI* [42] is a practical protocol that recursively uses the *CPISync*.
- Section 2.1.3 – *IBLT* [26] is a probabilistic model based on the Bloom filter data structure.

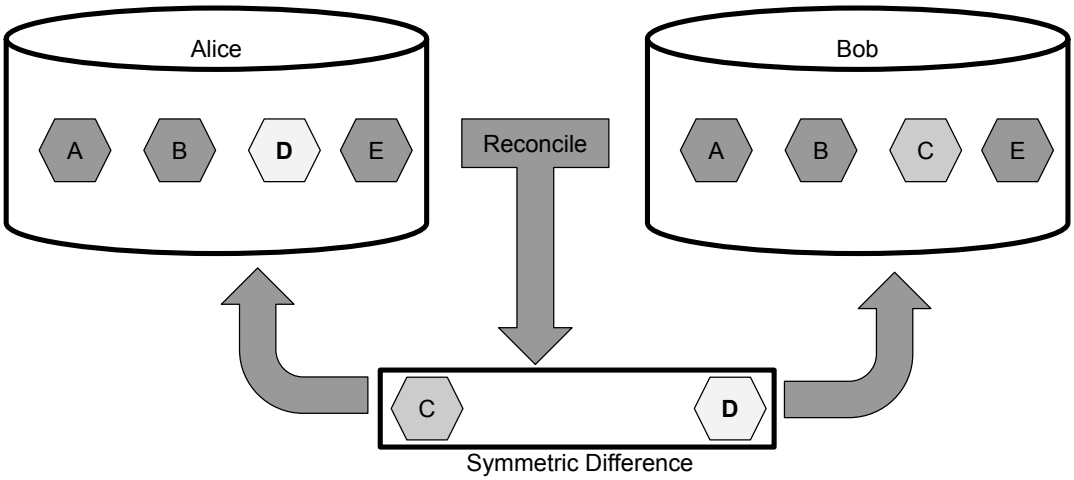

Figure 2·1: The problem of set reconciliation.

- Section 2.1.4 – *Strata Estimator* [18] uses the *Flajolet-Martin* algorithm and the *IBLT* to estimate the number of symmetric difference between two reconciling set.

- Section 2.1.5 – *Robust-Recon* uses a *quadtree* data structure and the IBLT to archive single-round set reconciliation without prior knowledge on the upper-bound of the set symmetric difference.

2.1.1 Characteristic Polynomial Interpolation

The Characteristic Polynomial Interpolation (CPI) set reconciliation algorithm [41] represents a set by its characteristic polynomials and reconciles based on the polynomial evaluations at a collection of evaluation points. The common knowledge between the two reconciling hosts includes a set difference upper bound $\bar{m}$, a finite field that includes all elements from both sets, and $\bar{m}$ number of random evaluation points. The protocol starts with two hosts, Alice and Bob, evaluating their own polynomials based on all evaluation points. The results are transmitted over for Bob to perform

finite field division and recover coefficients of the reduced rational function. Without a prior knowledge on the upper bound of set difference, the protocol goes through a combination of guessing and verifying to guess a large enough upper bound by iteratively doubling the $\bar{m}$ as a guessing value.

The protocol has a communication complexity of $O(b(m+k))$ where b is the length of bitstrings representing each element, m is the number of set symmetric difference, and k is a small redundancy value to reduce the probability of failure. In other word, the cost of communication is only affected by the amount of symmetrical difference, regardless of the reconciling set size. Unfortunately, the protocol has a computation complexity of $O((m+k)^3)$, which makes it only feasible to reconcile a small number of symmetric differences.

2.1.2 Interactive Characteristic Polynomial Interpolation

The *Interactive CPI* [42] is a recursive solution to remedy the large computation complexity of *CPI* algorithm. It uses the notion of divide-and-conquer, recursively partitions the reconciling set in different space buckets and reconciles each partition. The key idea is to keep a small $\bar{m}$ for *CPI* to process fast and probabilistically try to reduce m in every recursive partition to meet with the fixed $\bar{m}$ value. The full list of set differences is obtained by taking the union of recovered sets as the last partition. The controlling parameter of *Interactive CPI* includes partitioning factor p, redundancy factor k, and an upper bound for set differences $\bar{m}$. The k and $\bar{m}$ are both parameters inherited from the *CPI* algorithm. The partitioning factor is the new parameter controlling the number of partitions in each recursion.

The recursive set partitions is expected to isolate the set symmetric difference into different buckets and reduce the workload for the CPI algorithm at each step. However, the worst-case scenario is when all set differences are concentrated at one

space bucket in every recursive partition causing a extended amount of partitions without reducing the amount of symmetric differences. The *Interactive CPI* can promote the chance of expected-case isolating set symmetric difference into different partition buckets by hashing the set elements.

The *Interactive CPI* protocol has an expected computation complexity of $O(mpb(\bar{m}^2+k))$, which is sufficient for reconciling two sets of data with small amount of symmetric difference. The communication complexity is between the expected $O(mb)$ and the worst-case scenario of $O(mb^2)$.

2.1.3 Invertible Bloom Lookup Table

The IBLT [26] is a table version of Bloom filter data structure that can be used in a set reconciliation protocol. The IBLT compresses the entire set of a host into a table of hash values and extracts the set symmetric difference on another host by inserting all its set elements. Specifically, Host Alice inserts all of its elements into a two-field table based on the XOR of all of its inserted elements through r number of designation hash functions, $d_{1...r}(\cdot)$, and one fingerprint hash function $fp(\cdot)$. The two fields are referred to as *keySum* and *fpSum* respectively. For r number of designation functions chosen, there are r different locations in the IBLT an element is inserted into, given the chosen hash functions have low collision rates. When extracting differences, due to $XORsum$, the insertion of all set elements from the other host removes the same elements from the IBLT, adds the differences, and leaves the set symmetric differences in the IBLT after the process.

To display these symmetric difference left in the IBLT, the algorithm searches for pairs of *keySum* and *fpSum* that satisfies $fp(keySum) = fpSum$, which indicates that the *keySum* itself is one key. The next step is to reinsert this key into the IBLT to remove it. This peeling process, removing one element at a time could create more

table cells containing only one element to continue the process, or else the process fails or ends if IBLT is empty. Based on [26], a full displaying process has a high success rate if the number of field pairs in an IBLT is at least the size of the symmetric differences times a small constant. In another word, the size of IBLT determines the number of set symmetric differences it can recover.

2.1.4 Strata Estimator

The *Strata Estimator* [18] is a way to estimate set differences by sampling set element into a IBLT hierarchy using a probabilistic counting algorithm [20]. The estimator creates 32 IBLT's of 80 cells, which of which can successfully extract about 60 set elements. When estimating the symmetric difference between two sets, the estimator uses a hash function to map all set elements into a space. In the IBLT hierarchy, each IBLT accepts the elements hashed into their corresponding space. For 32 IBLT's accepting elements hashed into a space s, each IBLT corresponds to $\frac{s}{2^i}, i = 1, ..., 32$ of the space. For example the first IBLT would corresponds to 1/2 of the space and, therefore, containing about 1/2 of the set elements. To estimate the symmetric difference, two reconciling set each creates 32 IBLT's and try to extract set differences form each pairs of IBLT's corresponding to the same hash space. The estimator takes the successfully extracted IBLT that corresponds to the largest hash space and double the extracted number of symmetric difference as the estimation.

2.1.5 Robust Set Reconciliation

The *Robust-Recon* [9] targets a data synchronization situation in the context of set reconciliation problem. The goal is to synchronize data under a known communication budget at a cost of tolerating incomplete reconciliation. The set differences are defined by the measurement of Earth's Mover Distance (EMD) as minimum difference

matching between two sets. The allowed set differences, if not considered as seeds of accumulators to large differences, are either considered tolerable or intentionally introduced. Some example contributors to set differences include data variety from noise or compression, rounding errors, and privacy-preserving data. By allowing such differences and avoiding *exact* set reconciliation, the *Robust-Recon* provides a conditionally better communication complexity than the information-theoretic lower bound [41], which is data representing the exact amount of symmetric differences between two reconciled sets.

The *Robust-Recon* uses random shift, *quadtree* data structure, and IBLT. The random shift plays a role in reducing the probability of failed recovery; The *quadtree* data structure is for grouping the similar data from distance measurement, and IBLT is for compressing set differences to reduce communication complexity.

Quadtree

A tree data structure that recursively partitions a space into four quadrants and registers the number of elements in each subspace. The root of a tree contains the entire space of the set and has the value as the number of elements in the set; Each child of a node has the range of the quadrant as key and count of elements in the range as value. A full tree would have its leaves each correspond to an element of the set with value 1. For a visual example, Figure 2·2 shows some elements in 2-D space and how they would be represented in a *quadtree*. The difference label on the right-hand side shows the number of symmetric difference when comparing each *quadtree* level for the same set of elements. Since we are comparing the range and the count of elements in the range, as we go up to the root, the range of each node gets larger, and level-wise amount of difference gets smaller.

The *Robust-Recon* protocol is non-resumable and can not benefit from saving pre-

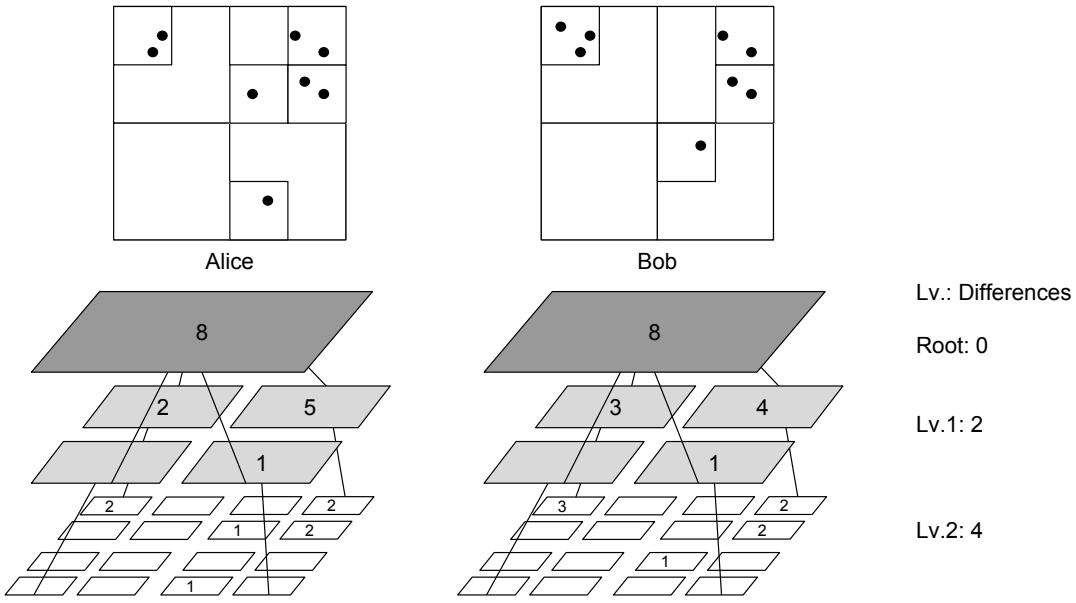

Figure 2·2: A quadtree visual example.

vious computations. Every reconciliation has to go through the entire process due to the new choice of points from a random shift. Since IBLT's setup is the same for every quadtree level and determines the maximum amount of retrievable differences, the size of IBLT becomes a tuning parameter controlling trade-off between reconciling accuracy and message size. By moving up quadtree levels to lower reconciling differences, the accuracy of reconciliation decreases. Since $\bar{m}$ is the only controlling factor for communication cost, *Robust-Recon* is relating the communication cost as a trade-off with reconciliation quality. Due to its overhead, the *Robust-Recon* only performs better than an exact set reconciliation protocol when the number of large element differences is much less than the number of tolerable element differences.

Complexity	**Communication**	**Computation**
Robust-Recon	$O(\alpha\bar{m}\,log(n\Delta^d)log(\Delta))$	$O(dn\,log(\Delta))$

α = Redundancy factor
d = Dimensions of set data
Δ = number of all d integers in a set
n = Number of elements in a set
$\bar{m}$ = Upper-bound on the number of symmetric difference

Table 2.1: Robust set reconciliation performance.

Fortunately, for a modern applications such as backing-up family photos where data accuracy is less important, this protocol would perform well under its limitations.

2.2 String Reconciliation

The problem of string reconciliation and its direct application to file synchronization inspired a rich body of work including the most wildly used *rsync* utility [12]. Moreover, some existing algorithms share our approach of reducing the problem of string reconciliation into a set reconciliation problem including [1, 35, 62, 22]. Other work such as [16, 49, 63] use error-correcting codes to reconcile strings under limited amount of edit distance.

In this section, we will elaborate on existing file synchronization protocols, including:

- Section 2.2.1 – the *rsync* algorithm [55],
- Section 2.2.2 – *String-Recon* [35] based on shingling the input,
- Section 2.2.3 – *Uniquely Decodable Shingles* [35] that attempts to reduce computation complexity for the *String-Recon* protocol,

- and Section 2.2.4 – *Low Latency File Synchronization* [62] which is based on content-dependent partitioning.

2.2.1 Rsync

The *rsync* algorithm [55], later developed into the ubiquitous file synchronization utility [12] for Linux platform, is one of the most practical solution to the problem of string reconciliation. The *rsync* algorithm uses fixed partition and rolling checksum to recognize the matching data between two strings and reconcile their differences. In a specific situation as shown in Figure 2·3, between two separate hosts, where Bob holding string σ_B wishes to update a similar string σ_A on Alice. Alice would first compute 128-bit strong and 32-bit weak rolling checksums for every w sized non-overlapping partition of σ_A. She sends all these checksums to Bob where Bob tires to find all matching data. Bob tries to find matches to his string partitions and partition offsets by using a *windowing* method moving w sized window through σ_B calculating the checksums for each offset. Upon each checksum calculation, Bob searches for matching weak and strong checksums from Alice's checksum list and construct sequences of edit instructions for unmatched parts of the string. To save computation cost, on every iteration where a match is found, Bob sends edit instructions for the preceding unmatched data to Alice.

The *rsync* requires host Alice and Bob to communicate data references with respect to absolute position within the string which can not be reduced to a set reconciliation problem. In addition, the number of hashes communicated between Alice and Bob is linear to the reconciling string length given a fixed block size. Therefore, the communication complexity is linear to the length of the input string.

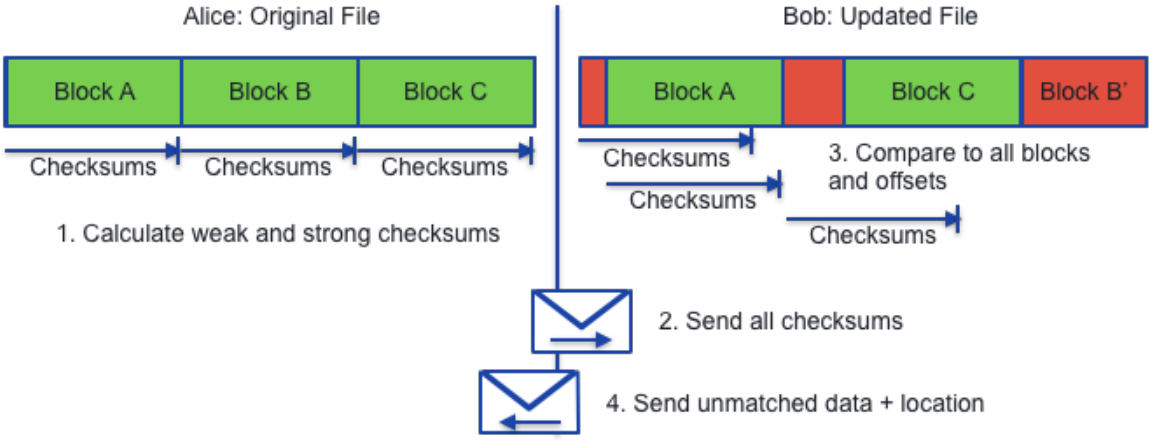

Figure 2·3: The rsync protocol visual illustration.

2.2.2 String-Recon

The *String-Recon* protocol [1], on the other hand, has a communication complexity of $O(m \lg^2(n))$ sub-linear to the length of reconciling strings, where m is the edit distance between the pair of reconciling strings and n is the string length. It transforms the input strings into multisets of fixed length shingles and synchronizes using set reconciliation algorithms. Unfortunately, the protocol uses an exhaustive backtracking method while unambiguously piece back the string from a set of shingles, places exponential computation complexity when scaling up the reconciliation string size.

The *String-Recon* protocol has the following detailed steps: hosts Alice and Bob break their input strings σ_A and σ_B into two multisets of shingles. The multisets are made into sets of unique elements, S_A and S_B, by concatenating shingles with their occurrences in the multisets. For Alice's string to be updated, Bob has to compute a weighted de Bruijn digraph based on S_B and trace through the digraph exhaustively to find all possible path till the path recreates the string. At last, Alice and Bob edit their modified de Bruijn digraphs from the reconciled two new sets, S'_A and S'_B, and backtrack the graph to generate σ'_A and σ'_B.

k-shingling

The *k-shingling* method to breaks a string into a set of substrings by a rolling a fix-sized window through the entire string character by character. The length of the mask is directly controlling the length of all shingles and the performance of the protocol. If a mask is very short, each substring will have a higher chance to be duplicated and increase the backtracking time, and if the mask is too long, the substrings will not be sufficiently small to be efficient communicated. As a default setting, the shingle length is set to base 2 log of input string length.

De Bruijn Digraph

The protocol converts a set of shingles back to a string through exhaustively back-tracking a de Bruijn digraph. The de Bruijn digraph used is a weighted digraph with each vertex containing an unique first $k-1$ characters of a shingle and the last character as an outward edge. The weight of an edge is the occurrence of the shingle.

The head and tail shingles are made unique by adding stop-word '$' at the front and end of the string. Bob can follow the digraph path while arranging potential paths at each step in lexicographic order. Every time Bob crosses a path, he decreases the weight of the path by one. A path dies whenever there is no more exiting path that has a weight higher than 0 or that the path hits the ends vertex. A complete path uses up all weights in the digraph. If the resulting string of a complete path matches with the original one, Bob records the number of complete path he traced before reaching this path and send to Alice to recreate the string.

2.2.3 Uniquely Decodable Shingles

As an extension to *String-Recon* protocol, [35] provides an online algorithm inspecting the unique decodability (UD) of a string. Upon creating a multiset of shingles of the input string, the protocol merges shingles that prevents the string from being uniquely decodable from the set. The online streaming algorithm goes through each character of the string once and storing the outcome of each increment in a de Bruijn digraph.

In the streaming process, each digraph vertex has two Boolean properties of whether the vertex belongs to a cycle and if it is previously visited, and each edge connecting two vertices contains a value of occurrence. The string is considered not UD if an incremental character belongs to an existing cycle in the string with a different prefix or an incremental character is seen before and has a different prefix.

In a shingle-based string reconciliation protocol, hosts Alice and Bob both hold a similar string and break their strings into shingles of q length. The algorithm uses $q = 2$ and believes it could be useful to q-gram, q characters shingle, by dividing a q-gram before its last character into two parts. Using a set reconciliation protocol [42], Alice and Bob reconcile their set of shingles and merge the none-UD causing shingles by combining two bigram vertices into one until the set is UD. The merging protocol may produce uneven length shingles, but would not affect the decoding process. After necessary merges, both Alice and Bob sort their set in canonical order and exchange merged shingle's indices. The last step is to backtrack a Eulerian cycle which is guaranteed by this algorithm to obtain a unique reconciled string.

The online streaming algorithm can determine if a string is UD with absolute confidence. However, adapting the algorithm into a shingle-based string reconciliation protocol unpredictably increases shingle sizes and raises concerns about the increasing communication cost. Furthermore, the message size for the merging algorithm has

the worst case of $O(n \log(n))$ for exchanging indices which would be dominating the communication complexity, where n is the number of shingles before merging.

2.2.4 Low Latency File Synchronization

The Low Latency File Synchronization (LLFS) protocol [62] uses a content-dependent chunking method from [53] to partition the reconciling strings into multisets of substrings on the reconciling hosts Alice and Bob where Bob wishes to update the string content on Alice with his local string. The partitions are hashed and reconciled using a set reconciliation algorithm. After calculating the symmetric difference between the two multisets of partitions, Bob sends the Alice her missing string partitions, bit vectors that indicates partition arrangement differences and unmatched partition position sequence.

Compare to *rsync*, the LLFS uses content-dependent partitioning method to maximize partition similarities between two reconciling strings with similar content, and employs the set reconciliation algorithms to avoid sending hash values directly. However, the protocol lacks a way to reassemble partitions and relies on sending sequence and bit vectors which, while reducing a large amount of overhead, persists its communication cost on a linear scale to the reconciling string size.

Chapter 3

String Reconciliation via Recursive Content-Dependent Shingling

Our *Recursive Content-Dependent Shingling* protocol is designed to exploit content similarities between two reconciling strings. The protocol distinguishes differences between the input strings by reconciling their multisets of substring hashes and transfer the substrings unknown to the other party. We use a *local minimum chunking* technique, inspired by the *Rabin-Karp* algorithm [34] and the *local maximum chunking* method [5], to extract substrings based on non-strict local minima over content hash values. By using 64-bit hash values to represent string partitions, we create a multiset of hash shingles concatenating hash values of two adjacent partitions and compose a de Bruijn digraph by turning shingles into pairs of vertex and edge. We employ one of the set reconciliation primitives mentioned in Section 2.1 on the hash shingle multiset with its counter party and then transfer substrings corresponding to hash values unknown to the other party. Finally, we use an exhaustive *Backtracking* method [52] to uniquely decode the multiset of shingles back to a string. We recursively partition the string to create subpartitions and only transfer the partitions at the bottom level of the partition tree to reduce required communication cost.

For the following sections, we will elaborate on the protocol details:

- Section 3.1 – *Underlying Data Structures* presents the construction of partition

tree and transformation of hashes to a multiset shingles.

- Section 3.2 – *Iterated Backtracking* describes a modified exhaustive backtracking method.
- Section 3.3 – *The Main Protocol* contains detailed procedure reconciling strings between hosts Alice and Bob.
- Section 3.4 – *Expected-Case Analysis* presents the Communication, Time, and Space complexity.
- Section 3.5 – *Worst-Case Analysis* describes the situation where protocol performs the worst.
- Section 3.6 – *Success Rate* provides an upper-bound on the overall probability of failure.

3.1 Underlying Data Structures

Our protocol maintains a *content-dependent partition* hierarchy based on the *p-ary* tree data structure where p is the maximum number of children a node could have. We build the tree from partitioning a string recursively using a *local minimum chunking* method, and each partition corresponds to a node in the tree. Our algorithm requires reconciling strings partitioned in a similar manner using the same parameters to increase the chance of common partitions.

3.1.1 Local Minimum Chunking

We adapt the approach from [5] to our *local minimum chunking* method to partition a given string based on its content hash values. We obtain the hash values by a rolling

hash algorithm, as described in Figure 3·1, that could produce pseudo-random values based on the context of the input string using a fixed window size w and hash space s. We move the rolling hash window through the input string one character at a time and hash the w-length substring inside the window to a number in the space s. The resulted hash value array would be $N - w + 1$ in length, where N is the number of characters in the input string.

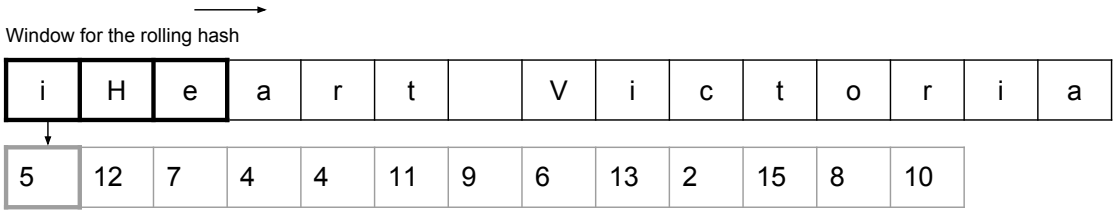

Figure 3·1: An example of getting content hash values using the rolling hash algorithm with window size $w = 3$ and hash space $s = 16$.

We then partition the string based on the hash value array using a minimum partition distance h to control the minimum chunking size. Potential partition places include all locations of non-strict local minimum hashes considering h hash values in both directions. Shown in Figure 3·2, there are 3 values in the array that satisfy as non-strict local minimum hashes for $h = 2$. With the minimum partition length $h = 2$, the second hash value 4 is, therefore, not a valid partition spot. Using the example hash value array, we obtain 3 partitions for the phase "iHeart Victoria". In addition, the minimum partition distance also defines an upper-bound on the number of partitions of a string. The greatest number of partitions of a string is $p = N/h$.

3.1.2 Hash Partition Tree

We build a hash partition tree based on string partitions from the *local minimum chunking* method and hash each string partition by a 64-bit hash function, $H(\cdot)$, to store in a *p-ary* tree. From our previous example, we created 3 partitions to fill in the

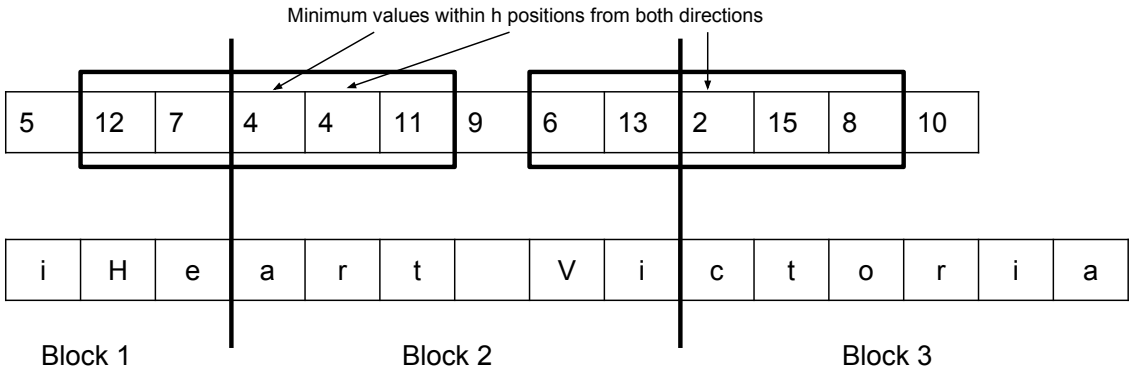

Figure 3·2: An example of content-dependent chunking using minimum partition distance $h = 2$.

first level of the partition tree, shown in the Figure 3·3. We can grow the partition tree by creating subpartitions recursively for L times where L is the levels of recursion. Since we are unlikely to partition a substring further with the same parameters, we reduce the partition parameters, s and h, at every next recursion level by a fixed ratio and apply them for the entire level of partitions. In our implementation, we calculate the partition parameters at each recursion level based on p and L: $s = wp^{L-l+1}$ and $h = N/p^l$, where $l = \{1, 2, ..., L\}$. We consider p and L to be the main tuning parameters. Figure 3·3 shows a 2-level hash partition tree where we refer the second level partitions as *terminal strings*.

The partition tree tolerates changes to the input string without affecting most of the partitions. For example, if we change the phrase to "Heart Victoria" with the "i" removed from the front. According to Figure 3·1, the edits to the hash values array would be removing the first hash value. Figure 3·2 shows that this change would not affect the the relative partition positions, and the resulting partitions would be "He", "art Vi", and "ctoria". If we extend our recursive partition level, we also expect slight changes to the subpartitions in the first branch. Since partition decisions for other branches are isolated after the first level, the rest of the partition tree should stay the

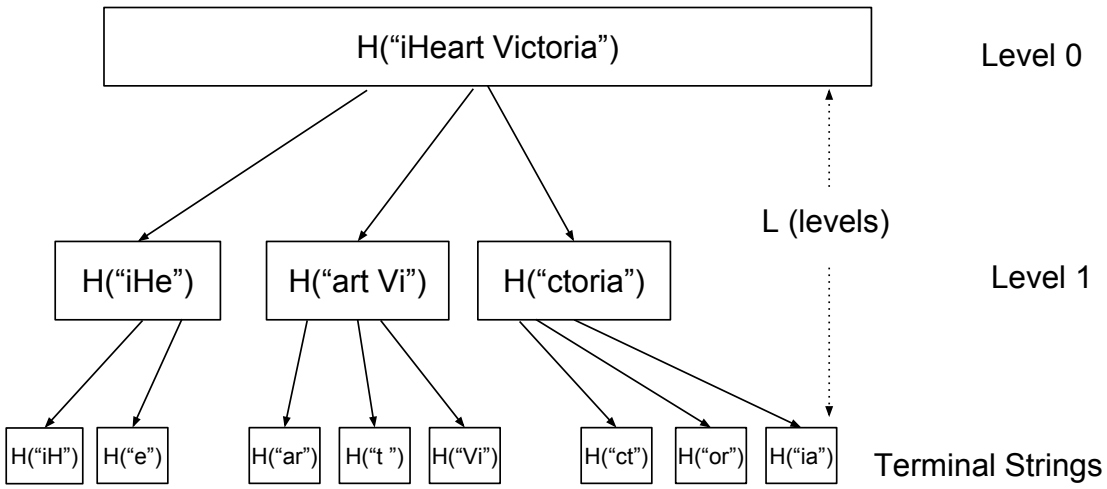

Figure 3·3: An example of content-dependent partition tree.

same. This property not only shows how small differences between two reconciling strings are likely to generate similar partition trees, but also allows maintaining a partition tree for incremental string editions. We avoid terminal strings falling shorter than their hash representations by setting a minimum terminal string size and skip further partitions to smaller substrings.

The minimum terminal string size caps the communication cost reduction from increasing the number of partition levels. Normally, by partitioning a string into smaller substrings, we reduce the amount of literal data to be transferred if the partition contains string edits compared to its counter part. However, if we stop partitioning a substring further than a certain size, the edits stay in the same substring moving to the next level of partition tree and do not contribute to additional communication cost.

Converting a Partition Tree into a Shingle Set

The last step is to transform the partition tree into a multiset of shingles before we use a set reconciliation algorithm to solve the reduced problem of string reconciliation.

We start by constructing a de Bruijn digraph at every level using a 2-shingling method modified from Section 2.2.2. Figure 3·4 shows a de Bruijn digraph created from our example partition tree at the first level. For each vertex, we create a shingle to capture the hash values of the previous node, the hash value of itself, and its number of occurrence within one graph.

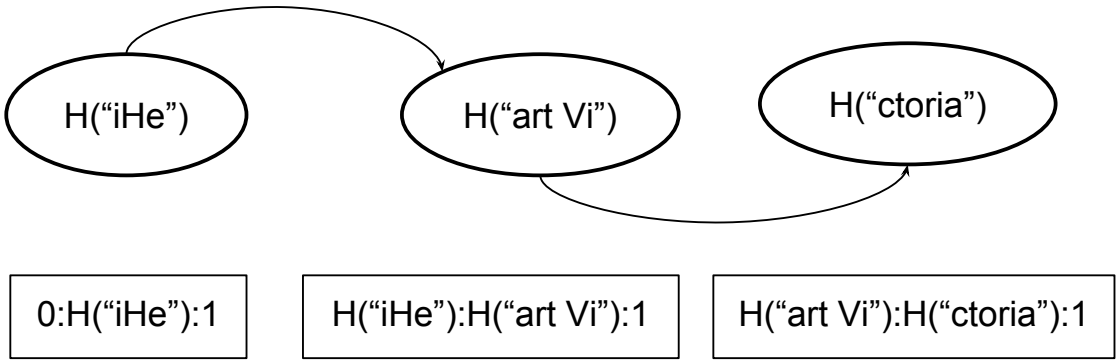

Figure 3·4: An example de Bruijn digraph of the level 1 partitions from the partition tree in Figure 3·3.

To create a hash shingle multiset, we insert all shingles into a set and label each of them with its level number. For example, Table 3.1 shows the hash shingle multiset for our example hash partition tree. We reconcile this multiset with the one on the other reconciling host to obtain a new multiset of shingles. In order to reconstruct the string from the other host, we need to exclude the shingles not known to the other host from our multiset.

3.2 Iterated Backtracking

The backtracking starts after both reconciling hosts possess the equivalent shingle multisets. To reconstruct a string, we need to backtrack the hash shingles from the lower partition level using an exhaustive method [48]. Our exhaustive backtracking method uses “brutal force” to trace through all possible Eulerian paths of a de Bruijn

0:H(“iHeart Victoria”):1:0	0:H(“iHe”):1:1	H(“iHe”):H(“art Vi”):1:1
H(“art Vi”):H(“ctoria”):1:1	0:H(“iH”):1:2	H(“iH”):H(“e”):1:2
0:H(“ar”):1:2	H(“ar”):H(“t ”):1:2	H(“t ”):H(“Vi”):1:2
0:H(“ct”):1:2	H(“ct”):H(“or”):1:2	H(“or”):H(“ia”):1:2

Table 3.1: An example of a multiset of hash shingles from the partition tree in Figure 3·3.

digraph in lexicographic order. As an example from Figure 3·3, if host Bob would like to help Alice to obtain the partition string “iHe” for hash value H(“iHe”), he has to first compute its *composition information* including the head partition hash H(“iH”), number of partitions 2, and Eulerian tracing number 1. Bob uses the *Depth First Search* algorithm starting from the head shingle 0:H(“iH”):1:1 and searches for the next potential partitions and increment the Eulerian tracing number every time he incorrectly traces to 2 shingles. After receiving the *composition information*, Alice can then trace through the shingles using the composition information from Bob. After Alice inquires all unknown terminal strings, she would be able to put back all partition strings from the bottom up.

3.3 The Main Protocol

We now describe the Recursive Content-Dependent Shingling (RCDS) protocol for hosts Alice and Bob who wish to reconcile their string σ_a and σ_b respectively.

Recursive Content-Dependent Shingling

1. Hosts Alice and Bob create their *hash partition* trees from σ_A and σ_B respectively using the same parameters and keep dictionaries for hash-to-string conversions in a key-val store.

2. Both Alice and Bob export their partition trees into two sets of shingles S_A and S_B respectively.

3. Using one of the set reconciliation protocols mentioned in Section 2.1, Alice extracts the symmetrical differences between the hash shingle sets.

4. Alice remaps her partition tree with all shingles known to Bob.

5. Alice extracts unknown hashes by comparing against her local dictionary, and requests information from Bob.

6. Bob responds to the hash values of *terminal strings* by sending the strings as literal data. For non-terminal hashes, Bob needs to use the exhaustive *Backtracking* algorithm to obtain the *composition information* and send them to Alice.

7. Alice computes hash values of the literal data and includes them to her dictionary while uses the composition information to reconstruct all other unknown partitions from the bottom up. The last string reconstruction at the top level should give Alice exactly σ_b.

8. If Bob wishes to obtain the string from Alice as well, he can go through the same procedure from step 5 at the same time as Alice.

Protocol 3.3.1: Stepping through the Recursive Content-Dependent Shingling protocol

3.4 Asymptotic Analysis

The RCDS protocol uses a content-dependent partition that relies on hash functions to extract string patterns. Therefore, the performance of our protocol is content-dependent by nature. In the following analysis, we refer to all steps from Protocol 3.3.1 (RCDS) reconciling two similar strings with d number of edit distance. We use L to denote the number of recursive partition levels, p to represent the maximum of number partitions at each recursion, h as the minimum inter-partition distance, and N to be the larger input string size of the two reconciling strings. Since we can not control the terminal string size directly, we denote the average terminal string size for the partition trees as T.

We describe the following analysis under the expected-case where we disregard the possibility of cascading effects from partition mismatch. The partition mismatch happens when some number of string edits inside a partition changes the partition end-point compared to its counterpart causing the next partition to start at a different position in the string, thereby causing a cascading effect. We will explain this concept with more detail in Section 3.5.2.

3.4.1 Communication Complexity

The RCDS requires $O(dL\alpha + dT)$ bytes of communication to reconcile two strings, where α is the overhead constant for the chosen set reconciliation protocol, d is the edit distance between the two reconciling strings, and T is the average terminal string size from the partition tree.

The main communication cost of the protocol consists of set reconciliation cost from Step 3 and transferring of terminal strings from Step 6 under a fixed 2-round communication. The hash and string composition information transferring from Steps

5 and 6 are strictly less than the set reconciliation cost and, thus, is not included in the communication complexity.

For the $O(dL\alpha)$ bytes of set reconciliation communication cost, both CPI and IBLT algorithms in 2.1 have their communication cost linearly dependent on the number of symmetric set differences with their individual constant which we included as α. Our protocol requires to reconcile $O(dL)$ number of symmetric set differences for the hash shingles. To prove this value, we assume only one edit is made in a string, the resulting partition tree would have $2L$ number of symmetrical differences. Differences are at most doubled in a symmetry. With an additional edit, if the edit is not within the same terminal string partition, we would have at most $4L - 2$ symmetric differences since the level 0 partition is already counted. In general, each partition level would have at most d number of differences for d edit distance between the reconciling strings and we count these differences for all levels in the partition tree as an upper-bound. We concatenate these partitions into shingles as described in Section 3.1.2 and at most double the number of differences.

At last, in Step 6, if we consider every edit distance is in a separate terminal string, Bob would then transfer at most $O(dL)$ composition information, and $O(dT)$ bytes of literal data to Alice. Since T is usually controlled to be a lot bigger than the size of fixed-length hash representation, therefore, bigger than the size of composition information, and L is usually a small value logarithmic to the input string length, the literal data transfer would dominate the communication cost in Step 6.

3.4.2 Time Complexity

The RCDS protocol has a time complexity of $O(N \lg(h)L + dL\gamma)$ plus the set reconciliation time, where γ is the exhaustive backtracking time reconstructing orders for partitions of each recursion.

In Step 1, the protocol requires each reconciling party to expend $O(N \lg(h)L)$ time to initialize a *hash partition* tree for a given string. We assume the hash computation takes constant time. For each level of a L-level partition tree, the host requires $O(N)$ time to compute content hash values and $O(N \lg(h))$ time to partition. Specifically, a host moves a window of size $2h+1$ through its hash value array of length $N-w+1$, where at each movement the host takes $O(\lg(h))$ time to compute the local minimum value inside the window and compare the minimum with the value at the center of the window. We achieve a $O(\lg(h))$ time by maintaining a balanced Binary-Search Tree data structure for storing all values within the window range while inserting the next incoming value and deleting the exiting value as the window moves.

Backtracking Time Complexity

In Steps 6 and 7, Alice and Bob perform the similar tasks of reconstructing strings from shingle set which takes $O(dL\gamma)$ time. The protocol reduces the exhaustive backtracking time to a small value, γ, in the expected case. With our exhaustive backtracking implementation, each string reconstruction would take time $O(p^D)$, where D is the maximum degree in the de Bruijn digraph and p is the maximum number of partitions allowed for each recursive partitioning. Using the Depth First Search, we require p number of traversal steps, and in every step, we can have up to D potential paths. The degree, D, corresponds to the largest number of occurrences of any partition within a partition level. Provided no duplicated partition exists, we would have the best case performance of $\gamma = p$. To avoid duplicated partitions, where two partitions in the same level have the exact same content, we set a lower bound on the *terminal* string size large enough to avoid reoccurring partitions. Empirically, our experiment shows that string reconstruction is not dominating the time cost.

In syntactic content such as programming scripts, D could be a large number due

to high recurrence of specific syntax substrings which poses a potential exponential time complexity to our algorithm. However, we can predict the computation feasibility of our protocol after constructing the partition tree by determining D. Besides, since our partition method is based on the Rabin-Karp algorithm [34] over string content hash values, we can change the hash function for an alternative partition tree that has fewer partition duplicates.

3.4.3 Space Complexity

The space complexity of the RCDS protocol is defined $O(p^L + dT)$ bytes, where we assume that a fixed number of bytes is representing a hash value of a string partition.

Our hash partition tree constructed in Step 1 has at most $\sum_{i=0}^{L-1} p^{(i)}$ number of partitions, where p is the maximum number of partitions in each partition recursion and L is the number of levels within the partition tree. In our partition tree, level 0 refers to the input string and each child node in the partition tree could have up to p number of children. We multiply the maximum number of nodes in a partition tree with the space required for every fixed-length hash values. Since the maximum number of partitions available in a tree is represented by a geometric sum, we simplify the form to get $\frac{p^{L+1}-1}{p-1}$, which could be represented by $O(p^L)$. We then transfer the partition tree into a multiset of shingles in Step 2, which at most doubles the amount of space required by the partition tree because each shingle is consist of two partition hashes.

Finally, in Step 6, after Bob sends all requested information for unknown hashes, Alice would have to save $O(dT)$ bytes of literal data for unknown terminal hashes in her dictionary.

3.5 Worst-Case Performance

When creating a partition tree, both partition mismatch and duplication can introduce a large amount of communication and computation cost. Partition mismatch can result from string edits or the cascading effect of partition mismatch from previous partitions or the upper-level partitions of the tree. As for partition duplication, while it saves some communication cost, the duplicated partitions increase string reconstruction time exponentially as described in Section 3.4.2.

3.5.1 Sparse String Edits

Since the RCDS protocol requires hosts to transfer all unmatched *terminal* strings, string edits are best to occur in one location resulting in the least number of unmatched terminal strings and internal partitions of a partition tree. If the string edits are sparse and spread across all terminal strings, we would have to send all terminal strings as unmatched data. If there is no duplicated terminal partition, then the number of bytes needed to transmit all unmatched terminal partitions would be precisely the size of the entire string, and our protocol would perform worse than the *full-sync* baseline protocol due to the partition tree overhead.

Moreover, the computation cost is also related to the string edit sparsity. For every unmatched non-terminal partition, we are required to compute its string *composition information* described in Section 3.2 and use it again on the other host to reconstruct the partition string. If edits are spread across the entire string, we would have to reconstruct all internal string partitions in the partition tree.

3.5.2 Partition Probability and Cascading Partition Mismatch

Our *local minimum chunking* technique guarantees the same partition cut-points given the same input string and cutting parameters. We define cut-points as the two end-points of a partition. As described in Section 3.1.2, the ideal case is that a small number of string edits do not change the partition position related to the string content, namely the partition with string edits will still have the same start and end points compared to its counterpart. The characteristic of partitioning at each recursion level depends on parameters including the minimum partition distance h and the content hash space s described in Section 3.1.2. In this analysis, we present the probability of local minimum chunking and describe the worst case of partition mismatch where two partitions share a content relative start point but end differently by the string edits in the effective area. The effect of partition mismatch can get carried over to its subsequent partitions causing cascading mismatches.

Partition Probability

We adopt an analysis for strict maximum chunking method from [5] to our non-strict minimum partitioning method. Our content-dependent chunking parameters include the rolling hash window size w, the space of content hash s and minimum partition distance h. The use of rolling hash converts w number of consecutive characters to a hash value. The array of hash values should occupy the entire hash space and avoid biased partition decisions from reoccurring string content. In any type of strings, we would create more unique hashes by hashing longer substrings, since it is more likely to see unique substrings inside a larger rolling hash window. Generally, we want $|\Sigma|^w >> s$, where $|\Sigma|$ is the size of string content alphabet, so that the intended space s of the content hash is filled.

The choices of h and s directly affect the probability of partition. Like we discussed in Section 3.1.1, the minimum partition distance h controls the upper bound number of partitions allowed at each recursion. Besides, h is also the window size for the local minimum chunking method where we look for h content hash values before and after an arbitrary point to decide whether if it is a potential cut-point. We partition at a potential cut-point if the location of the cut-point is h distance away from the last partition end point. In Lemma 3.5.1, we described the probability of an arbitrary point in a string to be a potential cut-point. If the probability of the cut-point is too high compared to the string size, we lose the uniqueness of each cut-point. In the worst-case scenario, where all content hashes have the same value making the probability of potential cut-point to be one hundred percent, our partition algorithm would be no different than a fixed-size partition method with its block size equal to h. On the other hand, if the probability is too small compared to the string size, we could have no partition at all, and, consequently, not reducing the communication cost for reconciling the strings. In general, we want $O(p)$ number of potential cut-points which is roughly equal to *string size * cut-point probability.*

Lemma 3.5.1. *The following is based on [5, Remark 57], given an independent and identically distributed (i.i.d.) number array A, where* $A[j] \in \{0, 1, \ldots, s-1\}$, $j = \{0, 1, \ldots, 2h\}$, *and* $s, h \in \mathbb{N}^+$. *For any* $k \in \{0, \ldots, 2h\}$, *the probability that* $A[k] \leq A[j]$ *for all* $k \neq j$ *is given by:*

$$p = \sum_{j=0}^{s} \frac{1}{s} \left(\frac{j}{s}\right)^{2h}. \tag{3.1}$$

Proof. We consider an arbitrary value $A[k]$ within the number array of $2h+1$ values to be a non-strict minimum value if all other numbers are equal or larger than said value. For example, if $A[k] = 0$ is a minimum value in an array of numbers in the

range of $[0, s-1]$ inclusive, then all $2h$ other values would have to choose from values that are 0 or above. The probability for all $2h$ other i.i.d. values being larger than or equal to 1 is $\left(\frac{s-0}{s}\right)^{2h}$, because all of these $2h$ values can be a value from 0 to $s-1$. Since there is an equal chance, $\frac{1}{s}$, for $A[k]$ to be any number between 0 to $s-1$ inclusive, we sum up all the probabilities for $A[k] \in \{0, \ldots, s-1\}$ while $2h$ other values being greater or equal to $A[k]$:

$$\frac{1}{s}\left(\frac{s-0}{s}\right)^{2h} + \frac{1}{s}\left(\frac{s-1}{s}\right)^{2h} + \cdots + \frac{1}{s}\left(\frac{s-s}{s}\right)^{2h}. \tag{3.2}$$

The last term in the Equation 3.2 is trivial and equals to zero since it is not possible to choose the value s in the range of 0 to $s-1$ inclusive.

□

According to Equation 3.1, we determine our potential cut-point probability with parameters h and s as shown in Figure 3·5. Since $h = \frac{N}{p}$ where N is the string size and p is the maximum number of partitions at each recursion, the only variable parameter is s, and we use it to control the number of potential cut-points in a string. For example, if we want to partition a string of 10^4 characters with $p = 10$, then h should be about 10^3. According to Figure 3·5, we would want to choose $s = 10^3$ so that the partition probability is 10^{-3}, and we could have about 10 potential cut-points. Since the probability is linear to h and s, we can fix a rate to decrease s and h as we move down to lower recursive partition levels. We use the same s and h for all partitions in a recursion level to avoid parameter mismatch. We will generally get the same number of partitions at each recursion as we move down the partition tree. In our experiments, we fix the rate of reducing s and h to p and control the maximum number partitions of each recursion at p.

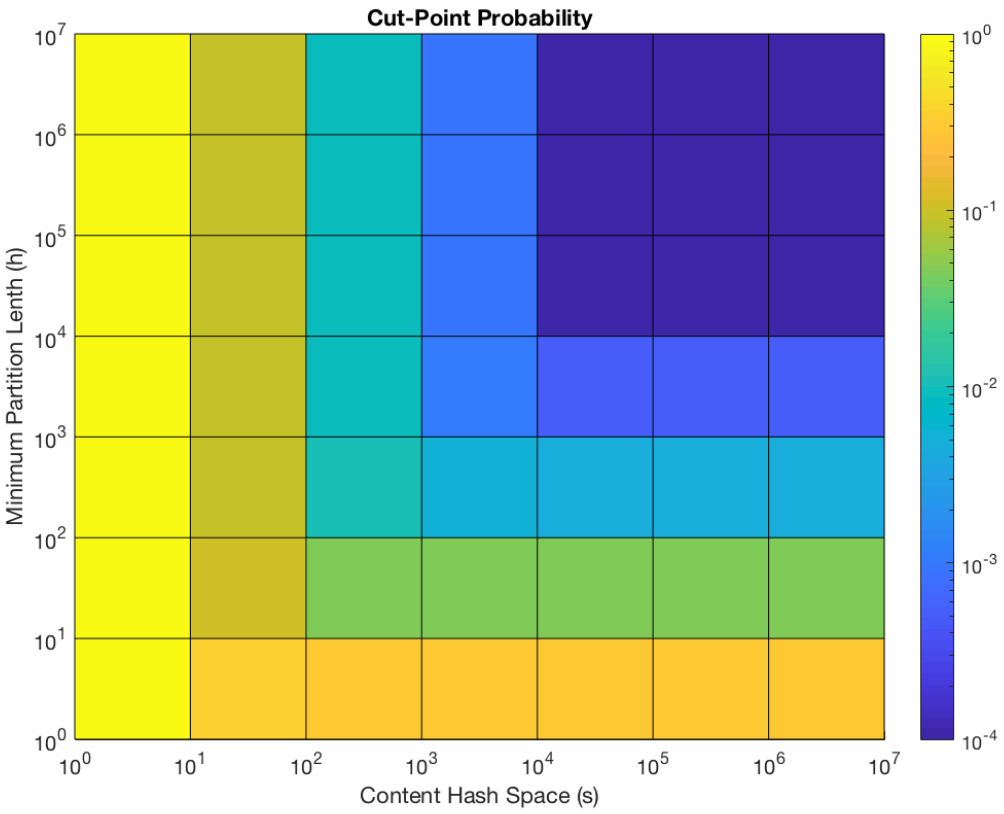

Figure 3·5: Potential cut-point probability based on Lemma 3.5.1.

Cascading Partition Mismatch

Like we discussed earlier, a mismatched partition can cause its next adjacent partition to mismatch as well. Moreover, it is possible that the second next partition is also affected and so on. Partition mismatch is inflicted by edits within h positions away from a cut-point. In addition, string edit distance does not directly translate to the differences in the content hash values.

Lemma 3.5.2. *Given d amount of edit distance between the reconciling strings, we could have $O(d * w)$ number of differences between the content hash arrays, where w is the rolling window size from Section 3.1.2.*

We consider each insertion, deletion, and replacement as one edit. Using a rolling

window of size w, we create a string content hash array by moving the window from the start of the string to the end as described in Section 3.1.1. Each hash value is created by hashing w number of continuous string characters. For each edit, there would be at the most w number of different values at the edit location. If we are deleting or inserting a string character, we inherently are truncating or extending the size of a content hash array by 1 respectively. For replacement edit, there would be at most w different values at the replacement position without changing the length of the string or its content hash array. Given d amount of edit distance between the two reconciling strings, we would have $O(d * w)$ number of different content hashes at the edit location. The new values that could affect the partition decisions would either be a new strictly minimum value or the same value as the original minimum value residing on the left side of the old cutting point.

3.6 Failure Modes

The RCDS protocol can recover a string from a correct partition tree deterministically using an exhaustive *Backtracking* method. The only possible causes of failure are hash collisions and set reconciliation failures.

Theorem 3.6.1. *The upper-bound probability of failure using CPISync [41] as the set reconciliation protocol is:*

$$\left[1 - exp\left(- \frac{(n+1)^2}{2(2^{64}+1-n)}\right)\right] + m\left(\frac{n-1}{2^{64}}\right)^k \tag{3.3}$$

where n is the maximum total number of partitions created by both reconciling hosts, m is the total number of symmetric shingle differences, and k is the CPISync redundancy factor.

Proof. The first term in Equation 3.3 is the hash collision rate for fixed-length hash functions and the second term is the failure rate of the CPI set reconciliation. We can swap these terms accordingly if we change the underlying hash function or the set reconciliation protocol. Generally the hash collision rate is a function of unique input collection size and the set reconciliation failure rate is a function depending on total number of set elements and symmetric differences between the reconciling sets. In the case of reconciling strings σ_A and σ_B on Hosts Alice and Bob, the RCDS protocol uses set reconciliation to reconcile differences between the two hash shingle multisets S_A and S_B. The hash shingle multisets are shingles consist of every two adjacent partition hashes in each partition recursion. We label their recursion levels as described in Section 3.1.2 to put them into one multiset and extract set differences using a set reconciliation protocol. Therefore, the total number of set elements of our multiset is at most the total number of partitions from both reconciling hosts and the number of symmetric differences is the number of different hash shingles between the two multisets. We use the maximum total number of partitions from both reconciling parties, n, without considering the uniqueness of these partitions to form an upper-bound. □

3.6.1 Collision Probability

During the process of string reconciliation between two hosts, any two different substrings that create the same hash value on either the same or different hosts would lead to a fatal collision. For universal fixed-length hash functions, the collision rate Q given n number of inputs can be estimated by the following inequality [46]:

$$Q < 1 - exp\left(- \frac{(n+1)^2}{2(2^b + 1 - n)} \right) \tag{3.4}$$

where $b \geq 3$ is the length of the hash values in bits, while $n < 2^{b-1}$. Our protocol produces hash inputs from all partitions in a partition tree. We control the partition tree size which is the number of nodes in a partition tree by fixing the maximum number of partitioning, p, at each recursion and the number of recursion levels L. Therefore, the maximum number of nodes in a partition tree is characterized by $\frac{p^{L+1}-1}{p-1}$, where $L, p \in \mathbb{N}^+$. We plot Figure 3·6 to give the upper-bound collision rate using $b = 64$ and $n = 2 * \frac{p^{L+1}-1}{p-1}$, because we are using a 64-bit fixed-length hash function and the same partition tree parameters to partition the two reconciling strings.

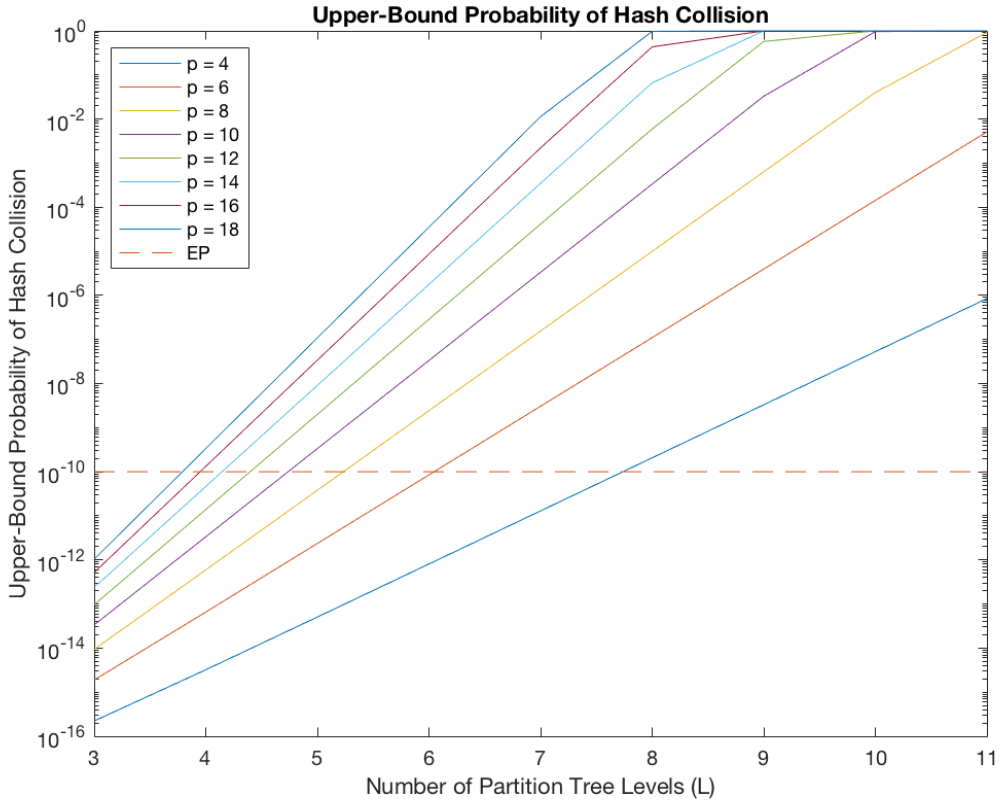

Figure 3·6: Upper-bound probability of hash collisions vs. number of recursion levels as a function of maximum number of partitions at each recursion p.

In Figure 3·6, we set the effective probability (EP) at 10^{-10} and consider partition tree configurations below EP to be effective. For example, if the partition tree is configured such that $p = 4$ and $L = 7$, the probability that we will see a hash collision is less than about 10^{-11}. To lower the probability of collision given the partition tree parameters, we could increase the space of the hash values by introducing additional hash functions or substitute for a longer fixed-length hash. We could also select from more collision-resistant hash functions. Empirically, our experiments show that the average number of partitions yield from every recursion in a partition tree is usually half of the configured value p. Therefore, we have not seen a collision event in our experiments. However, the average number of partitions from all partition recursions is content-dependent. This analysis gives a loose upper-bound on the hash collision probability, which means, in practice, our protocol is much less likely to suffer from hash collision failures.

3.6.2 Reconciliation Failures

The set reconciliation failure rate varies from one algorithm to another, but generally considers the reconciling set size $|S_A|$ and $|S_B|$ and the number of set symmetric differences m. In Table 3.2, we list the upper-bound probability of failure for some algorithms mentioned in Section 2.1. All algorithms mentioned here can lower their probability of failure by increasing their versions of redundancy. For IBLT-based set reconciliation, we can increase the table size and the number of hash functions, whereas for CPI-based set reconciliation, we can increase the number of additional evaluation points.

CPI Sync [41]	$m\rho^k$
Interactive CPI [42]	ρ^k
IBLT [26]	$O(m^{-a+2})$

where:
m = Number of symmetric set difference
$\rho = (|S_A| + |S_B| - 1)/2^b$
$|S_A|+|S_B|$ = Combined size of the reconciling sets
b = Size of a set element in bits
k = Additional evaluation points
a = Number of hash functions

Table 3.2: Set reconciliation fail rate.

Chapter 4

Evaluation

We implemented[1] RCDS protocol as described in Chapter 3.3 and tested against random strings and random collections of books from Project Gutenberg [29] and programming scripts from Github[2]. The two reconciling hosts are organized as parent and child processes through which they communicate via a socket on one piece of commodity hardware[3]. To control the edit distance between the two reconciling strings, we generate a similar string by randomly inserting or deleting substrings from parts of the original copy with a controlled overall upper-bound on the *edit distance*. We use an *edit burst* at each insertion or deletion by picking edit length uniformly at random to mimic the human edits. The edit length is also counted towards the overall upper-bound *edit distance*. In the following graphs, for every point, we use 95% confidence interval among 1000 observations.

Our evaluation examines the communication, space, and time complexities described in Section 3.4. As a support for the worst case analysis, we also include empirical results for the percentage of partition mismatches to demonstrate the effectiveness of recursive content-dependent partitioning. We count the partition differences resulting from both string edits and their cascading effects. We present the

[1]The implementation is available at `https://github.com/Bowenislandsong/cpisync.git`.

[2]All testing data set is available in a public github repository at `https://github.com/Bowenislandsong/sync_database.git`.

[3]iMac 2012 with 2.9GHz quad-core Intel Core i5 processor and 8GB of 1600MHz DDR3 memory.

communication cost as a percentage over the size of the original input string to show the effectiveness of our protocol, since the baseline protocol is to send the entire string data over using *full-sync*. The communication cost includes the total number of bytes transmitted and received on a reconciling host, including the cost of set reconciliation. For individual time analysis, we separated the time cost for string reconstruction, set reconciliation, and partition tree construction arranged in our figures from bottom to top respectively. The region between set reconciliation time and total time implies the partition tree construction time.

In this chapter, we compare our protocol against the following:

- Section 4.1 – Different tuning parameters for fixed inputs.
- Section 4.2 – Different string inputs for fixed synchronization parameters.
- Section 4.3 – Comparison with the *rsync* utility [12].

4.1 Varying Protocol Parameters

To provide a general performance and find the best parameters for detailed analysis in Section 4.2, we randomly select strings from our data set to accommodate a general experimental result for different types of strings and compare it with our analytic analysis in Section 3.4. By fixing the reconciling string size to $1 * 10^6$ characters and creating a similarly sized copy with an upper-bound of $1 * 10^3$ edit burst distance, we observe the performance trend by changing the partition tree parameters.

From the graph of communication percentage cost (Figure 4·1), we see a sharp decline for the amount of literal data transferring as the size of the partition tree grows with respect to the growth of maximum number of partitions and the number of recursion levels. Similarly, for *mismatched partitions*, Figure 4·2 shows the percentage

of partition difference dropping dramatically as the size of the partition tree grows. In other words, most of the partitions match with their counterparts and reduces the amount of literal data needed to be transferred. Combining the two graphs, we see the result of partition hierarchy successfully isolating string edits while matching unchanged substrings to lower the communication cost.

Unfortunately, we see a tail raise in Figure 4·1 for the total communication cost as the partition tree grows, leaving the middle ground as the optimal area. The optimal area is where the number of unmatched partitions combining with set reconciliation overhead contributing to the communication cost is not significant enough. Besides, we see a steady gap increase between the total communication cost and literal data which describes the communication cost for set reconciliation. This increase in set reconciliation communication cost is caused by having to resolve more partition differences. Even though Figure 4·2 shows that the percentage of unmatched partition is dropping as the partition tree grows, the actual number of unmatched partitions, shown in Figure 4·3, is raising, therefore, increasing the communication cost for set reconciliation.

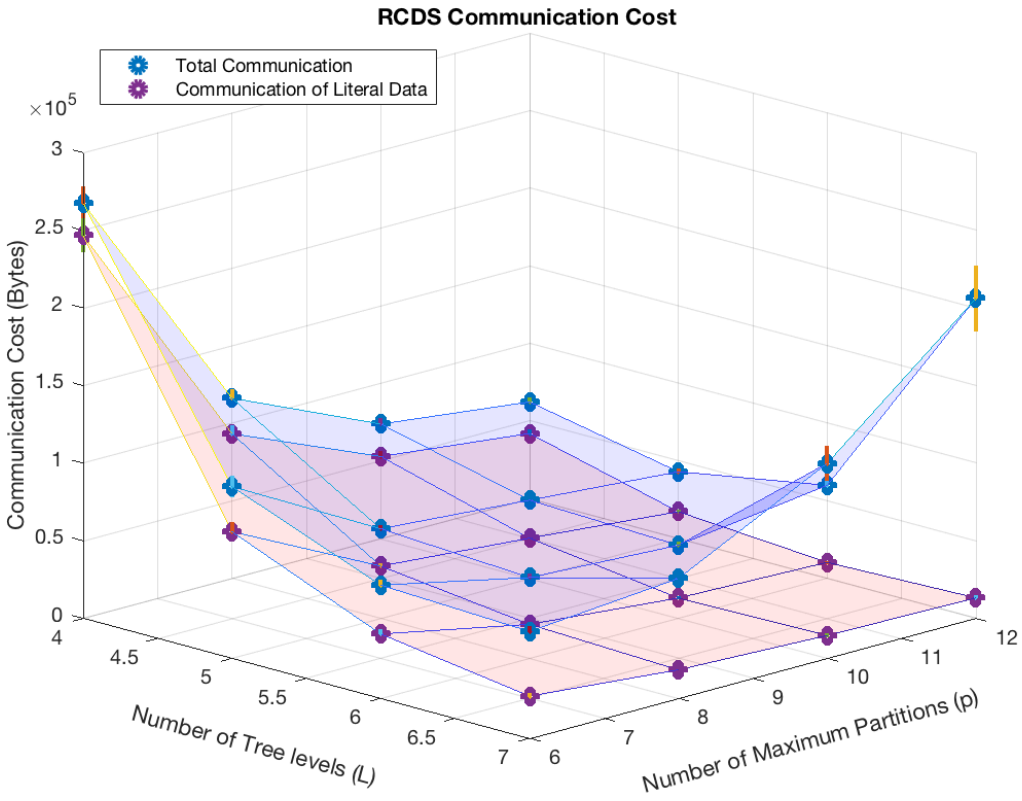

Figure 4·1: Stack plot of internal operations in RCDS comparing communication cost for different partition tree parameters. The operations in the stack plot include CPI-Based set reconciliation and communication of literal data arranged from top to bottom as regions between the surfaces. The region between total communication and communication of literal data is the communication cost of set reconciliation. The literal data are strings of the unmatched terminal partitions.

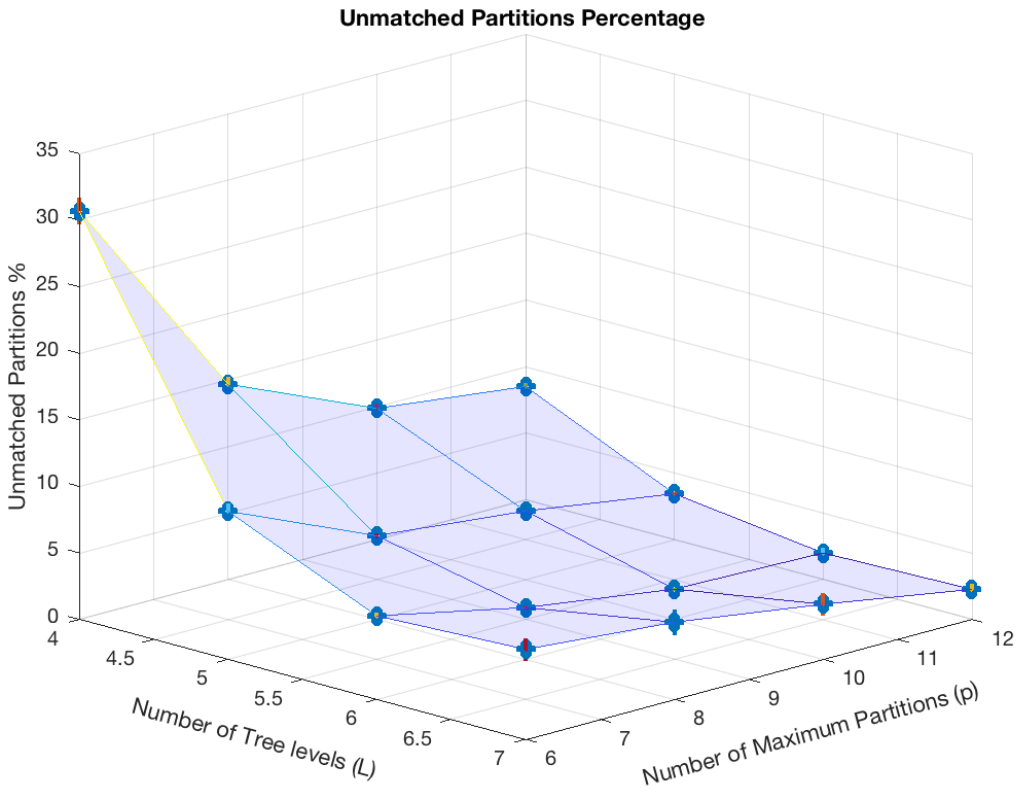

Figure 4·2: Comparing the percentage of unmatched tree partitions vs. the total number of partitions for different partition tree parameters. The percentage of unmatched tree partitions decreases as the partition tree grows which shows the effective usage of partition hierarchy.

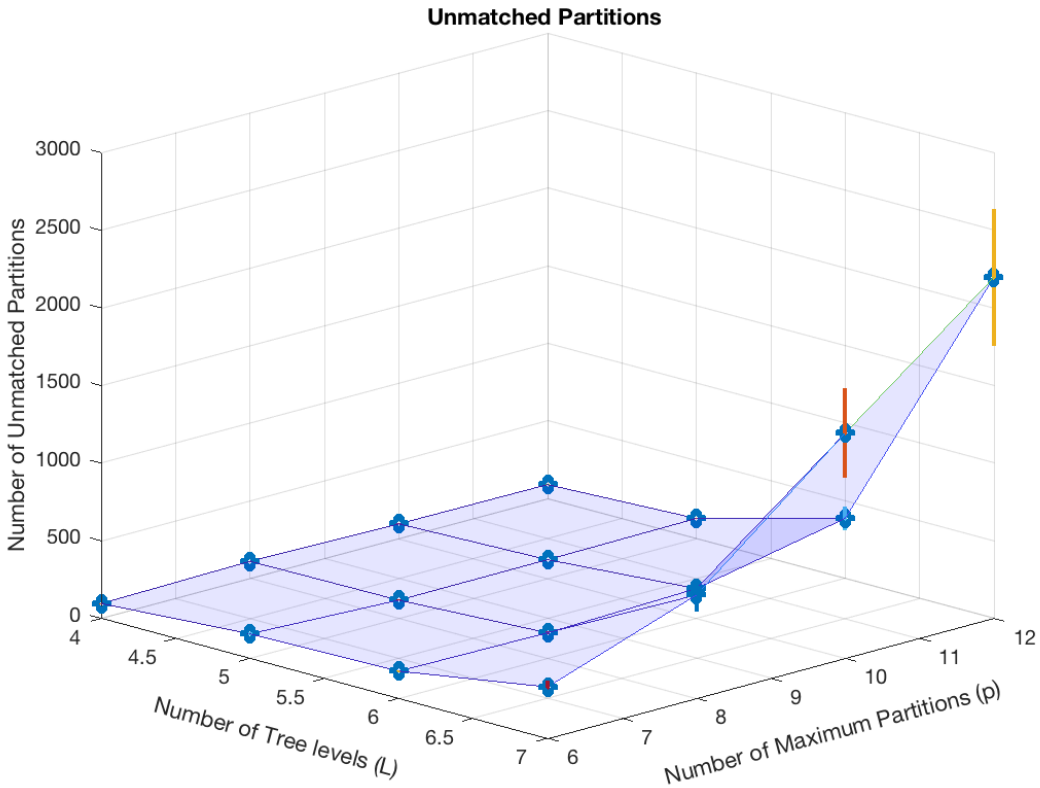

Figure 4·3: Comparing the total number of unmatched tree partitions for different partition tree parameters. The number of unmatched partitions increases as the partition tree grows and exceeds the fixed 1000 edit burst distance. The extra unmatched partitions are from the partition tree hierarchy which is reflected by set reconciliation communication cost rather than the cascading effects of mismatch partitions. This is confirmed by the communication costs plot in Figure 4·1 where the set reconciliation cost increases as the partition tree grows while the amount of literal data transferred decreases.

In the time complexity graph (Figure 4·4), the string reconstruction time at the bottom is having the least amount of increase with the same trend as the other operations thanks to the small amount of overhead. The number of different partitions is bounded by the fixed number of edit distance. The backtracking time ,γ, as described in Section 3.2, is small enough to not dominate the total reconciliation time. On the other hand, the set reconciliation time shows a sensitive increase to the increasing number of set elements and symmetric differences. At last, the partition tree construction time, shown as the gap between the set reconciliation time and total time, follows a log-linear increase over the number of recursive levels while staying constantly parallel to the number of partition changes which agrees with our expectation of $O(N \lg(h) L)$.

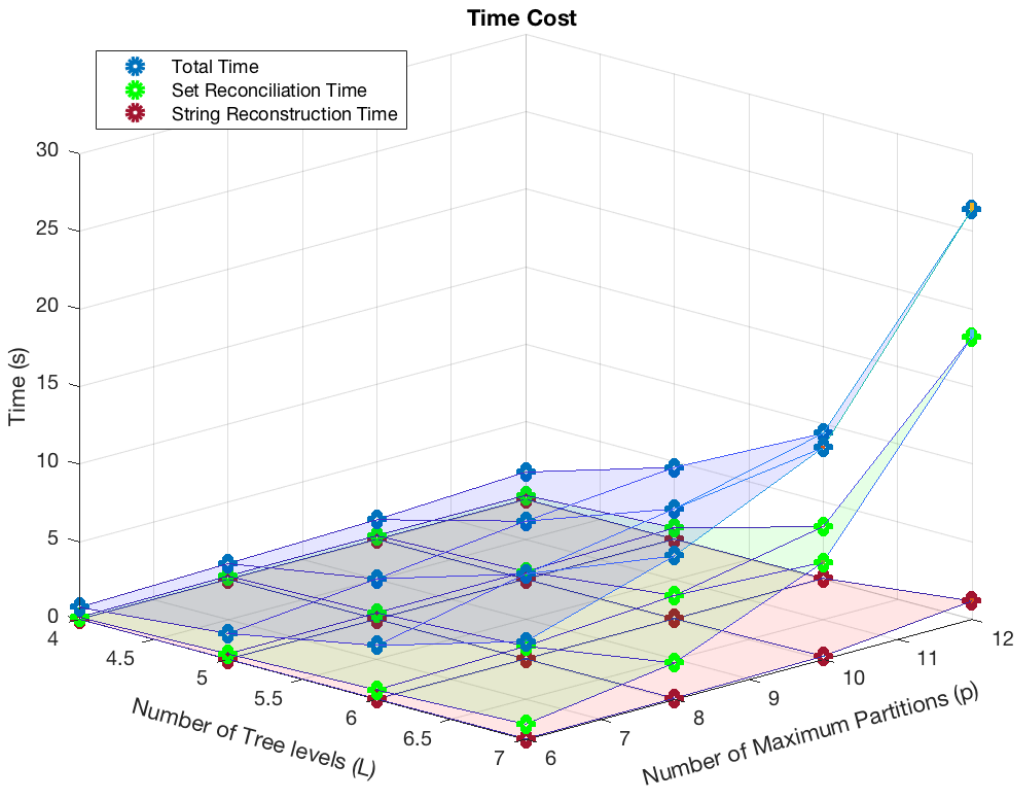

Figure 4·4: Stack plot of internal operations in RCDS comparing time cost for different partition tree parameters. The operations in the stack plot include partition tree construction, CPI-Based set reconciliation, and string reconstruction arranged from top to bottom as regions between the surfaces. The top surface corresponds to the total string reconciliation time cost.

The space complexity in Figure 4·5 shows a heap size trending that follows the increase of the partition tree size. We use percentage of space occupation comparing to the string size to show the effectiveness of space consumption. The majority of the space is occupied by the partition tree, hash-to-string dictionary, and hash shingle multiset. We are not counting the space required for accepting literal data transferred after set reconciliation since we know the worst-case scenario is when the entire string is transferred as literal data. The hash-to-string dictionary is implemented such that

each hash values is referring to the absolute location index and its string length pointing at the original string. For example, in the first level of partition, as we partition the original string into a few chunks, if the first partition is starting from the head of the original string with a length of 1000, we register this partition in to the dictionary as *[hash value of the first partition:0,1000]*. The partition tree requires to store $\frac{p^{L+1}-1}{p-1}$ number of hash values, and hash shingle multiset requires at most double the size of partition tree since we are concatenating every pair of adjacent partitions in each partition recursion.

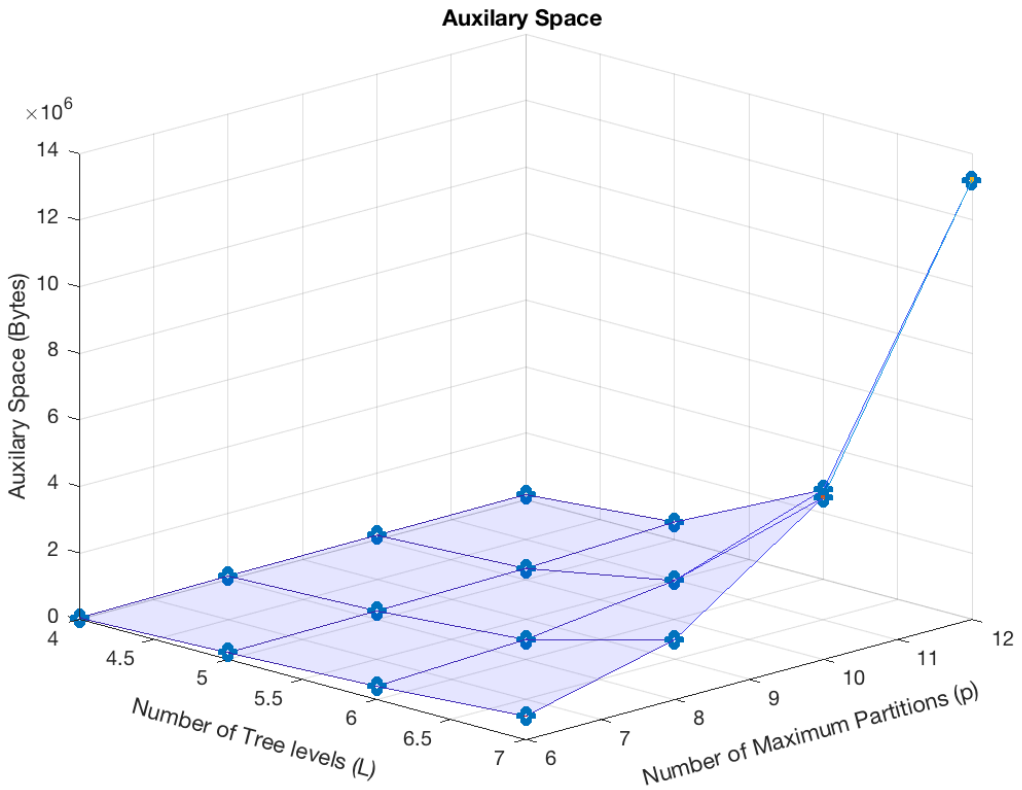

Figure 4·5: Comparing space complexity for different partition tree parameters. The space requirement increases as the partition tree grows in size with respect to the number of maximum partitions and recursion levels.

By combining all these performances, we observe the optimal performance is when the level of recursions $L = 4, 5$ and maximum number of partitions $p = 8, 9$ to reconcile $1 * 10^6$ character strings. We highlight the importance of tree parameter differences in the next section by varying the size of input string around $1 * 10^6$ characters and the edit distance between the input strings while fixing $L = 4, 5$ and $p = 8$.

4.2 Varying String Inputs

We compare the reconciliation performance of RCDS protocol under different sized partition trees by changing the reconciling string size in the range of $[10^5,\ 2 * 10^6]$ characters and edit burst distance between $[10^{-3},\ 100]\%$ of the string size. We fix the number of maximum partitions at each recursion $p = 8$ and levels of partition recursion $L = 4, 5$.

The communication cost percentage in Figure 4·6 shows the amount of edit distance the protocol can handle at different string length. For shorter strings, the protocol introduces a relatively large amount of overhead from set reconciliation and hash representations. The overhead becomes less significant as the size of reconciling string increases. As expected, the amount of communication also increases with the edit distance.

Given two fixed tree sizes (i.e., fixed upper-bound on the number of tree nodes), Figure 4·7 shows the percentage of unmatched partitions varying string size and edit burst distance. The general trend of increasing difference as the edit burst distance grows, while staying constant along string length increase, suggests an overall sub-linear performance with respect to the input string size. We see the 4-level partition tree gets saturated after the 10% edit burst distance while the 5-level partition tree still has room to grow. We combine this observation with Figure 4·6 to show that the

extra partition level helped to break partitions further, and created more matching terminal partitions to reduce the communication cost.

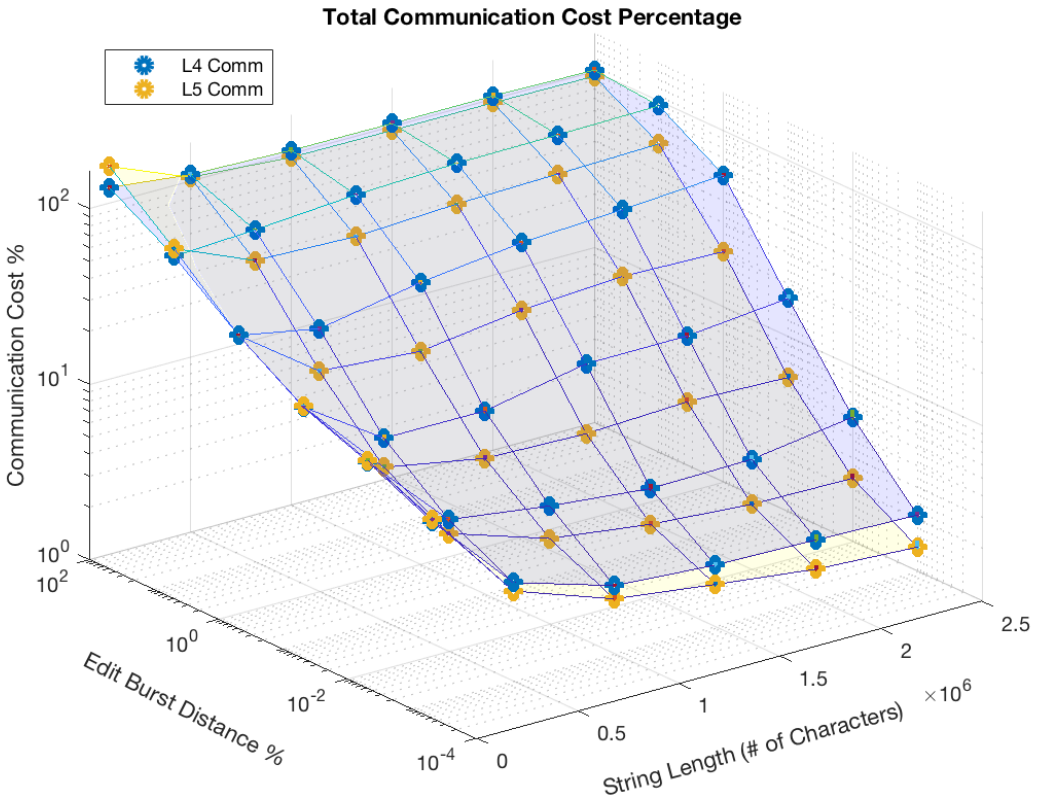

Figure 4·6: Communication performance reconciling strings of different sizes and edit burst distances using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The performance is measured by the percentage of the total number of bytes communicated during reconciliation vs. input string size in bytes. Using partition trees with a higher number of partition levels adds more overhead to communication cost when input string size is small but is more scalable to larger input strings.

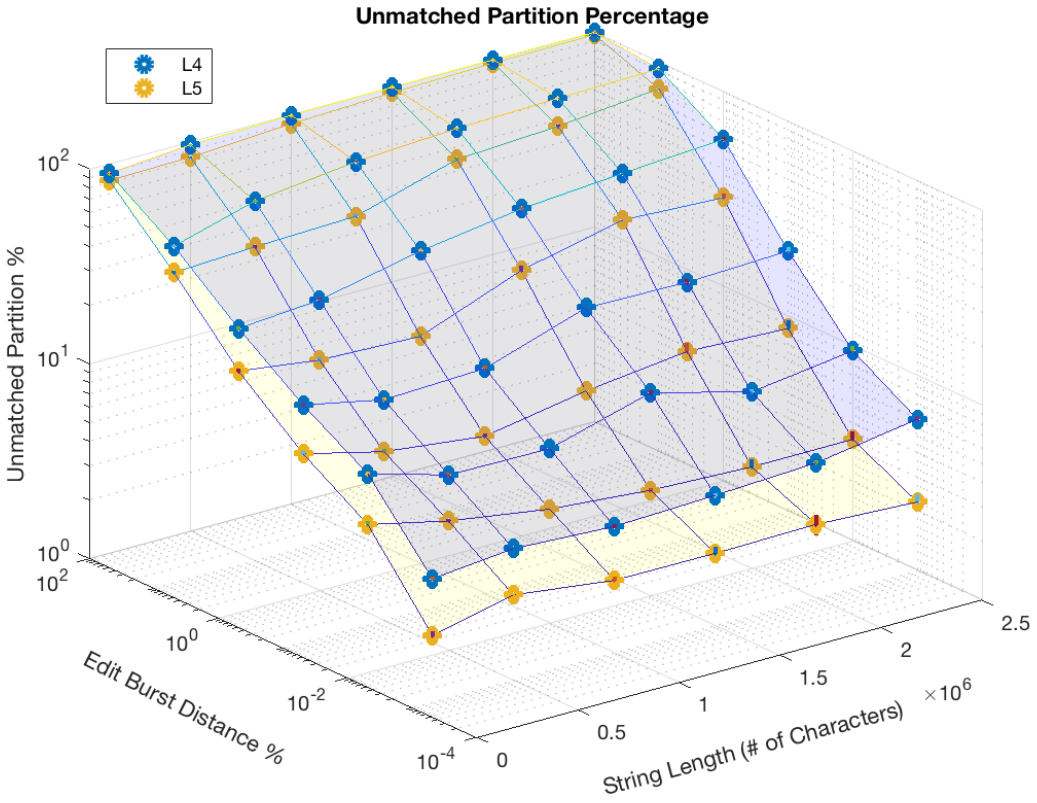

Figure 4·7: Percentage of unmatched partitions vs. the total number of partitions reconciling strings of different size and edit burst distance using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The unmatched partition percentage of L5 partition tree is much lower than that of the L4 partition tree when reconciling the same strings. As the input string length increases, the percentage of unmatched partitions stays almost constant while increasing in a steady trend as the edits burst distance grows. The number of unmatched partitions is proportional to edit distance, referred as d in the analysis from Section 3.4.

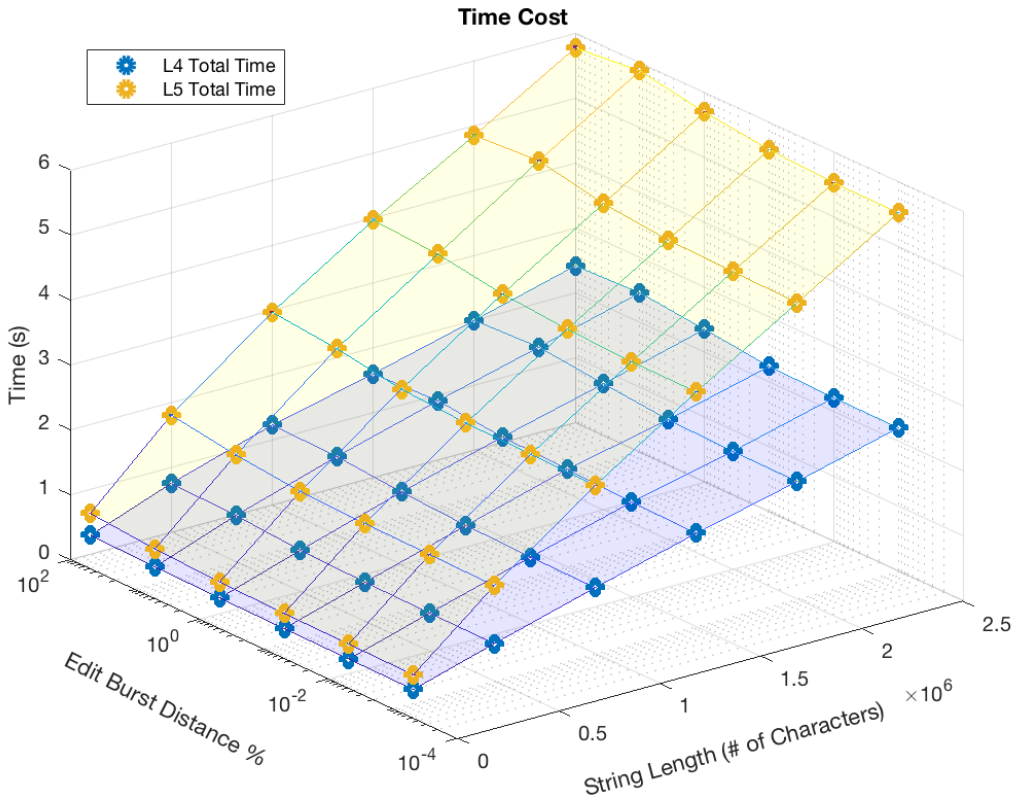

Figure 4·8: Total time cost reconciling files of different size and edit burst distance using L=4 and L=5 level partition trees with a maximum of p=4 partitions. The time used to reconcile different strings using L4 and L5 partition trees increase linearly to the input string size while staying almost constant to increasing edit burst distance. Reconciling strings using L5 partition trees costs more time than that of L4 due to the extra level of recursive partition creating more partitions for a given string.

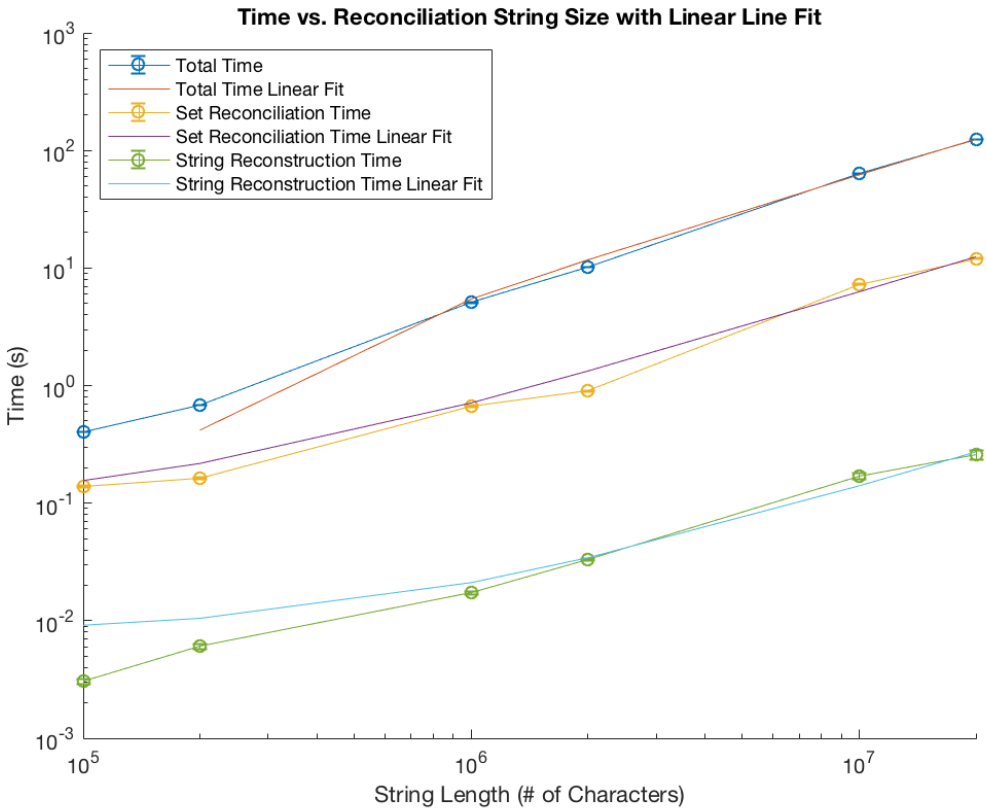

Figure 4·9: Time cost of every internal operation of RCDS in a stack plot including partition tree construction, CPI-Based set reconciliation, and string reconstruction for reconciling different length strings with 1000 edit burst distance. Each curve is fitted with Linear polynomial curve model with the goodness of fit presented in Table 4.1. The time cost corresponds to the linear time complexity analysis described in Section 3.4.

Goodness of Fit	Total	Set Reconciliation	String Reconstruction
Sum of Squares Due to Error	2.739	0.8379	0.0001
R-Square	0.9995	0.9893	0.9969
Root Mean Squared Error	0.8275	0.4577	0.0045

Table 4.1: The goodness of fit of the linear polynomial curve model for the time cost of every operation arranged from top to bottom in Figure 4·9 listed from left to right.

In our implementation, the time cost lies heavily on partition tree construction. We can reduce this time cost by the number of threads available, using parallel sorting algorithms mentioned in [55]. As expected, the tree construction time in Figure 4·8 is solely dependent on the size of the input string. It also shows that the string reconstruction time change is too small to show on the graph due to the small chance of partition duplication. Since each hash value stands for a relatively long string, an exact duplication of large string partition is less likely. However, such a chance could dramatically increase in certain types of content.

We enlarge the range for input string and fix the edit burst distance at 1000 to elaborate on the time cost for each operation. Figure 4·9 shows the time cost and its linear fit for constructing partition tree, reconciling set differences, and reconstructing string from hash shingles, respectively stacked from top to bottom. The gap between total time and set reconciliation time is the time cost for constructing partition tree. We also present the goodness of fit for each operation in Table 4.1.

The space complexity in Figure 4·10 under a fixed partition tree size is also dependent the reconciling string size and saturates after a certain point. We see that the 5-level partition tree is, as expected, more capable of handling longer strings and gets saturated much later than the 4-level tree. This is because the number of partitions

at each recursion is less likely to approach the maximum number of allowed partitions if the string is small. By fixing the probability of partition, a longer string is more likely to have more partitions than a shorter string. Since the space cost is directly related to the partition tree size, a partition tree with less number of nodes would require less space.

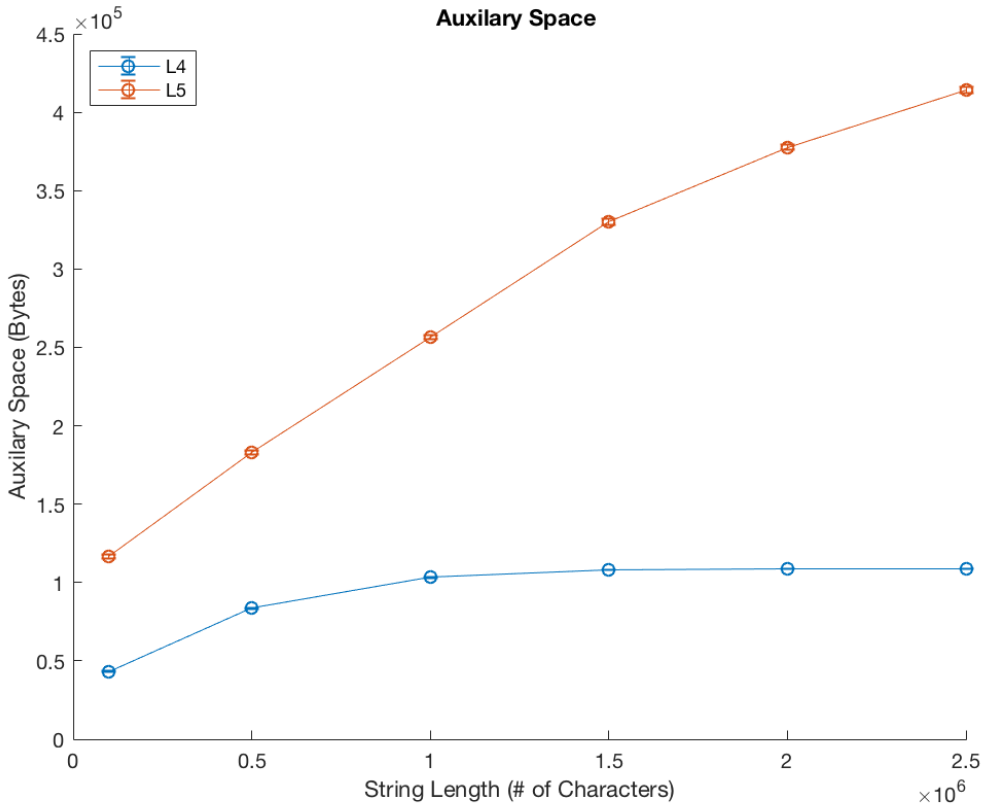

Figure 4·10: Space complexity reconciling strings of different size using L=4 and L=5 level partition trees with a maximum of p=4 partitions.An L5 partition tree can handle longer strings than an L4 partition tree by creating more partitions in the extra level and find more common substrings between the two reconciling strings to save communication cost. However, an L5 partition tree also requires more hash values to represent the extra partitions, thereby, costing more space.

4.3 Comparison to Existing Work

We compare the performance of RCDS protocol to that of the *rsync* utility[4] by synchronizing arbitrary local files and Git repositories using the same hardware as the previous experiments. The goals are, first, to provide a general idea of the protocol performance compared to *rsync*, synchronizing various single files, and, second, to give a sense of our performance for synchronizing git repositories, which is one of the applications mentioned in our motivation.

4.3.1 Synchronizing Single Files

For single files, we show a comparison of the communication cost synchronizing text files of different length under different random edit burst distances. Shown in Figure 4·11, *rsync* has a communication cost linear to the reconciling string size, due to the use of the rolling hash technique. In contrast, the RCDS protocol communicates hash shingles of symmetrical differences with some redundancies which makes the total communication cost proportional to the string edit distance. The communication cost is, therefore, staying sub-linear to the increasing reconciling string size.

The Figure 4·12 , shows the plane intersection of the total communication cost for *rsync* and RCDS protocol on a 2-dimension graph with respect to the equation:

$$y = 1.71x - 17.17, \tag{4.1}$$

where y is the edit burst distance and x is the input string size. Below the line of plane intersection on the 2-d graph, is where RCDS protocol performs better than *rsync* in terms of accumulative bandwidth consumption. We use a linear polynomial

[4]rsync version 2.6.9 protocol version 29 Copyright (C) 1996-2006 by Andrew Tridgell, Wayne Davison, and others. http://rsync.samba.org Capabilities: 64-bit files, socketpairs, hard links, symlinks, batchfiles, inplace, IPv6, 64-bit system inums, 64-bit internal inums.

surface model to fit the planes of communication cost and present the goodness of fit in Table 4.2.

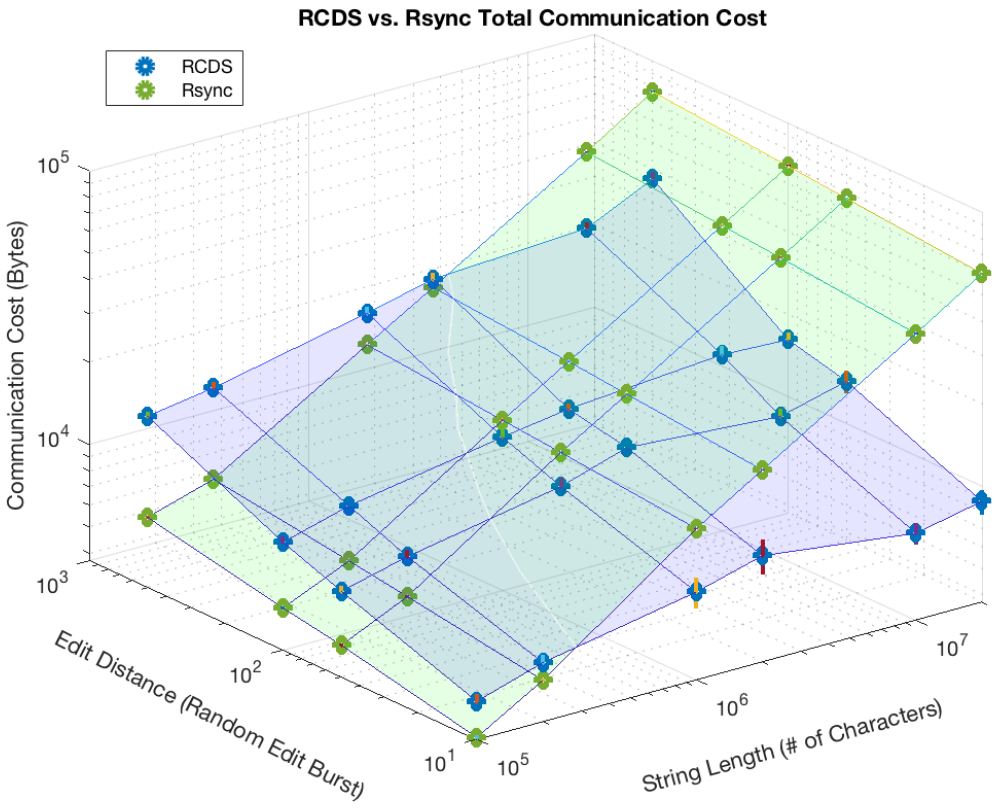

Figure 4·11: Comparing RCDS and *rsync* on the total number of bytes transmitted reconciling different text files. The green surface represents communication cost for the *rsync* utility which grows linearly to input string length whereas the blue surface representing the communication cost of RCDS remains sub-linear to input string length increase. Both RCDS and *rsync* require more communication cost to reconcile strings with more substantial edit burst distance, and RCDS appears to have a larger overhead.

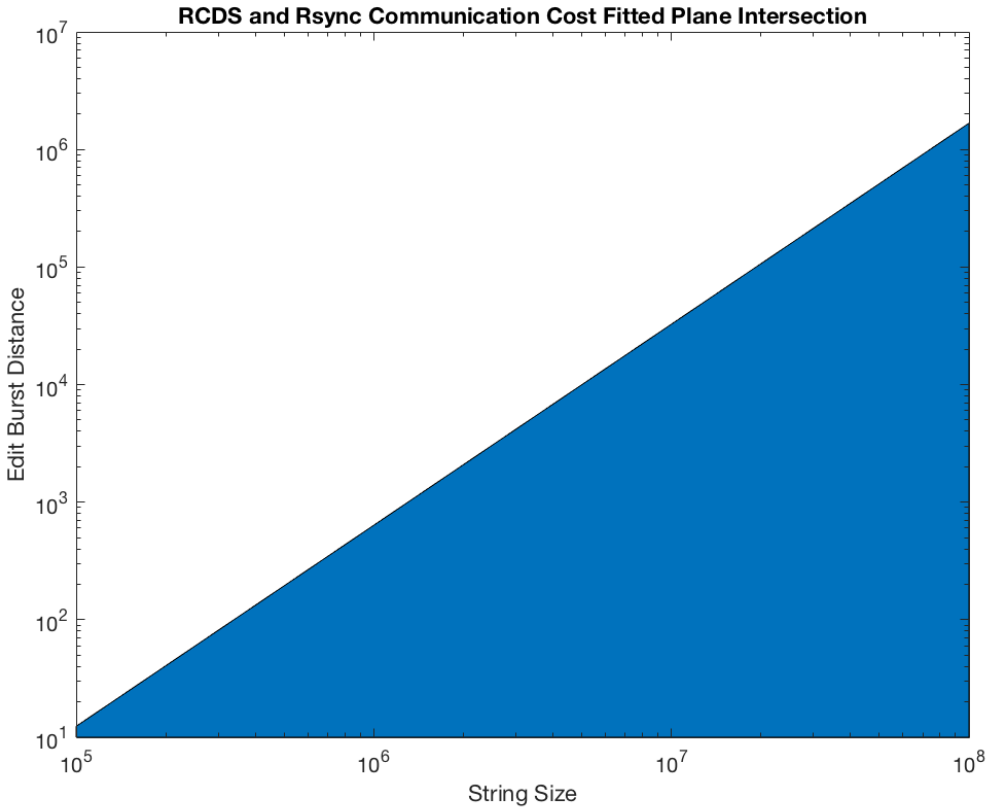

Figure 4·12: Line of intersection of linearly polynomial fitted planes from Figure 4·11 projected to a 2-D plot of edit burst distance vs. input string length. The blue area defined by the line of intersection and x-axis describes the circumstances where RCDS performs better than *rsync* in terms of communication cost.

Goodness of Fit	RCDS	rsync
Sum of Squares Due to Error	0.2084	0.0626
R-Square	0.9645	0.9971
Root Mean Squared Error	0.0996	0.0546

Table 4.2: The goodness of fit of the linear polynomial surface model for planes in Figure 4·11 comparing total communication cost of RCDS and *rsync* reconciling various input strings.

4.3.2 Synchronizing Folders

Gitlab [25] is one of the widely used DevOps lifecycle tools that mostly manages git repositories among other use cases. Gitlab uses *rsync* to support repository migration, synchronization, and backup services which manage between different versions and states of repositories. We collect some popular repositories and present the performance characteristics of our protocol compared to that of the *rsync* utility [12]. We utilize a basic heuristic to accommodate folder synchronization by checking file name and file size to determines if a file is different. We also check the list of files that RCDS considers different with that of *rsync* to ensure a fair comparison.

In Figures 4·13 and 4·14, we present the communication and time cost synchronizing the second latest to the latest release version of publicly available repositories, as of April 17th, 2019. Table 4.3 shows the name and the exact release versions of the repositories that are used in our experiments. In most cases, RCDS transmits less data than *rsync* reconciling repositories between releases.

Figure 4·13 shows the communication cost of synchronizing different releases of repositories, arranged in alphabetical orders by their repository names. The experiments reveal the communication performance of RCDS synchronizing data from real human edits. We compare and use the performance of *rsync* as a baseline. The repositories have varying numbers of files that are also different in size. The edits between the latest and the second latest release are different from one repository to another. If the new release introduces a new file, the file will be transferred in its entirety. In general, when the total difference is small compared to the repository size, RCDS performs much better than that of *rsync*, transferring much fewer bytes to reconcile between releases. However, some repositories, such as *Ansible* and *kubernetes*, have a considerably large number of differences between their releases, which require *rsync*

to transfer a large amount of data comparable to the size of the entire repository. In this case, RCDS performs even worse than *rsync* because the number of difference is significant.

There are some repositories that require much less communication cost from *rsync* than RCDS, such as *Electron*, *go-github*, and *react*. This could be because that the number of changed files is small, but the amount of changes to individual files is significant.

The time cost comparison shown in Figure 4·14 gives the total repository synchronization time of RCDS compared to the *rsync* utility. Since most of the files in a git repository are at most 100 megabytes (a rule imposed by GitHub), each file synchronization time is capped at some constant for both protocols. The general trend is that the time cost of synchronization is related to the collective edit distance of all files, between different release versions. Interestingly, some repositories such as *ansible* and *electron* which have fewer differences between releases, require more amount of time to synchronize than repositories such as *kubernetes* under both RCDS and *rsync*. This discrepancy could be because the *kubernetes* repository has a smaller average size of edited files or a more substantial amount of newly created files which are transferred directly. Nevertheless, we see the performance of *rsync* exhibiting the similar trend as RCDS in that the time cost is relevant to the amount of difference between releases.

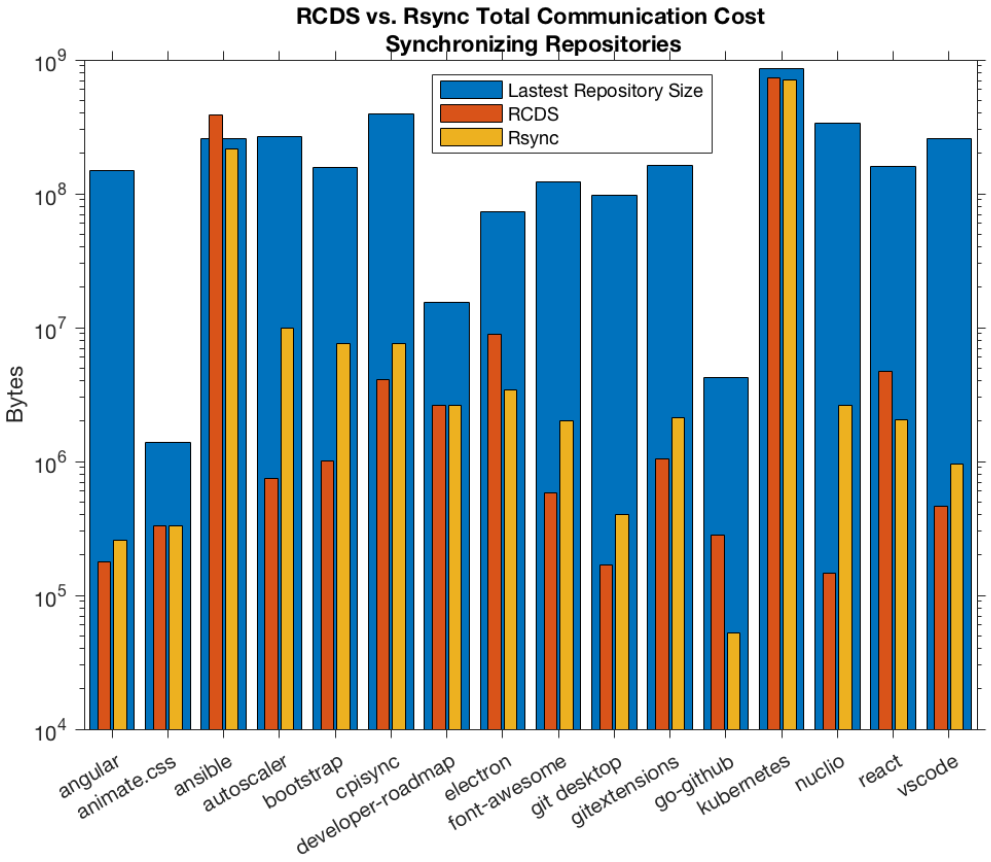

Figure 4·13: Comparing the communication performance of RCDS and *rsync* when synchronizing the entire repositories between the latest and the second latest releases. In most of the cases, the communication cost of RCDS is much less than that of *rsync*. The performance includes all communication cost reconciling edited files between releases. The average percentage of file edits determines the overall performance of RCDS and *rsync*.

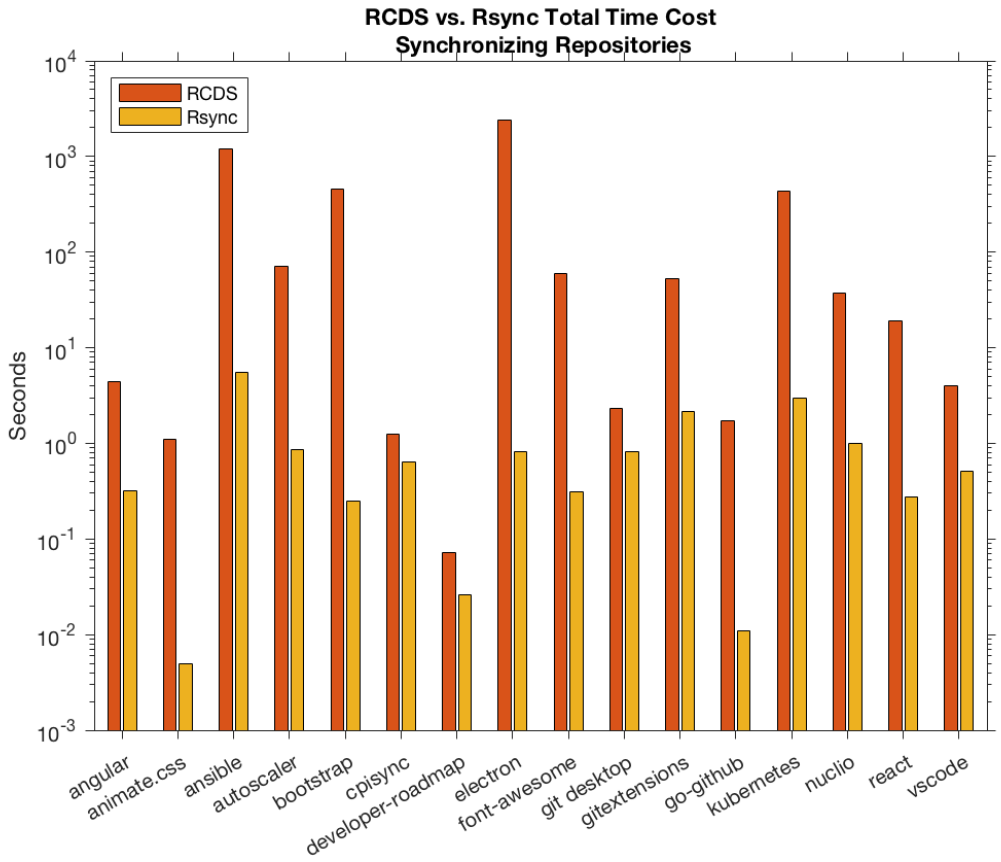

Figure 4·14: Comparing the time performance of RCDS and *rsync* synchronizing the entire repositories between the latest and the second latest release. The time cost is primarily determined by the number of different files between releases.

Repository Name	Latest Release	Second Latest Release
angular.js [3]	v1.7.8	v1.7.7
animate.css [11]	v3.7.0	3.6.2
ansible [4]	v2.8.0a1	v2.7.10
autoscaler [36]	vertical-pod-autoscaler-0.5.0	cluster-autoscaler-1.14.0
bootstrap [56]	v4.3.1	v3.4.1
cpisync [6]	8b1e9d3	288f8b1
developer-roadmap [33]	2018	2017
electron [17]	v6.0.0-nightly.20190404	v4.1.4
font-awesome [21]	5.8.1	5.8.0
git desktop [13]	release-1.6.5	release-1.6.4
gitextensions [24]	v3.0.2	v3.0.1
go-github [27]	v24.0.1	v24.0.0
kubernetes [37]	v1.14.2-beta.0	v1.15.0-alpha.1
nuclio [44]	1.1.2	1.1.1
react [19]	v16.9.0-alpha.0	v16.8.6
vscode [40]	1.33.1	1.33.0

Table 4.3: Git repository versions used comparing synchronization performance between RCDS and *rsync* shown in Figures 4·13 and 4·14.

4.4 Summary of Performance

The RCDS is a bandwidth-efficient string reconciliation protocol that has its communication cost linear in the edit distance between the reconciling strings. The protocol has a specific range of optimal parameters depending on the length of input strings and applies to various types of string content. Therefore, we can determine a set of optimal protocol parameters just by looking at the input string length. We see from Section 4.2 that the protocol is scalable to larger string sizes by increasing the levels of recursion, trading time cost with communication cost. Depending on specific situations, the protocol can be adjusted to suit different system conditions in terms of available computing power and network characteristics.

As a synchronization utility, RCDS performs well reconciling large strings with small edit distance. The time cost is significantly larger than that of the *rsync* utility. We believe that a more efficient implementation could improve our time cost to be more comparable to *rsync* time cost. When synchronizing git repositories, RCDS requires less communication cost than that of *rsync* when the amount of edits between repository releases is small and is less efficient in reconciling mostly different repository releases.

Chapter 5

Conclusion

We have surveyed the existing algorithms for both set and string reconciliation and analyzed their contributions and weaknesses. For our contributions, we have presented a new string reconciliation protocol with sub-linear communication complexity that is scalable regarding the input string size. The protocol integrates the benefit of partitioning and shingling techniques and uses a backtracking algorithm with reduced computation cost to rearrange reconciled partitions. The protocol successfully takes advantage of set reconciliation using reduced communication cost without introducing too much redundancy under the expected case. Finally, we compared our protocol performance to that of *rsync* using our open source implementation.

From our results, the RCDS is proven to be a scalable string reconciliation protocol to large-size string data, minimizing the communication cost using a reasonable amount of computation. Although the RCDS is a content-dependent protocol, its optimal parameters are still mostly dependent on the input string size, which makes parameters readily determined by reconciling string size. The RCDS can also control the trade-off between the time and communication cost to fit into different networks and system conditions considering computing power and network characteristics. The RCDS has a lower communication cost compared to the *rsync* utility synchronizing folders and files when the amount of edit is small and concentrated in a few locations; conversely, the RCDS is terrible to reconcile a large number of sparse edits.

Overall, our algorithm is a bandwidth-efficient protocol for reconciling ordered data. Our application includes situations like incremental file updates, consistency maintenance, and content distribution in distributed networks; especially with wireless networks involving long ranged transmissions where the bandwidth of the communication is highly restrictive.

5.1 Future Work

We believe that our protocol can benefit from sufficient parameter studying for different types of data and string size. Furthermore, if we use different but similar underlying data structure, the overall performance of the protocol can change dramatically. Finally, while not explicitly stated, our partition tree supports incremental changes to the input strings. By maintaining a partition tree, we can reduce the overhead for each reconciliation.

5.1.1 Parameters and Data Structures

The existing synchronization protocols such as the *rsync* utility, has its performance determined by parameters like block granularity [38], of which is adjusted by different implementations for synchronizing different types of data. Likewise, we can also improve our protocol performance by studying its parameters, since the performance of our protocol is content-dependent by nature.

5.1.2 Underlying Algorithms

In our protocol design, we combine a local minimum chunking method with set reconciliation algorithms. Although not explicitly compared, using different set reconciliation protocols yields different performances. For example, the IBLT [26] is a

probabilistic model that has a large communication overhead, whereas the CPI [41] takes a longer time to resolve set differences. Moreover, we can explore other chunking methods mentioned in [5, 32] such as the chunking method in the Low Bandwidth File System [43] to increase the amount of matching partitions or reduce partition time.

In addition, an alternative design to our partition tree is to compute recursive partitions based on the minimum terminal string size without fixing the levels of partitions. For example, after the first iteration of chunking, we can compare the partition hashes with its counter-party and only continue to partition the unmatched substrings until the terminal string size. While this iterative method increases the number of communication rounds, it might help to reduce the overall communication cost.

5.1.3 Incremental Partition

One of the weaknesses of the RCDS protocol is the time cost for constructing a partition tree in each reconciliation. While the cost can be reduced by using parallel sorting algorithms [55], we believe some future work to formalize maintaining partition trees for incremental edits can reduce the overhead of the protocol and supports continuous [60] and resumable [61] file synchronization.

5.1.4 Hybrid Approach

At last, we can also improve our protocol by combining the RCDS with other protocols to complement our deficiency at reconciling strings with sparse or a large number of edits. We can first assess if our protocol is fit to reconcile the strings by constructing the partition tree for both reconciling parties. We could use Strata Estimator (Section 2.1.4) to guess the amount of partition tree difference to predict the amount of time

and communication needed to reconcile the string. We could quote the amount of communication cost by estimating the number of terminal partitions that are different and take the average terminal string size to give an estimation. Such estimation could also be useful for general string edit distance estimation aside from our reconciliation. In the end, we could decide if we should use RCDS for reconciling the target strings.

Bibliography

[1] S. Agarwal, V. Chauhan, and A. Trachtenberg. "Bandwidth Efficient String Reconciliation Using Puzzles". In: *IEEE Transactions on Parallel and Distributed Systems* 17.11 (Nov. 2006), pp. 1217–1225. ISSN: 1045-9219. DOI: 10.1109/TPDS.2006.148.

[2] Saurabh Agarwal et al. "Adaptive incremental checkpointing for massively parallel systems". In: *Proceedings of the 18th annual international conference on Supercomputing*. ACM. 2004, pp. 277–286.

[3] Angular. *angular.js*. `https://github.com/angular/angular.js`. Accessed: 2019-04-17.

[4] Ansible. *ansible*. `https://github.com/ansible/ansible`. Accessed: 2019-04-17.

[5] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. "Content-dependent chunking for differential compression, the local maximum approach". In: *Journal of Computer and System Sciences* 76.3-4 (2010), pp. 154–203.

[6] Bowenislandsong. *cpisync*. `https://github.com/Bowenislandsong/cpisync`. Accessed: 2019-04-17.

[7] John W Byers, Michael Luby, and Michael Mitzenmacher. "Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads". In: *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*. Vol. 1. IEEE. 1999, pp. 275–283.

[8] John W Byers et al. "Informed content delivery across adaptive overlay networks". In: *IEEE/ACM transactions on networking* 12.5 (2004), pp. 767–780.

[9] Di Chen et al. "Robust set reconciliation". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM. 2014, pp. 135–146.

[10] Graham Cormode et al. “Communication complexity of document exchange”. In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2000, pp. 197–206.

[11] Daneden. *animate.css*. `https://github.com/daneden/animate.css`. Accessed: 2019-04-17.

[12] Wayne Davison. *rsync*. `https://rsync.samba.org/`. Accessed: 2019-03-25.

[13] Desktop. *desktop*. `https://github.com/desktop/desktop`. Accessed: 2019-04-17.

[14] Yevgeniy Dodis, Leonid Reyzin, and Adam 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.

[15] Richard Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[16] Salim El Rouayheb et al. “Synchronization and deduplication in coded distributed storage networks”. In: *IEEE / ACM Transactions on Networking* 24.5 (2016), pp. 3056–3069.

[17] Electron. *electron*. `https://github.com/electron/electron`. Accessed: 2019-04-17.

[18] David Eppstein et al. “What’s the difference?: efficient set reconciliation without prior context”. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 218–229.

[19] Facebook. *react*. `https://github.com/facebook/react`. Accessed: 2019-04-17.

[20] Philippe Flajolet and G Nigel Martin. “Probabilistic counting algorithms for data base applications”. In: *Journal of Computer and System Sciences* 31.2 (1985), pp. 182–209.

[21] FortAwesome. *Font-Awesome*. `https://github.com/FortAwesome/Font-Awesome`. Accessed: 2019-04-17.

[22] Marco Gentili. “Set Reconciliation and File Synchronization Using Invertible Bloom Lookup Tables”. PhD thesis. Harvard University, 2015.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google file system*. Vol. 37. ACM, 2003.

[24] Gitextensions. *gitextensions*. https://github.com/gitextensions/gitextensions. Accessed: 2019-04-17.

[25] GitLab. *GitLab*. https://docs.gitlab.com/. Accessed: 2019-03-25.

[26] Michael T Goodrich and Michael Mitzenmacher. "Invertible bloom lookup tables". In: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE. 2011, pp. 792–799.

[27] Google. *go-github*. https://github.com/google/go-github. Accessed: 2019-04-17.

[28] Google. *gsutil Tool*. https://cloud.google.com/storage/docs/gsutil/. Accessed: 2019-03-25.

[29] Project Gutenberg. *Project Gutenberg*. https://www.gutenberg.org/. Accessed: 2019-03-25. n.d.

[30] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. "Resource discovery in distributed networks". In: *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM. 1999, pp. 229–237.

[31] John H Howard et al. "Scale and performance in a distributed file system". In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 51–81.

[32] Piotr Indyk and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM. 1998, pp. 604–613.

[33] Kamranahmedse. *developer-roadmap*. https://github.com/kamranahmedse/developer-roadmap. Accessed: 2019-04-17.

[34] Richard M Karp and Michael O Rabin. "Efficient randomized pattern-matching algorithms". In: *IBM Journal of Research and Development* 31.2 (1987), pp. 249–260.

[35] Aryeh Kontorovich and Ari Trachtenberg. "Efficiently decoding strings from their shingles". In: *arXiv preprint arXiv:1204.3293* (2012).

[36] Kubernetes. *autoscaler*. https://github.com/kubernetes/autoscaler. Accessed: 2019-04-17.

[37] Kubernetes. *kubernetes*. https://github.com/kubernetes/kubernetes. Accessed: 2019-04-17.

[38] Zhenhua Li et al. “A Quantitative and Comparative Study of Network-Level Efficiency for Cloud Storage Services”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 4.1 (2019), p. 3.

[39] Nan Ma, Kannan Ramchandran, and David Tse. “A compression algorithm using mis-aligned side-information”. In: *2012 IEEE International Symposium on Information Theory Proceedings*. IEEE. 2012, pp. 16–20.

[40] Microsoft. *vscode*. `https://github.com/Microsoft/vscode`. Accessed: 2019-04-17.

[41] Y. Minsky, A. Trachtenberg, and R. Zippel. “Set reconciliation with nearly optimal communication complexity”. In: *IEEE Transactions on Information Theory* 49.9 (Sept. 2003), pp. 2213–2218. ISSN: 0018-9448. DOI: `10.1109/TIT.2003.815784`.

[42] Yaron Minsky and Ari Trachtenberg. “Practical set reconciliation”. In: *40th Annual Allerton Conference on Communication, Control, and Computing*. Vol. 248. 2002.

[43] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. “A low-bandwidth network file system”. In: *ACM SIGOPS Operating Systems Review*. Vol. 35. ACM. 2001, pp. 174–187.

[44] Nuclio. *nuclio*. `https://github.com/nuclio/nuclio`. Accessed: 2019-04-17.

[45] Alon Orlitsky and Krishnamurthy Viswanathan. “Practical protocols for interactive communication”. In: *2001 IEEE International Symposium on Information Theory, 2001. Proceedings*. IEEE. 2001, p. 115.

[46] Mohammad Peyravian, Allen Roginsky, and Ajay Kshemkalyani. “On probabilities of hash value matches”. In: *Computers & Security* 17.2 (1998), pp. 171–176.

[47] Calicrates Policroniades and Ian Pratt. “Alternatives for Detecting Redundancy in Storage Systems Data.” In: *USENIX Annual Technical Conference, General Track*. 2004, pp. 73–86.

[48] Edward M Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial algorithms: theory and practice*. Prentice Hall College Division, 1977.

[49] Frederic Sala et al. “Synchronizing files from a large number of insertions and deletions”. In: *IEEE Transactions on Communications* 64.6 (2016), pp. 2258–2273.

[50] Stefan Savage et al. "The End-to-end Effects of Internet Path Selection". In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 289–299. ISSN: 0146-4833. DOI: 10.1145/316194.316233. URL: http://doi.acm.org/10.1145/316194.316233.

[51] Marc Shapiro et al. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[52] Steven S Skiena and Gopalakrishnan Sundaram. "Reconstructing strings from substrings". In: *Journal of Computational Biology* 2.2 (1995), pp. 333–353.

[53] Dan Teodosiu et al. *Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression*. Tech. rep. https://www.microsoft.com/en-us/research/publication/optimizing-file-replication-over-limited-bandwidth-networks-using-remote-differential-compression/. Microsoft Research Lab - Redmond, Nov. 2006, p. 16.

[54] Ari Trachtenberg, David Starobinski, and Sachin Agarwal. "Fast PDA synchronization using characteristic polynomial interpolation". In: *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3. IEEE. 2002, pp. 1510–1519.

[55] Andrew Tridgell. "Efficient algorithms for sorting and synchronization". PhD thesis. Australian National University Canberra, 1999.

[56] Twbs. *bootstrap*. https://github.com/twbs/bootstrap. Accessed: 2019-04-17.

[57] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. "A gossip-style failure detection service". In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag. 2009, pp. 55–70.

[58] Ramji Venkataramanan, Vasuki Narasimha Swamy, and Kannan Ramchandran. "Efficient interactive algorithms for file synchronization under general edits". In: *2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE. 2013, pp. 1226–1233.

[59] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[60] Charles Wu and George T Hu. *Continuous object sychronization between object stores on different computers*. US Patent 6,125,369. Sept. 2000.

[61] Enning Xiang, Eric Knauft, and Pascal Renauld. *Resumable replica resynchronization*. US Patent App. 15/223,337. Feb. 2018.

[62] H. Yan, U. Irmak, and T. Suel. "Algorithms for Low-Latency Remote File Synchronization". In: *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. Apr. 2008. DOI: 10.1109/INFOCOM.2008.40.

[63] SM Sadegh Tabatabaei Yazdi and Lara Dolecek. "A deterministic polynomial-time protocol for synchronizing from deletions". In: *IEEE Transactions on Information Theory* 60.1 (2014), pp. 397–409.

CURRICULUM VITAE