2013

Prioritized data synchronization with applications

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

PRIORITIZED DATA SYNCHRONIZATION WITH APPLICATIONS

by

JIAXI JIN

B.S., Donghua University, 2009

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2013

Approved by

First Reader

Ari Trachtenberg, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

David Starobinski, PhD
Professor of Electrical and Computer Engineering

Third Reader

Prakash Ishwar, PhD
Associate Professor of Electrical and Computer Engineering

PRIORITIZED DATA SYNCHRONIZATION WITH APPLICATIONS

JIAXI JIN

ABSTRACT

We are interested on the problem of synchronizing data on two distinct devices with differed priorities using minimum communication. A variety of distributed systems require communication efficient and prioritized synchronization, for example, where the bandwidth is limited or certain information is more time sensitive than others. Our particular approach, P-CPI, involving the interactive synchronization of prioritized data, is efficient both in communication and computation. This protocol sports some desirable features, including (i) communication and computational complexity primarily tied to the number of differences between the hosts rather than the amount of the data overall and (ii) a memoryless fast restart after interruption. We provide a novel analysis of this protocol, with proved high-probability performance bound and fast-restart in logarithmic time. We also provide an empirical model for predicting the probability of complete synchronization as a function of time and symmetric differences.

We then consider two applications of our core algorithm. The first is a string reconciliation protocol, for which we propose a novel algorithm with online time complexity that is linear in the size of the string. Our experimental results show that our string reconciliation protocol can potentially outperform existing synchronization tools such like rsync in some cases. We also look into the benefit brought by our algorithm to delay-tolerant networks(DTNs). We propose an optimized DTN routing protocol with P-CPI implemented as middleware. As a proof of concept, we demonstrate improved delivery rate, reduced metadata and reduced average delay.

Contents

1 Introduction **1**

1.1 Data Synchronization Problem 1

1.2 Taxonomy of Existing Approaches 2

1.2.1 Version Vector 2

1.2.2 Direct Hashing 3

1.2.3 Bloom Filter 5

1.2.4 Error-Correction Code 6

1.2.5 Interpolation 7

1.3 Data Synchronization in Network Protocols 7

1.3.1 Routing Protocols for Delay-Tolerant Networks 7

1.3.2 Gossip-based Protocols 9

1.3.3 On-line Encryption Key Exchange 10

1.4 Contribution 10

1.4.1 Outline 11

2 Characteristic Polynomial Interpolation **13**

2.1 The Data Synchronization Problem Model-up 13

2.2 CPIsync 13

2.3 Probabilistic 15

2.4 Interactive 16

2.5 Prioritized 18

2.5.1 Protocol: Priority CPI(P-CPI) 19

2.5.2 Worst-case analysis 20
2.5.3 High-probability analysis 22
2.5.4 Restarting interrupted P-CPI 26
2.5.5 Overhead introduced by priority 27
2.6 Experiment 28
2.6.1 Communication Complexity 29
2.6.2 False Positive Rate 29
2.6.3 Estimation Modeling 30

3 String Reconciliation **34**
3.1 Unique String Decoding 34
3.2 Problem Setup and Notations 34
3.3 Reconstruction Ambiguity in String Reconciliation 35
3.4 De Bruijn Graph 36
3.5 Algorithms 37
3.5.1 Checking Unique Decodability 37
3.5.2 Patching Unique Decodability 40
3.5.3 Example 45
3.6 Analysis 47
3.6.1 Data Structure 47
3.6.2 Runtime Analysis: Algorithm 1 47
3.6.3 Runtime Analysis: Algorithm 2 48
3.7 Experiments 49
3.7.1 Bernoulli String 49
3.7.2 English Text 50

4 An application to Delay Tolerant Networks **55**
4.1 The RAPID Routing Protocol 55

4.2 Modified RAPID with P-CPI Deployed 56
4.3 Performance Evaluation 57

5 Conclusion **63**

5.1 Conclusion 63

References **65**

Curriculum Vitae **69**

List of Figures

1·1 Conflict in version vector based file system: After generated, file A and B are accessed/updated at left side, while file C is updated at right side, a typical conflict occurs when trying to merge the copies from two sides since neither copy dominates. 3

1·2 Exchanging hashed values instead of verbatim chunks via slow link. . 4

1·3 A query of w to a Bloom filter representing the set $\{x, y, z\}$, $m = 10$, $k = 3$. 5

2·1 Example execution of I-CPI given $\bar{m} = 1$ 18

2·2 Example execution of P-CPI on priority partition trees with priorities 0, 1, 2. 20

2·3 Communication complexity of P-CPI 29

2·4 False Positive Rate of P-CPI, two different implementations, input sets on 2^{10}, $k = \eta = 1$ 31

2·5 The training data and estimate model 32

3·1 A De Bruijn graph corresponding to the string $abaa$, note that another Eulerian path $aaba$ can also be found in this graph. 37

3·2 Demonstration of Algorithm 2 on the string "arkansas" with $l = 2$ and delimiter $ 46

3·3 Average number of merges needed to transform a length 1000 Bernoulli string to unique decodable shingle set, on varying shingle length. . . 50

3·4 Average number of merges needed to uniquely decode a length 1000 Markovian bit string with a length-10 substring appear with probability p. 51
3·5 Cumulative distribution for a length 1000 markovian bit string with a length-10 substring appear with probability p being naturally unique decodable, on varying shingle lendgth. 52
3·6 Average number of merges needed to transform 2000 consecutive characters taken from English texts to unique decodable, on varying shingle lengthes, along with the amount of communication bits. 53
3·7 Comparison with rsync of reconciliation tasks on English text and file. 53
3·8 Reconciling length 100000 English texts. 54
3·9 Computation complexity: Online time (in seconds) of Algorithm 2 on varying shingle/string length. 54

4·1 Fraction of metadata versus packet generation rate 58
4·2 Fraction of exchanged metadata to all data sent in the network 59
4·3 Average packet delivery time: delay from packet generation till delivery or end of simulation, no delivery deadline 59
4·4 Average packet delivery time: delay from packet generation till delivery or expiration with delivery deadline of 10000 s. 60
4·5 Percentage of packets delivered before expiration with delivery deadline of 10000 s 60
4·6 Average communication overhead per meeting 61

Table of Notations

Name	Representation	First Usage
b	length of bit string	Page 13
m	size of symmetric difference	Page 14
$\bar{m}$	upper bound estimation on m used in CPIsync	Page 15
k	number of verificationb points used in CPIsync	Page 16
ϵ	probability of error of CPIsync	Page 16
p	partition factor used in I-CPI and P-CPI	Page 17
η	priority ratio	Page 21
n, w	string w of n characters	Page 34
Σ	a finite alphabet	Page 34
\$	unique delimiter	Page 34
l	length of shingling	Page 34
$G(S)$	De Bruijn graph of shingle set S	Page 36
$\psi(v)$	parent edge of a vertex $v \in G(S)$	Page 36
$\phi(v)$	number of cycles that v is involved	Page 40

Chapter 1

Introduction

1.1 Data Synchronization Problem

Data synchronization is the process of establishing consistency among data from a source to one or more target data storages, which need to keep multiple copies of a set of data coherent with one another. Examples of data synchronizations can be found in a variety of cluster file system or database replication techniques. In many applications, communications links are easily saturated by the amount of data that must be synchronized, and this is especially the case for wireless systems, where a number of hosts share a broadcast medium, or large networks, where many hosts communicate through a common bottleneck. In such cases it is important that synchronization occur both quickly and with little communication.

In cases of extreme asymmetry, when the data to be synchronized is large and the communication channel particularly limited, as can happen between two passing cars in a vehicular network or between moon rovers sharing video data, one needs to be able to make real-time decisions about which data should be synchronized. In even harsher environments, such as delay tolerant networks where connectivity may outright fail for long periods of time, such synchronization must be resilient to repeated stopping and restarting. It is within this context that we consider priority-based synchronization, where data is provided with priority values under the stipulation that higher priority data should be synchronized before lower priority data. Our protocol further has the desirable properties, of synchronizing higher priority items before lower priority

items, and being able to be stopped and restarted with minimal overhead.

1.2 Taxonomy of Existing Approaches

To evaluate data synchronization approaches by the amount of communication, one can safely start with the worse-case named Slow Sync, which is a wholesale exchange of the databases. This approach, while simple, becomes inefficient when the number of databases or their sizes are large. The problem of efficiently synchronizing data in a variety of distributed systems without Slow Sync has been studied from different perspectives, and based on the foundations for data synchronization primitives we briefly summarize them into five categories.

1.2.1 Version Vector

In order to detect conflicts among file copies, version vectors were at first proposed for use in the distributed operating system LOCUS (Popek et al., 1981). In this approach, each copy of a file f has a version vector associated with it that counts the number of updates of f originating at each site at which f is stored. The vector consists of a sequence of n pairs, where n is the number of sites at which f is stored; the ith vector entry $(S_i : v_i)$ counts the number of updates to f, denoted v_i, originating at site S_i. Conflicts that occur when more than one partition updates the file can be detected by comparing version vectors. Vector v is said to dominate vector v' if v and v' are version vectors for the same file and $v_i \geq v'_i$ for $i = 1$, ..., n. Intuitively, if v dominates v', the copy with vector v has seen a superset of the updates seen by the copy with vector v'. Two vectors are said to conflict if neither dominates. When two sites discover that their version vectors for f conflict, an inconsistency has been detected. Figure 1·1 provides a possible case in version vector file system that causes conflict. How to resolve the inconsistency is left up to the database administrator (DBA).

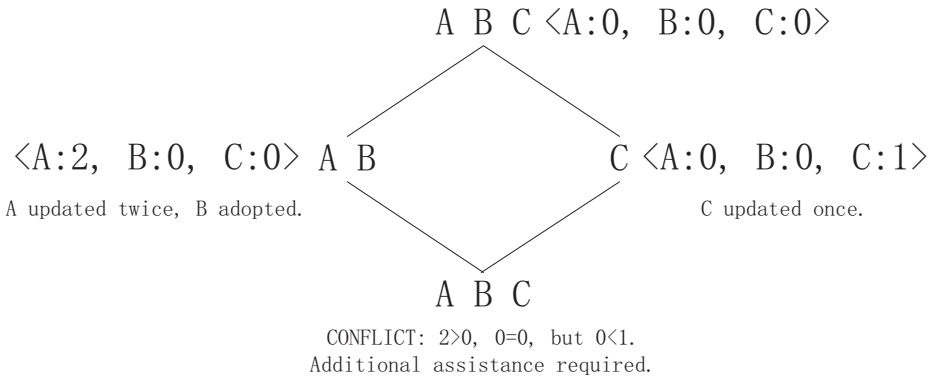


Figure 1·1: Conflict in version vector based file system: After generated, file A and B are accessed/updated at left side, while file C is updated at right side, a typical conflict occurs when trying to merge the copies from two sides since neither copy dominates.

In 1987, the classical Coda file system (Satyanarayanan et al., 1990) has proposed version vector for guaranteeing data consistency and resolving conflicting updates among replicated, distributed databases. Although the focus on Coda already shifted from research to robust performance for commercial use, Coda-alike storage systems using cache and version vector to keep track of data modification are still under development (Eshel et al., 2010). Systems in Coda family replicates files for high availability at the expense of consistency.

1.2.2 Direct Hashing

Another approach to synchronization is simply based on the idea of hashing. To synchronize two files A and B over a slow communication link, with the assumption that A and B are quite similar (perhaps both derived from the same original file), is the basic problem for hash-based synchronization algorithms. To really speed up the synchronization one would need to take advantage of this similarity. A natural idea is to send just the differences between A and B down the link and then use this list of differences to reconstruct the file. The problem is that the normal methods

for creating a set of difference between two files rely on being able to read both files. Thus they require that both files are available beforehand at one end of the link. If they are not both available on the same place the the idea fails. The hash-based synchronization algorithms concentrate on efficiently computing which parts of a source file match some part of an existing destination file. These parts need not to be sent over the link; all that is needed is a reference to the part of the destination file. Only parts of the source file which are not matched in this need to be sent verbatim. The receiver can then construct a copy of the source file using the references to parts of the existing destination file and the verbatim material. Figure 1·2 gives out an example procedure for the use of hashing in determining the differences between file A and B.

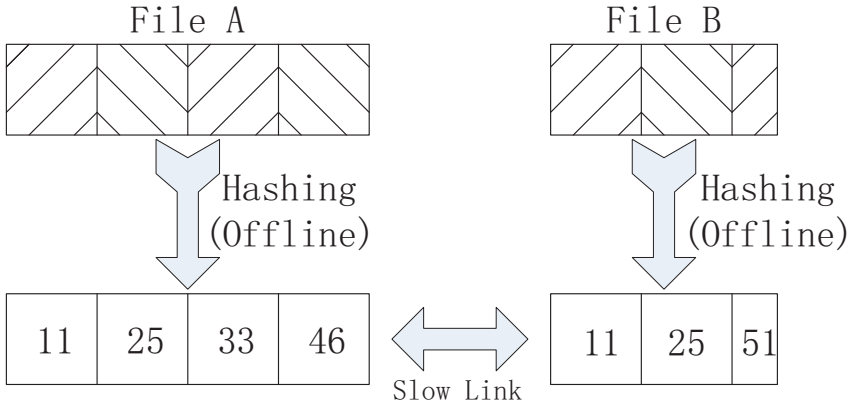


Figure 1·2: Exchanging hashed values instead of verbatim chunks via slow link.

For example, differential compression (Teodosiu et al., 2006) attempts to compare files in chunks; rsync (Tridgell, 2000) and xdelta (MacDonald, 2000; Suel and Memon,) utilize a running hash to detect differences between files. Other approaches include (Lindholm et al., 2009; Li et al., 2011; Yang et al., 2008), to name a few. Though fairly efficient for small files, these approaches do ultimately have communication cost that is linear in the size of the overall data being synchronized. For

networks with large number of nodes or databases, such methods can be prohibitive.

1.2.3 Bloom Filter

As another hash-based approah, Bloom filters were named after Burton H. Bloom, who introduced them in (Bloom, 1970). In this paper he compared the space-time trade-offs of different types of hash-tables. He found that a new type of hash-table, which is now known as a Bloom filter needed less time to reject elements that are not in the table and less space to store these elements. In essence, Bloom filter is a *probabilistic* data structure to test whether an element is a member of set, with arbitrarily low false positive rate. The construction algorithm for Bloom filter starts with an empty, all-zero bit array of m bits, and k different hash functions, each of which maps set element to one of the m array positions with a uniform random distribution. Figure 1·3 is a Bloom filter with $m = 10$ and $k = 3$, in which elements $\{x, y, z\}$ are stored, and a query of another element w would return *false* since it maps to some 0 bit.

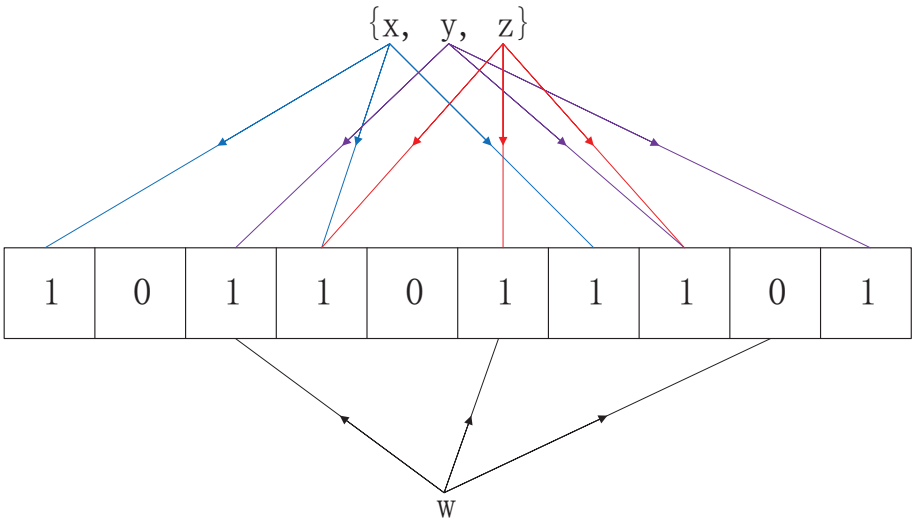


Figure 1·3: A query of w to a Bloom filter representing the set $\{x, y, z\}$, $m = 10$, $k = 3$.

The Bloom filter is another popular solution to achieve fast data synchronization, due to its efficiency in space and time. However, Bloom filter based solutions are not optimal in communication complexity. In a recent paper (Eppstein et al., 2011), a new scheme based on invertible Bloom filter is proposed. This scheme has communication overhead that depends on the size of set differences but gives the optimal performance when the size of the set differences is accurately estimated, a difficult task of its own.

1.2.4 Error-Correction Code

From the perspectives of scalability and performance, it is important that data synchronizations occur with minimum communication, measured both by the number of transmitted bits and by the number of rounds of communication. To examine the data synchronization problem within a generalized framework in which differences between multisets correspond to evaluations of arbitrary "error" functions. There are literatures showing that this problem is equivalent to a variation of error-correction codes. One model involves synchronizing two discrete random variables with some known joint probability distribution using a minimum communication complexity. Orlitsky (Orlitsky, 1991) showed how to use linear error-correction codes for a specific instance of data synchronization. Another model involves two hosts reconciling files (or strings) that differ by a bounded number of insertions, deletions, or modifications (collectively: edits). The problem of efficient reconciliation under these circumstances, also known as the edit-distance problem, has been extensively studied (Cormode et al., 2000; Schwarz et al., 1990) because of its connections to important fields such as file synchronization and pattern recognition. Levenshtein (Levenshtein., 1965) pioneered work in this area by developing error-correcting codes capable of correcting precisely these types of errors.

The communication-optimal solution to data synchronization algorithm has also been shown to exist from the information theory perspective, by the result in (Abdel-

Ghaffar and Abbadi, 1994), which uses Reed-Solomon codes to determine differences between files. However, no applicable algorithms on finding corresponding error-control codes on any generalized field size is proposed. More recently, a work about function computation between two terminals using interactive distributed source coding appeard in (Ma and Ishwar, 2008).

1.2.5 Interpolation

The most related work to this paper are based on mathematical synchronization of data (Minsky et al., 2003; Minsky and Trachtenberg, 2002; Abdel-Ghaffar and Abbadi, 1994), which are described in detail in Section 2.2. The work in (Minsky et al., 2003) describes the characteristic polynomial interpolation technique, an algorithm which is nearly-optimal in communication complexity and (Minsky and Trachtenberg, 2002) outlines an interactive scheme for synchronization together with an expected-case analysis of the algorithm. The high-probability analysis of P-CPI performed in this paper can also be applied to the algorithm proposed in (Minsky and Trachtenberg, 2002).

1.3 Data Synchronization in Network Protocols

Data synchronization techniques can be applied in many network protocols, optimized features of synchronization, such as prioritization can be introduced on purpose to meet some special QoS requirement.

1.3.1 Routing Protocols for Delay-Tolerant Networks

Delay-tolerant networks (DTNs), also known as disruption-tolerant networks, can be viewed as a generalization of mobile networks where all the nodes (not only end-systems) experience sporadic connectivity access to the rest of the network (Fall, 2003; Lindgren et al., 2003; Jain et al., 2005). Examples of such networks include vehicular

ad-hoc networks (VANET) (Menouar et al., 2006) and rural networks connected to the Internet via low earth orbit satellites (LEOs) (Jain et al., 2004).

One of the key issues in DTNs is how to route a packet towards its destination when, at any point of time, the network is not fully connected. An approach first suggested in (Fall, 2003) is to make use of "custody transfer" in which a node communicates opportunistically when a link is available and transfers the responsibility for reliable delivery of a packet to another node. Further, in order to speed-up delivery and make it more reliable, it has been suggested that the same packet should be transferred to several different nodes, thus creating several replicas (see, e.g., (Jain et al., 2005; Balasubramanian et al., 2007)).

The problem with the replication approach is that the number of replicas can potentially become very large, thus wasting precious communication and storage resources and severely degrading performance (Balasubramanian et al., 2007). Within such a context, a communication-efficient data synchronization protocol is essential to ensure that nodes exchange only the replicas that they do not already possess.

Furthermore, due to the limited bandwidth and communication time available at each meeting, two nodes may only be able to exchange a subset of their differing packets. Thus, in order to achieve desirable system performance, several DTN protocols (Balasubramanian et al., 2007; Ramanathan et al., 2007) assign priorities to packets based on metadata information, such as delivery deadlines and statistics of inter-meeting times between different pair of nodes. Packets with higher priority are transmitted first.

The RAPID (Balasubramanian et al., 2007) DTN routing protocol, used as the basis for simulations in our paper, can be configured to minimize average packet delay, maximize average delivery rate and minimize the maximum delay of all packets. Nodes prioritize packets by their utility functions, which are calculated in terms

of these metrics. Experiments in (Balasubramanian et al., 2007) show that under conditions such as high packet generation rates and small packet sizes, the fraction of metadata may considerably increase. This fact motivates us to propose schemes to efficiently manage the exchange of such metadata.

PREP (Ramanathan et al., 2007) is another routing protocol that prioritizes packets, according to the estimated cost from the current node to the destination. Transmission cost is estimated between each pair of nodes. In this manner, the network is seen as a graph. The utility function of a packet is equivalent to the cost of the shortest path in the graph (based on the information available at the node).

In PROPHET (Lindgren et al., 2003), each node keeps a vector of *delivery predictability*, one entry for each destination. A node decides to transfer a packet to the other node if the delivery predictability of the destination of the packet at the other node is higher than its own. The metadata exchanged include the vectors of delivery predictability.

1.3.2 Gossip-based Protocols

Another potential appliance of prioritized data synchronization is in gossip-based protocols, where each node maintains a partial view of the system and forwards messages to relatively small set of nodes known as "partners" chosen randomly out of the entire membership. And the reliability of the protocol depends upon some critical properties of these views, such as degree distribution and clustering coefficient. Research in (Allavena et al., 2005) proposes a gossip protocol transmitting messages with two priorities to ensure high level of reliability even in the presence of high rates of node failure. Priority Issue is also a highly concerned issue in real-time synchronization (Sha et al., 1990) and distributed computing (hong Lim and Agarwal, 1991) since proposed.

1.3.3 On-line Encryption Key Exchange

For symmetric key cryptography to work for online communications, the secret key must be securely shared with authorized communicating parties and protected from discovery and use by unauthorized parties. Public key cryptography can then be used to provide a secure method for exchanging secret keys online. Common key exchange algorithms such as Diffie-Hellman Key Agreement or RSA key exchange process always call for a fast, error-free, and secure key exchange between communicating parties.

1.4 Contribution

In this paper, firstly we propose a prioritized data synchronization protocol for reconciling two remote prioritized sets of data in a communication-efficient manner, we name the protocol Priority-based Characteristic Polynomial Interpolation (P-CPI). We conduct a worst-case and high-probability analysis of P-CPI and prove that if the number of differences between the data sets of two hosts is m, then both the communication complexity and the computational complexity are $O(m \log m)$ with *high-probability* (i.e., with probability approaching one as m gets large). In addition, we prove that P-CPI can be stopped and restarted with minimal overhead, meaning there is no need to keep state information between any two meetings, and we derive bounds on the computational overhead of such restarts. Experiment results show the performace of our approach, together with a model for predicting the probability of fully synchronization with a given time cutoff.

Secondly, we consider another problem of reconciling two remote strings of arbitrary and unknown similarity using minimum communication, which can be efficiently converted to a data synchronization problem between (sub)strings. For an efficient CPI-based solution (Kontorovich and Trachtenberg, 2012), we come up with an al-

gorithm which reduce ambiguity in the decoding process of the CPI-based approach, along with the proof of the correctness. For a length n string, our algorithm can determine its unique decodable set in $O(n)$ time. We also provide related analysis and experiment results on both bit-strings and English text.

Finally, we discuss the potential benefit that could be brought by our CPI-based techniques to distributed network. As a proof of concept, we demonstrate P-CPI as a synchronization middleware, for the RAPID DTN routing protocol (Balasubramanian et al., 2007). Simulations for typical DTN settings demonstrate the potential of sizable improvement for various performance metrics, including metadata overhead, delay and delivery rate. We also establish a random network with P-CPI deployed for data synchronization task, simulation shows that by adjusting certain parameters of P-CPI we are able to reach optimality in terms of communication or time complexity.

1.4.1 Outline

The rest of this paper is organized as follows. Chapter 2 provides abstract descriptions of Characteristic Polynomial Interpolation (CPI)-based approaches (Minsky et al., 2003; Minsky and Trachtenberg, 2002; Starobinski et al., 2003) to data synchronization, which are the building blocks of our protocol, and then presents our new protocol. We describe the Priority-based CPI (P-CPI) algorithm to handle partial and priority-based synchronization and conduct a detailed analysis of its performance, along with simulation results and an estimation model. Chapter 3 presents a new algorithm of linear time complexity for unique decoding strings, which is an important part of an existing CPI-based approach proposed in (Kontorovich and Trachtenberg, 2012). We also provide corresponding proofs, analysis and experiment results on bit-strings and English text. Chapter 4 shows the implementation of P-CPI into RAPID with simulation results, and discusses the potential benefit that coube be brought by properly adjusting certain parameters in P-CPI to achieve data synchronization over

a network. Chapter 5 concludes the paper.

Chapter 2

Characteristic Polynomial Interpolation

We start this chapter with a review of the salient properties of the basic CPI algorithms that serve as the core of Prioritized CPI.

2.1 The Data Synchronization Problem Model-up

The basic model of data synchronization problem is as follows. A local host A and another remote host B possess sets S_A and S_B respectively. While neither of hosts is assumed to know the contents of the other host's set in advance, their goal is to compute the *symmetric difference* between two sets using minimum amount of communication. The *symmetric difference* of sets S_A and S_B is defined as the set of elements which are in either of the sets but not in their intersection, i.e., $S_A \oplus S_B = (S_A - S_B) \cup (S_B - S_A)$. For the purpose of analysis, we assume that the element of the sets are all b-bit numbers (in practice this can be done by hashing), which bounds the size of the symmetric difference between two sets to 2^b elements.

2.2 CPIsync

The work in (Minsky et al., 2003) presents a *Characteristic Polynomial Interpolation*-based synchronization algorithm (dubbed CPIsync (Agarwal et al., 2002; Starobinski et al., 2003)) that translates the set reconciliation problem into the problem of rational function interpolation. More precisely, given two sets of b-bit numbers S_A and S_B respectively, this algorithm can synchronize the two sets using one message of $S_A \oplus S_B$

samples. The algorithm is only logarithmically dependent on the sizes of the sets (i.e., its complexity is proportional to b). Thus, two data sets could each have millions of entries, but if they differ in only m of them, then each set can be synchronized with the other by a single round communication using one message whose size is about that of mb bits (i.e. the number of differences multiplied by the number of bits per set element).

The *characteristic polynomial* of set $S = \{x_1, x_2, ..., x_n\}$ is defined as

$$P_S(Z) = \prod_{i=1}^{n}(Z - x_i). \tag{2.1}$$

The translation of the data into characteristic polynomials is the key to CPIsync algorithm. During the synchronization, one host sends sampled values of its characteristic polynomial to the other host; the number of samples must be at least as many as the number of symmetric differences (i.e. m) between the two hosts. The second host then computes the differing entries by interpolating the rational function corresponding to the ratio of the two characteristic polynomials from the received samples.

CPIsync requires hosts to have a bound $\bar{m}$ on the number of symmetric differences m between sets S_A and S_B to know how many samples must be communicated between them. In other words, one is assured to recover the symmetric difference of size $|S_A \oplus S_B|$ if it is no larger than $\bar{m}$.

Protocol CPIsync$(S_A, S_B, \bar{m})$: Set Reconciliation for S_A and S_B with at most $\bar{m}$ differences. (Minsky et al., 2003)

1. Each host evaluates its *characteristic polynomial* at $\bar{m}$ sample points.
2. Host A sends the $\bar{m}$ evaluations of S_A to Host B.
3. By characteristic polynomial interpolation, host B computes the set differences, $\Delta A = S_A - S_B$ and $\Delta B = S_B - S_A$.
4. Host B sends the result back to Host A.

As given in (Minsky et al., 2003), CPIsync has a communication complexity of $\Theta(\bar{m}b)$. The main bottleneck of CPIsync is its computation complexity, which is $\Theta(b\bar{m}^3 + bmk)$, cubic in the upper bound $\bar{m}$. This computation cost would be unnecessarily expensive when the upper bound guess $\bar{m}$ on the symmetric difference m is conservative (i.e. $\bar{m} \gg m$).

2.3 Probabilistic

When CPIsync runs with a *guessed* upper bound $\bar{m}$ that is smaller than the actual number of symmetric differences, there is a chance that CPIsync produces false result rather than failing. This false-positive condition happens when the interpolation of rational functions agrees with the interpolation result of other sets with different but fewer symmetric differences. In such case one can verify the result of each CPIsync with k additional samples, making the probability of returning false-positive result arbitrarily small.

If no bound is known on the number of symmetric difference, one can repeatedly try CPIsync with increasing upper bounds. One of the strategies is to double the *guessed* bound on each attempt until the returned result (symmetric difference) is verified, we call this approach Probabilistic CPI algorithm (ProbCPI).

Protocol ProbCPI(S_A, S_B): Set Reconciliation for S_A and S_B with unbounded differences. (Minsky et al., 2003)

1. Each host makes a large number of evaluations on its *characteristic polynomial*.
2. Host A sends 1 evaluation of S_A to Host B.
3. Using received evaluations, the interpolation at Host B either returns ΔA and ΔB, or a flag of failure.
4. Host B sends the returned result back to Host A.
5. Host A executes Step 1 with doubled number of evaluations if a flag of failure is received, otherwise, it sends k additional evaluations of S_A for verification.
6. Host A goes to Step 5 with a flag of failure if the verification fails at Host B otherwise terminates.

Theorem 1. *A ProbCPI sync of sets S_A and S_B attains a probability of error no larger than ϵ if*

$$k \geq \lceil \log_\rho \frac{\epsilon}{b} \rceil \tag{2.2}$$

verification points are used in each constituent CPIsync. Here, b is the number of bits needed to represent a set element and $\rho = \frac{|S_A|+|S_B|-1}{2^b}$.

Proof. Assume one successful ProbCPI sync between two sets of m symmetric difference(s) makes t calls to CPIsync. Since our guessed $\bar{m}$ doubles at each iteration, the number of CPIsync calls t is upper bounded by:

$$t \leq \lceil \log_2 m \rceil \leq \log_2 2^b = b \tag{2.3}$$

A *single-point* CPIsync verification has a false-positive probability no more than $\frac{|S_A|+|S_B|-1}{2^b}$ (Minsky et al., 2003), and this is raised to the k-th power for k independent verifications. The probability ϵ of having an error among $t \leq b$ CPIsyncs is thus, by the union bound:

$$\epsilon \leq b \left[\frac{|S_A| + |S_B| - 1}{2^b} \right]^k \tag{2.4}$$

The result follows from algebraic manipulation. □

2.4 Interactive

The computational complexity of CPIsync can be further reduced by using interaction. *Interactive Set Reconciliation* (I-CPI (Minsky and Trachtenberg, 2002)) utilizes a "divide-and-conquer" approach that recursively partitions the field of all b-bit

strings into p partitions. The partitioning process continues recursively until the number of differences between the sets, restricted to each (sub)field, is at most a constant $\bar{m}$, in which case, using one CPIsync call can determine all the corresponding differences on that field. The constant $\bar{m}$ serves to bound the computational expensive components of CPIsync.

Protocol I-CPI(S_A, S_B): Interactive Set Reconciliation for S_A and S_B.

1. Each host builds up its partition tree.
2. Enter the partition trees from their roots in pairs, push CPIsync$(S_A, S_B, \bar{m})$ into stack as a task.
3. Popup and execute one CPIsync task from the stack, if it successfully determines set differences, go to Step 5; otherwise, continue to Step 4.
4. Denote the input pair of sets of previous CPIsync S_1 and S_2, for the p child nodes of each pair, which are denoted as $S_1.subset[0]$ to $S_1.subset[p-1]$ from left to right and the same as in S_B, push CPIsync$(S_2.subset[i], S_2.subset[i], \bar{m})$ into stack from $i = p-1$ to $i = 0$.
5. Terminate if the stack is empty, otherwise go to Step 3.

The data structure used to realize I-CPI algorithm is called *partition tree* (Minsky and Trachtenberg, 2002), in which each node represents a (sub)set that contains elements on a certain field only. The original range of the field is 2^b at the root node. and a node may equally partition its field into p subfields and assign them respectively to its p child nodes (if any). In a *partition tree*, there are two types of nodes:

- *active* (internal) nodes, representing a (sub)field on which the sets differ by more than $\bar{m}$ differences, meaning that further recursion is needed for synchronization;
- *terminal* (leaf) nodes, representing a (sub)field on which the sets differ by at most $\bar{m}$ differences and thus can be determined by one CPIsync call.

The execution flow of I-CPI can then be described as a simultaneous depth-first traversal of the two partition trees rooted at the original sets. Figure 2·1 illustrates the execution of I-CPI on a binary (p=2) partition tree, between two sets S_A =

$\{1, 2, 4, 16, 21\}$ and $S_B = \{1, 2, 6, 21\}$ on F_{32} (i.e. a finite field including integers from 0 to 31). The parameter $\bar{m}$ of each CPIsync call is set to 1. The call of I-CPI on the given sets S_A and S_B first executes CPIsync$(S_A, S_B, 1)$, labeled by sequence number 1 in the figure. This execution fails since the input sets S_A and S_B differ by three elements. Then I-CPI proceeds to the first left child nodes of the roots at both end, representing subsets of S_A and S_B, numerically $\{1, 2, 4\}$ and $\{1, 2, 6\}$ (labeled by 2), on which another CPIsync is called (and fails). As indicated, CPIsync is called eight times in total until all the differences are determined, each time with a pair of (sub)sets of the same sequence number (in increasing order) as input.

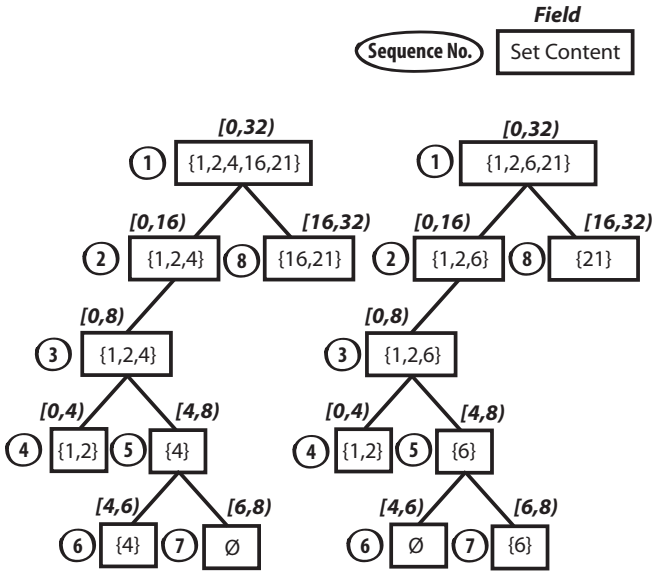


Figure 2·1: Example execution of I-CPI given $\bar{m} = 1$

2.5 Prioritized

In some applications, data may have priorities determined by some attributes (e.g. status, source, timestamp). In these cases, such as DTNs, when bandwidth is significantly constrained, hosts may need to conduct the synchronization in a prioritized fashion, for example, synchronizing data objects with higher priority first. In this sec-

tion we propose our main contribution, a new protocol named Priority CPI (P-CPI) that enables prioritized data synchronization. We analyze its communication and computation complexity, in worst-case , high-probability case, best-case, and in the case of restarting an interrupted synchronization. A brief analysis of communication and computation overhead bought by prioritization is also provided at the end.

2.5.1 Protocol: Priority CPI(P-CPI)

The Priority CPI (P-CPI) algorithm adapts I-CPI to handle prioritization. More precisely, in P-CPI, sets are firstly split by priority, then the synchronization is run on each pair of subsets in decreasing order of priority. As a result, records with different priorities are sent into different partition trees, which guarantees that the limited network bandwidth is first used for data entries with priority.

Figure 2·2 illustrates a sample execution of P-CPI with decreasing priorities 0, 1 and 2, in which S represents the pair of two sets being reconciled by P-CPI and S_i is subset of S that contains all elements of priority i, which is a union of subsets with the same priority at both sides. I-CPI(S_i) is called successively and conditionally to ensure an orderly priority-oriented synchronization. As the arrows indicate, only after the synchronization of S_0 succeeds does the P-CPI start to call another I-CPI with S_1 (the pair of subsets of lower priority) as input and so on.

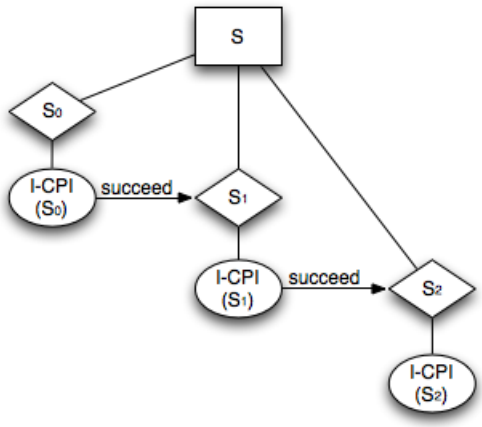


Figure 2·2: Example execution of P-CPI on priority partition trees with priorities 0, 1, 2.

As an added feature, P-CPI algorithm is capable of handling interruptions and returning intermediate synchronization results. Resumption of synchronization after interruption proceeds rather quickly, in time on the order of the bit-length of one data element, as detailed in our analysis in Section 2.5.4.

2.5.2 Worst-case analysis

Our analysis on P-CPI makes use of some common notation:

- *difference bound* $\bar{m}$, which is the designed upper bound of the size of the symmetric difference that can be determined by one call of CPIsync.
- *bit-string length* b, which is the number of bits needed to represent a set entry as a bit-string.
- *partition factor* p (described in Section 2.4), which means an internal node of a *partition tree* can have up-to-p child nodes.

- *priority ratio* η, which is defined as follows:

$$\eta = \frac{\text{number of differences of high priority}}{\text{number of all differences}} \tag{2.5}$$

  If there are more than two priorities in the system, η represents the ratio of number of differences with priority above a given threshold to the total number of differences. Note that $\eta = 1$ for the worst case when all the differences are of high priority and need to be synchronized.

- $I(\eta m)$, which is the overall number of invocations to CPIsync during the execution of P-CPI on two sets with ηm symmetric differences.

The following Lemma provides a worst-case bound on the number of CPIsync calls by P-CPI.

Lemma 1. *For P-CPI to synchronize two sets with m symmetric differences, the number of invocations of CPIsync is bounded by*

$$I(m) \leq 1 + \frac{m}{\bar{m}} p \lceil \log_p(2^b) \rceil \tag{2.6}$$

Proof. The work in (Minsky and Trachtenberg, 2002) bounds the number of invocations of CPIsync by

$$I(m) \leq 1 + \frac{m}{\bar{m}} p \lceil \log_p s \rceil \tag{2.7}$$

as the worst-case condition for I-CPI, where s stands for set size. Substituting $s = 2^b$ gives the desired bound. □

Given that $\Theta(b\bar{m}^3 + bmk)$ is the computation complexity of CPIsync (Minsky et al., 2003), multiplied by (2.6) we attain the *worst-case computation complexity* of P-CPI as $\Theta(m\bar{m}^2 b^2 \frac{p}{logp})$.

Similarly, the *worst-case communication complexity* of P-CPI is $\Theta(m\frac{p}{\log p}b^2)$ given that the communication complexity of CPIsync is $\Theta(mb)$.

2.5.3 High-probability analysis

In this subsection a new probabilistic analysis shows the number of CPIsync invocations is $O(\eta m \log(\eta m))$ with high-probability. Since a set element can be represented by any b-bit string, we assume a uniform-random distribution of the symmetric differences between sets, which is so implemented by a pseudo-random hashing of the data before the synchronization. Based on this assumption, we derive a high-probability bound on the number of CPISync calls by P-CPI. The analysis resembles that of quicksort (Dubhashi and Panconesi, 2009), but the partitioning process of the tree is different, thus requiring a different analysis.

Theorem 2. *For P-CPI to synchronize two sets with ηm uniform-randomly distributed differences in total, the number of calls to CPIsync is $O(\eta m \log(\eta m))$, with probability at least $1 - \dfrac{1}{\eta m}$.*

Proof. Let $m' = \eta m$ and, for the sake of exposition, let the partition factor $p = 2$. Other partition factors would simply change the base of our logarithms in the proof, and would not affect the conclusion.

In a binary partition tree, choose any root-to-leaf path P. Consider a node *good* if the partition made at the node results in two subsets, each with at least one third of its differences before partitioning. Otherwise we consider the node *bad*. If a root-to-leaf path in a partition tree contains t good nodes, then as we go through it, the number of symmetric differences at the t-th good node (denoted m'_t) can be bounded as

$$m'_t \leq \frac{2}{3} m'_{t-1} \leq \left(\frac{2}{3}\right)^t m'_0 \tag{2.8}$$

since a good partition reduces the number of differences contained by the current space to at most 2/3 of what is in the last good node.

It follows that there can be at most

$$t \leq \frac{\log_2 m'}{\log_2 \left(\dfrac{3}{2}\right)} < 2 \log_2 m' \tag{2.9}$$

good nodes in any path.

Next, we apply the Chernoff-Hoeffding bounds (Dubhashi and Panconesi, 2009) to show that

$$Pr\Big[|P| > 4\log_2 m'\Big] < \frac{1}{m'^2}, \tag{2.10}$$

where $|P|$ is the length of the chosen root-to-leaf path P, i.e. the sum of numbers of good and bad nodes.

Let X_i be a random variable taking the value 1 if the ith node is bad, or 0 if it is good. The number of differences contained by the set at ith node is denoted s_i. Since a node is considered good when the number of differences it represents, s_i, satisfies $s_{i-1}/3 \leq s_i \leq 2s_{i-1}/3$, we have:

$$Pr\Big[X_i = 1\Big] = 1 - \frac{\sum_{j=s_{i-1}/3}^{2s_{i-1}/3} \binom{s_{i-1}}{j}}{2^{s_{i-1}}} \leq \frac{2}{3}. \tag{2.11}$$

By our assumption made at the beginning of the subsection, all the X_is are independent due to the uniform-random distribution of symmetric differences. Let X be the total number of bad nodes in the path P, according to (2.11), $E[X] \leq \frac{2}{3}|P|$. Using the Chernoff bound, for $t > 4e|P|/3 \geq 2eE[X]$,

$$Pr\Big[X > t\Big] \leq 2^{-t} \leq 2^{-2\log_2 m'} = \frac{1}{m'^2}, \tag{2.12}$$

provided that $|P| \geq \dfrac{3}{2e}\log_2 m'$.

Since $|P| = X + t$ by definition, from (2.9) and (2.12) we have

$$\begin{aligned} Pr\Big[|P| > 4\log_2 m'\Big] &= Pr\Big[X + t > 4\log_2 m'\Big] \\ &\leq Pr\Big[X > t\Big] \\ &\leq \frac{1}{m'^2} \end{aligned} \tag{2.13}$$

provided that $|P| \geq \dfrac{3}{2e}\log_2 m'$. Note that $Pr\Big[|P| > 4\log_2 m'\Big]$ is trivially zero when $|P| < \dfrac{3}{2e}\log_2 m'$, therefore (2.10) holds for any $|P|$.

Thus, the total number of good and bad nodes along *any* root-to-leaf path does not exceed $4\log_2 m'$ with probability at least $1 - \frac{1}{m'}$. This claim then follows from

the union bound:

$$Pr\Big[\exists P, |P| > 4\log_2 m'\Big] \leq \\ mPr\Big[|P| > 4\log_2 m'\Big] \leq \frac{1}{m'}. \tag{2.14}$$

Since CPIsync called at each leaf node determines at least one symmetric difference, the number of leaf nodes in a partition tree is $O(m')$, which is the same as the number of root-to-leaf path. So the overall number of calls of CPISync, $I(m')$, is:

$$I(m') \in O(m' \log m') \tag{2.15}$$

with probability at least $1 - \frac{1}{m'}$ □

Once again, by multiplying number of CPIsync invocations in the high-probability case with CPIsync's basic communication and computation complexity, we have that the *high-probability computation complexity* of P-CPI is $I(m')\Theta(b\bar{m}^3 + bmk) \in O(\eta mb(\bar{m}^2 + k)\log(\eta m))$ and the *high-probability communication complexity* of P-CPI which is $I(m')(\bar{m}b + 2) \in O(\eta mb \log \eta m)$ both with probability at least $1 - \frac{1}{\eta m}$.

For comparison purpose, we assume that $\bar{m}$ and p are constants given in priori to the synchronization. Thus the number of CPIsync invocations is $O(mb)$ in worst-case and $O(m \log m)$ in high-probability case. Since m cannot succeed 2^b by definition, and in practice $m \ll 2^b$ is most likely the case, we then conclude that the high-probability (communication and computation) complexities of P-CPI are **practically less** than its worst-case complexities.

Since the P-CPI algorithm guarantees that high priority elements are synchronized before those with lower priorities, there is a risk of waiting starvation, which means some low priority elements may get "starved" if they waits too long before they can be synchronized. This best-case analysis suggests to users of P-CPI the bottom-line of taking waiting starvation into consideration.

Theorem 3. *For P-CPI to synchronize two sets with ηm uniformly-random dis-*

tributed symmetric differences, the total number of calls to CPIsync is

$$I(\eta m) = \Omega\left(\frac{\eta m}{\bar{m}}\right) \tag{2.16}$$

Proof. Let k denote the depth of the partition tree of this P-CPI, and p is the partition factor. In the best-case, the number of nodes (same as the number of CPIsync calls) in the partition tree is minimized. Used to represent certain number of differences, the partition tree of minimum size of all those qualified should be completely filled in every level lower than k, since one partition changing a leaf node to an internal node results in p new leaves. In a partition tree of depth k, the number of leaf nodes is upper bounded by p^k. Recall that at most $\bar{m}$ out of m differences can be discovered by one single CPIsync, so the number of CPIsync calls which succeeds to find at least one difference is then lower bounded by $\lceil \frac{\eta m}{\bar{m}} \rceil$. Note that such a "fruitful" CPIsync can appear only at leaf nodes of the partition tree, which makes use of the inequality derived from the upper bound of the number of leaf nodes:

$$\lceil \frac{\eta m}{\bar{m}} \rceil \leq p^k, \tag{2.17}$$

or,

$$k \geq \lceil \log_p \frac{\eta m}{\bar{m}} \rceil. \tag{2.18}$$

Note that $I(\eta m)$ is the same number as the total number of nodes in the partition tree, therefore lower bounded similarly by the minimum number of nodes in the partition tree, in the best-case:

$$I(\eta m) = \Omega\left(\sum_{i=0}^{k} p^i\right) = \Omega\left(\frac{p^{k+1}-1}{p-1}\right) \tag{2.19}$$

Substituting k in (2.18) into (2.19) gives the desired bound. □

Similar as in Section 2.5.2 and 2.5.3 , Theorem 3 gives the *best-case computation complexity* of P-CPI as $\Theta(b\bar{m}^3 + bmk)I(\eta m) \in \Omega(\eta m b(\bar{m}^2 + k))$ and the *best-case communication complexity* of P-CPI as $(\bar{m}b + 2)I(\eta m) \in \Omega(\eta m b)$.

Thus, we have showed the best-case computation and communication complexity of P-CPI are both linear in symmetric difference $m' = \eta m$ and bit-string length b.

According to the best-case analysis, assume that there are i partition trees corresponding to decreasing priorities from 0 to $i-1$, and there are $\eta_i m$ differences at priority i. Suppose further those trees are processed successively and conditionally according to priority. Then a tree must wait for a time period *at least* linear in the total number of differences with priority higher than its own.

2.5.4 Restarting interrupted P-CPI

A commonly-used technique for restarting a synchronization is to save the intermediate results at the interruption, and then reload them when the synchronization process resumes. However, in distributed systems such as DTNs, the memory capacity at each node is usually limited and therefore not capable of saving too many intermediate results.

P-CPI enables a *memoryless* fast restart of previously interrupted synchronization, and no modification to the original P-CPI is needed for this feature. By using P-CPI, a node in a mobile network may use exactly the same protocol to synchronize with any node it meets, and has a fast restart if i) the last synchronization between the same hosts was interrupted and ii) no new difference was added since then.

The execution of P-CPI is essentially a depth-first traversal of one or more pairs of partition trees. If the execution is interrupted and the synchronization breaks at a certain pair of nodes (i.e. a *break pair*), we can conclude that there must be unsynchronized pair(s) at positions not earlier than the break pair in the depth-first traversal of the same pair of partition trees. We call a pair of nodes *synchronizable* if they are unsynchronized and contains no more than $\bar{m}$ differences. In the case of an interruption, hosts update their databases with symmetric differences determined before the interruption and wait for a restart. By restarting P-CPI between the same hosts later (assuming no new differences are added), the synchronization will resume once it finds the first synchronizable pair in their partition trees.

Theorem 4. *The number of CPIsync calls before a synchronizable pair is found in a pair of p-ary partition trees with bit-string length b is upper-bounded by bp.*

Proof. The algorithm proceeds according to the following flow until it finds a synchronizable pair:

- if the CPIsync call on a pair of nodes fails to find differences, which means at least one synchronizable pair are their descendants, then the execution proceed to the first left child nodes of the pair;
- else if the CPIsync call returns no differences (which means no synchronizable pair exists in their descendants) and at least one node in the pair has a non-empty right sibling, then the execution proceed to their (first) right siblings,
- otherwise (when no differences are found and no right sibling exists), the algorithm returns and reports that the current pair of partition trees are fully synchronized.

Since a p-ary partition tree recursively partitions the field of range 2^b equally into p partitions and each leaf nodes is on a (sub)field of range at least (some constant) $\bar{m}$ differences, then height of the partition tree is at most b. The proceeding flow described above attains at most p CPIsync calls per level (as it proceeds along p siblings). Thus the number of CPIsync calls is at most bp. □

As an example, consider two sets containing one million elements and using P-CPI with a partition $p = 2$. In that case, the number of CPIsync calls before a synchronizable pair is found is at most 40, no matter what is the number of differences between the two sets, how many of these differences were previously reconciled, and the value of the parameter $\bar{m}$ used in each CPIsync call. Further, no special information needs to be maintained by the nodes for P-CPI to achieve this performance.

2.5.5 Overhead introduced by priority

Since we introduce priority into CPIsync and more bits are required to represent priority, P-CPI has higher communication and computation complexity than CPIsync

when they are used to reconcile two sets. Here we calculate the overhead of P-CPI over CPIsync.

For CPIsync to reconcile two sets with m differences, the communication complexity is mb and the computation complexity $m(b\bar{m}^3 + b\bar{m}k)$. Now we use P-CPI to reconcile the same two sets with v priority levels. Then we need $\log v$ extra bits to represent the priority of a set entry. Thus the bit-string for representing a set entry is extended to $b + \log v$ bits. Priority divides m differences into $\eta_1 m, \eta_2 m, ..., \eta_v m$ differences($\sum_{i=1}^{v} \eta_i = 1$), which are distributed in different subspaces. To reconcile two subsets with priority i, the communication complexity is $\eta_i m(b + \log v)$ and the computation complexity is $\eta_i m(b + \log v)(\bar{m}^3 + \bar{m}k)$. Summing over all priorities, we get that the total communication complexity is $m(b + \log v)$ and the total computation complexity is $m(b + \log v)(\bar{m}^3 + \bar{m}k)$. Then the overhead of P-CPI over CPIsync is $\frac{\log v}{b}$, in both communication and computation.

We note that when we synchronize two subsets with priority i, the priority bits of entries in the two subsets are the same and already known and thus are redundant. So we only need to confirm that entries in the two subsets have the same priority at the very beginning and call CPIsync on the entries without including priority bits. Therefore, to reconcile two subsets with priority i, the communication complexity is $\eta_i mb + \log v$ and computation complexity is $\eta_i mb(\bar{m}^3 + \bar{m}k)$. Then the total communication complexity is $mb + v \log v$ and the total computation complexity is the same with CPIsync. Thus under this protocol, P-CPI introduces an overhead of $\frac{v \log v}{mb}$ in communication while no additional cost in computation.

2.6 Experiment

In this section, we present numerical results illustrating the performance of P-CPI along with necessary analysis.

2.6.1 Communication Complexity

We implement the Priority CPI (P-CPI) algorithm on Mac computers with 2.16 GHz Intel Core 2 Duo processor and socket-based communication between hosts. Results are averaged over 1000 iterations.

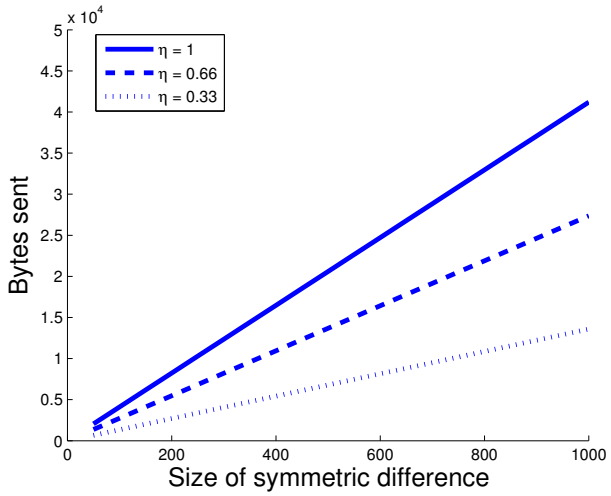


Figure 2·3: Communication complexity of P-CPI

We consider reconciling sets with an average size of 10,000 elements and varying symmetric differences. The P-CPI algorithm is set to terminate after all the elements of high priority are synchronized. Recall that η is the priority ratio indicating which proportion of the symmetric differences is of high priority.

Figure 2·3 shows the close-to-linear relationship between the communication complexity (i.e., the number of bytes transferred) and the number of symmetric differences. The communication complexity is also proportional to the *priority ratio* η, indicating that prioritization effectively reduces time (and bandwidth) usage.

2.6.2 False Positive Rate

To synchronize two sets S_A and S_B with m differences, CPIsync$(S_A, S_B, \bar{m})$ is assured to determine all the m differences correctly provided $\bar{m} \geq m$. However, when $\bar{m} < m$,

in which cases CPIsync$(S_A, S_B, \bar{m})$ is supposed to report a failure, but with a small probability ϵ, the algorithm produces false result(s) rather than failing. The false positive rate of one CPIsync, denoted ϵ, can be arbitrarily reduced to nearly zero by verifying the CPIsync results with k additional samples:

$$\epsilon \leq 2^{-bk}, \tag{2.20}$$

where b is the bit-string length of the entries of sets S_A and S_B. Recall that P-CPI splits original data sets by recursively partitioning the field (See Fig. 2·1), which introduces two different settings of running CPIsync along the partition tree: i) Keep the original bit-string length b of input sets S_A and S_B, which attains a false-positive rate no higher than $(|S_A| + |S_B| - 1)2^{-bk}$, ii) Call CPIsync with decreasing b as the field size decreases due to the partitioning, which saves some bits on communication, but brings in higher false-positive rate.

By recursively applying equation (2.20) on every CPIsync call within the P-CPI execution, we can also obtain the theoretic upper bound on the false positive rate of P-CPI with decreasing field size implementation. The solution is open-formed and only depends on m, $\bar{m}$, k and b(Here b stands for the field size of the original input sets). Figure 2·4 shows a performance contrast between the two implementations of P-CPI. Running P-CPI with CPIsync on decreasing field size can save a certain amount of bits during the communication (and some communication time as well) in the expense of consuming more time (and memory) on computation.

2.6.3 Estimation Modeling

In this subsection we propose an empirical approach to estimating the likelihood that P-CPI ends with complete differences within different cut-off times. The close-to-linear relationship between communication time and symmetric difference makes it

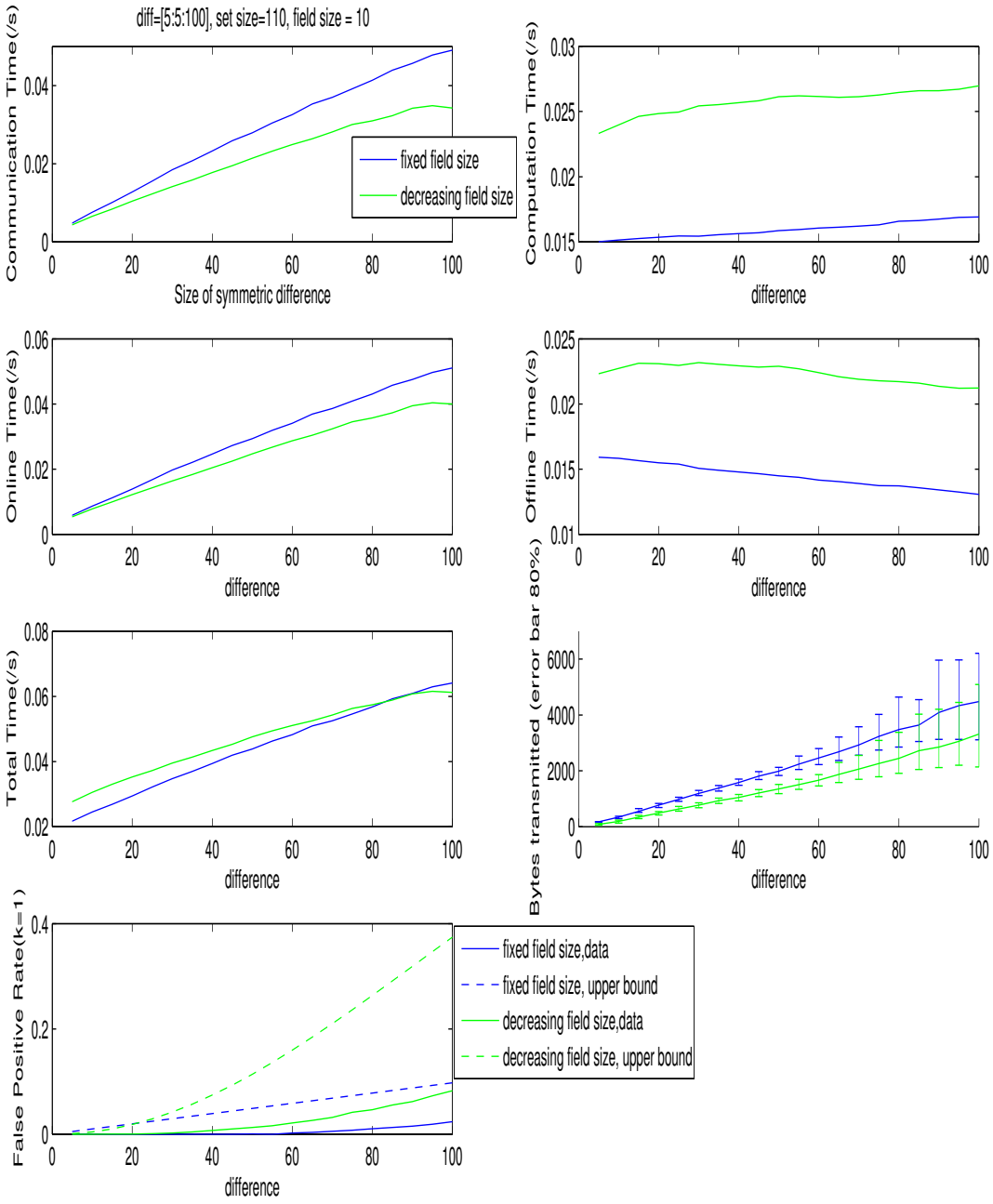


Figure 2·4: False Positive Rate of P-CPI, two different implementations, input sets on 2^{10}, $k = \eta = 1$

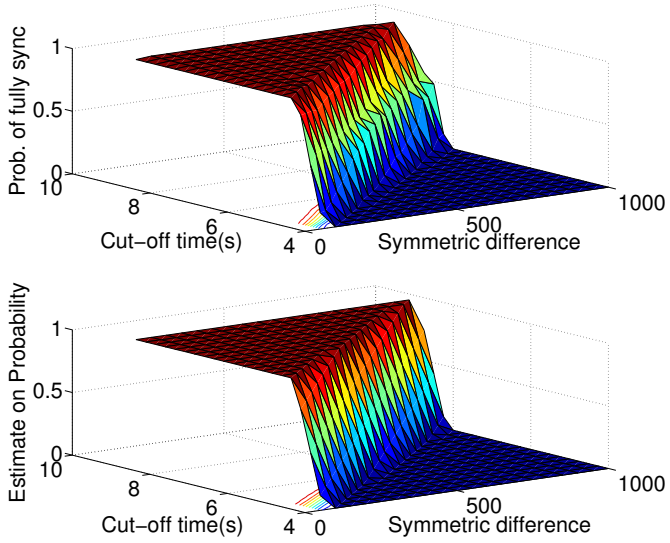


Figure 2·5: The training data and estimate model

possible for us to predict the total time cost for synchronization based on an estimation of the symmetric differences prior to an actual call. In practice, such an estimate can be made by synchronization hosts according to previous synchronization results kept by themselves or other information received from third-party. It is also possible for a host to determine the likelihood of a complete synchronization if it already has an estimate on the symmetric difference between itself and a remote host, and want the synchronizations end within some given cut-off time.

Figure 2·5 shows the contrast of training data (above) and the estimation model (below). The upper half of Figure 5 depicts the percentage (p) of complete synchronizations out of 1000 trials as a function of the symmetric difference (m) between synchronizing sets and cut-off time (t). The contour line is extracted from the training data to help constructing an estimate function. The time-difference linearity of P-CPI leads to a cluster of equally-spaced contour lines with nearly-equivalent slope, which is helpful to confine the estimate function in a fairly-low degree. So for $\forall h \in [0, 1]$, the contour line $p = h$ on the t-m plane can be described by a formula

$$Proj(p = h) : t \approx km + d \tag{2.21}$$

where k is the slope of the contour lines, a constant with our approximation, and t, m, and d are the time threshold, symmetric difference and intercept on t-axis, respectively.

Next, we look for the coefficient of a polynomial $P(p)$ of a degree N as low as possible, which fits $P(p) \approx d$ in a least-square sense, to substitute d in (18) in order to get an estimate on p which is only a function of m and t. For linearity, we use $N = 1$ to get two coefficients satisfying $\alpha p + \beta = d$.

Then, with some algebraic manipulation we arrive at:

$$p = \frac{t - km - \beta}{\alpha} \tag{2.22}$$

Based on three coefficients gained by the training data, this empirical model gives out a conservative estimate on the probability of fully synchronization of a P-CPI as a function of m and t. As an instance, the formula used to plot the second graph is

$$p = \frac{t - 0.0045m - 3.6843}{0.7689} \tag{2.23}$$

With an estimate model that matches the experiment well, a host of P-CPI can simply try with customized desired probability of complete synchronization and estimate on symmetric differences in prior to the synchronization, for a trade-off between time-efficiency and probability of success.

Chapter 3

String Reconciliation

3.1 Unique String Decoding

A string can be split into an ordered collection of overlapping substrings and thus the collection can be used to reconcile the original string. The problem to our interest is, if such a substring collection is shuffled to unordered, whether or not it can still be uniquely reconstructed into a consistent string. The motivation for this kind of unique string decoding or similar problems can be found in a variety of aspects including computational biology, information system, and cryptography. Within a biological text, the similar problem is named "fragment assembly", with a goal to assemble hybrid reads of bacterial genomes (Chaisson et al., 2004), or to (uniquely) reconstruct a protein sequence from its constituent K-peptides (Shi et al., 2007). Communications protocols (Agarwal et al., 2006; Broder, 1997) attempt to identify differences between related documents using string reconciliation. On cryptographically secure purpose, fuzzy extractors (Dodis et al., 2008) also employ similar techniques to find matches for noisy biometric data.

3.2 Problem Setup and Notations

To give a formal statement of the unique string decoding problem that we are going to solve in this chapter, we are provided a length n string $w \in \Sigma_{\$}^{n}$ over a finite alphabet Σ, with a unique delimiter character \$ used as string starter/terminator, $\$ \notin \Sigma$.

Define *shingling* as the operation through which a string w can be transformed into a multiset S, that collects all contiguous substrings of a given length (denoted l), including repetitions. For example, shingling the string \$abaa\$ into length 2 shingles produces the multiset: $S = \{\$a, ab, ba, aa, a\$\}$. The multiset S is considered *uniquely decodable* if and only if there is exactly one string would produce S after shingling. In this case, S produced by shingling \$abaa\$ with $l = 2$ is **not** uniquely decodable since shingling an alternative string \$aaba\$ can produce S too. In other words, S is uniquely decodable iff there is exactly one possible string reconstruction from S.

3.3 Reconstruction Ambiguity in String Reconciliation

To efficiently reconcile remote (and probably similar) strings, one existing approach (Agarwal et al., 2006) is to translate the problem into one of set or multiset synchronization. The transformation is accomplished through *shingling*, wherein a string is divided into overlapping substrings (called *shingles*). Two remote strings are thus tranformed into set of shingles, which can then be handled by existing data synchronization techniques such as our CPI. A high-level description of a string reconciliation protocol is presented in Protocol 1. (First presented in (Kontorovich and Trachtenberg, 2012), with minor modifications)

Protocol 1: Reconciliation of remote strings σ and τ.

1. *Shingling*: Split σ into a set S_σ of length l shingles, similarly split τ into S_τ.
2. *Synchronization*: Reconcile set S_σ and S_τ.
3. *Unique Decodability Maintenance*: Check the unique decodability of local set S (S_σ or S_τ), if not, merge some shingles of S until unique decodable.
4. *Indices Exchange*: If any merge is required in Step 3, exchange the indices of merged shingles in the alphabetical ordering of S.
5. *Decode*: Uniquely reconstruct σ and τ from S_σ and S_τ on the remote hosts.

The main challenge of this protocol is in Step 3, wherein a shingle set is known

and from which a string can be reconstructed by piecing those shingles together. The problem is, for a given shingle set there may be more than one possible reconstructions exist. For example, the collection of shingles $\{\$a, ab, ba, aa, a\$\}$ can be combined into the string **abaa** or **aaba**.

To solve the reconstruction ambiguity, we next propose an algorithm that takes in an *ordered* list of shingles, detects possible ambiguities and fixes them by merging some of the shingles, then outputs an *unordered* set of shingles which is proved to be uniquely decodable, i.e. exactly one possible string can be reconstructed from the set.

3.4 De Bruijn Graph

A De Bruijn graph is a directed graph representing overlaps between sequences of symbols. The process of combining shingles of S back into a string involves the construction of a modified De Bruijn graph $G(S)$, in which each shingle of S corresponds to an edge the graph, with weight equal to the number of times that the shingle occurs in S, and vertices of $G(S)$ are the length $l-1$ prefixes (and suffixes since shingles are overlapping) of the shingles in S. We also define $\Psi(v)$ as the *parent edge* of a vertex $v \in G(S)$, representing the **first discovered** incoming edge of a vertex **in cycle**. Figure 3·1 provides a example De Bruijn graph representing the string \$**abaa**\$ (and also \$**aaba**\$) for $l = 2$, in which case $\Psi(b) = `ab'$, and $\Psi(a)$ could be either $`\$a'$ or $`aa'$, up to which is discovered first.

The following lemma was first proved in (Kontorovich, 2004) for bigrams.

Lemma 2. *A shingle set S is uniquely decodable iff there is exactly one Eulerian cycle in the corresponding De Bruijn graph $G(S)$ that starts and ends with* \$.

The proof can be extended to n-grams based on the knowledge of that there are still one-to-one correspondences between shingles in S and edges in $G(S)$ by

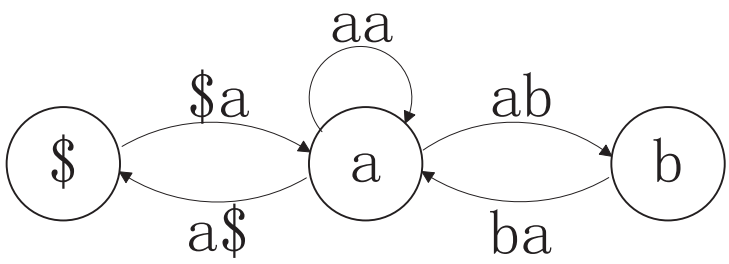


Figure 3·1: A De Bruijn graph corresponding to the string $abaa$, note that another Eulerian path $aaba$ can also be found in this graph.

construction.

3.5 Algorithms

We next present two algorithms that help maintainng the unique decodability of any shingle set produced by *shingling* a string, which can be used in Step 3 of **Protocol 1**.

3.5.1 Checking Unique Decodability

In this section we present Algorithm 1, which checks the unique decodability of a given length n string $w \in \Sigma_{\$}^{n}$ from its **ordered** list of shingles, by which we mean that the i-th shingle of w is denoted s_i.

```
Input: Ordered shingle set S = {s_1, s_2, s_3, ..., s_n} constructed from shingling
       string w with minimum shingle length l;
Output: true if S is uniquely decodable and false otherwise;
initialize the graph G(S) with vertex set V, each v_i ∈ V represents the length
  l − 1 prefix of s_i, v_i = v_j if s_i and s_j have the same prefix;
initialize each v ∈ V as UNVISITED;
initialize each v ∈ V as NOT IN CYCLE;
initialize each Ψ(v) as empty;
for i ← 1 to |S| do
    case 1: v_i is UNVISITED
        mark v_i as VISITED;
    endsw
    case 2: v_i is NOT IN CYCLE
        j ← i;
        repeat
            if v_j is NOT IN CYCLE then
                label v_j as IN CYCLE;
                Ψ(v_j) ← s_{j−1};
            end
            j ← j − 1;
        until v_j = v_i;
    endsw
    case 3: v_i is IN CYCLE
        if s_{i−1} = Ψ(v_i) then     /* stepping along an existing cycle */
            do nothing;
        else           /* intruding an cycle from a different edge */
            return false
        end
    endsw
end
return true
```

Algorithm 1: Checking the unique decodability of a shingle set

The following theorem establishes the correctness of Algorithm 1.

Theorem 5. *Algorithm 1 returns* ***true*** *iff its input set S is uniquely decodable.*

Proof. From Lemma 2 we know that to determine the unique decodability of S is equivalent to determining the existence of an unique Eulerian cycle in G that starts and ends with the special delimiter \$.

Completeness: Given an input set S that makes Algorithm 1 return **true**, what needs to be proved is that $G(S)$ has a unique Eulerian cycle. Assume that after S is processed by Algorithm 1 all the labels in $G(S)$ are fixed; we now restart from \$ along the Eulerian cycle to see if there were any opportunities to diverge from the cycle we found to porduce different Eulerian cycle in $G(S)$. During the traversal, there are four cases at any vertex v:

- **case 1**: v is labeled as **NOT IN CYCLE**
- **case 2**: v is labeled as **IN CYCLE** and has exactly one out-going edge;
- **case 3**: v is labeled as **IN CYCLE** and has two out-going edges;
- **case 4**: v is labeled as **IN CYCLE** and has more than two out-going edges;

In case 1, Algorithm 1 only visited v once, meaning that any traversal on G(S) must leave v along the only available edge. In case 2, since v has only one out-going edge, any traverse must leave v along the same edge. In case 3, there are two out-going edges of v. Suppose the traversal leaves v from one of the two edges first, denoted $e1$, and returns to v at some later point in order to traverse the second out-going edge, denoted $e2$. Note that by returning to v for the first time the traversal already forms a cycle, denoted C_{e1}, in which $e1$ is included while $e2$ is not. Were the traversal to leave on $e2$ and return to v again, it would cause an intrusion on C_{e1} and Algorithm 1 would return **false**. Bounded by this, any traversal to v must leave along $e1$ all but the last time, there is no opportunity to diverge from the existed cycle at v. In light of case 3, case 4 is therefore not possible since any path that leaves v along its second out-going edge is not allowed to return.

Soundness: To prove that Algorithm 1 returns **true** if its input set S is uniquely decodable is equivalent to proving that a shingle set S is **not** uniquely decodable if Algorithm 1 returns **false**.

Algorithm 1 only returns **false** when an intrusion on an existing cycle is detected at vertex v_x, at which time we know that: (i) v_x has been marked as **VISITED**,

so that the path between the last visit and the current visit forms a cycle. (ii) v_x is already in another cycle including its parent edge, which is necessarily different from the cycle just found in (i), since an intrusion is only detected when stepping onto v_x along an edge other than its recorded parent edge. Since v_x is in two different cycles that both return to v_x, there are at least two different Eulerian cycles on G(S) can be found based on which cycle is visited first. Lemma 2 tells us S is not uniquely decodable if $G(S)$ has two Eulerian cycles, and *Soundness* is proved. □

Algorithm 1 can determine the unique decodability of the length l shingling on a length n string in $\Theta(n) + |\Sigma|$ time. Analysis is provided in section 3.6.2.

3.5.2 Patching Unique Decodability

In cases where an unique decoding of a shingle set does not exist, Algorithm 2 provides method of merging some of the shingles in order to produce uniquely decodable shingle set that decodes to the same string. We name the checking and possible merging process as *patching* the unique decodability of a shingle set.

Algorithm 2 executes in almost the same way as Algorithm 1 to check the unique decodability of the input shingle set. We only change the boolean label **INCYCLE** in Algorithm 1 to a counter $\Phi(v)$, which keeps track of how many cycles (not necessarily distinct) that includes vertex v has been detected at the time. If the input shingle set fails to survive the check, Algorithm 2 makes use of procedure **deCycle** and its sub-procedure **mergePrevious**, in order to recover the unique decodability of the input shingle set by merging shingles and maintaining relevant metadata.

Input: *Ordered* shingle set $S = \{s_1, s_2, s_3, ..., s_n\}$ constructed from shingling string w with minimum shingle length l;
Output: Shingle set S' that decodes uniquely to w;

```
1  initialize the graph G(S) with vertex set V, each v_i ∈ V represents the length
   l − 1 prefix of s_i, v_i = v_j if s_i and s_j have the same prefix;
2  initialize each v ∈ V as UNVISITED;
3  initialize each Φ(v) = 0;
4  initialize each Ψ(v) as null;
5  initialize UD, the boolean flag indicating unique decodability, to be true;
6  i ← 1;
7  while i ≤ |S| do
8  |  case 1: v_i is UNVISITED
9  |  |  mark v_i as VISITED;
10 |  endsw
11 |  case 2: v_i is VISITED and Φ(v_i) = 0
12 |  |  j ← i;
13 |  |  repeat
14 |  |  |  if Φ(v_j) = 0 then
15 |  |  |  |  Ψ(v_j) ← s_{j−1};
16 |  |  |  end
17 |  |  |  Φ(v_j) ← Φ(v_j) + 1;
18 |  |  |  j ← j − 1;
19 |  |  until v_j = v_i;
20 |  endsw
21 |  case 3: v_i is VISITED and Φ(v_i) > 0
22 |  |  if s_{i−1} = Ψ(v_i) then      /* stepping along an existing cycle */
23 |  |  |  do nothing;
24 |  |  else            /* intruding an cycle from a different edge */
25 |  |  |  UD=false;
26 |  |  end
27 |  endsw
28 |  if UD=false then
29 |  |  (S, G, i) ← deCycle(S, G, i)      /* delete one cycle at v_i */;
30 |  |  UD←true;
31 |  end
32 end
33 i ← i + 1; return S
```

Algorithm 2: Patching the unique decodability of a shingle set

Procedure deCycle is called at line 29 of Algorithm 2, and its function is to delete

one cycle at v_i by merging all the edges backwards from current to just before the last occurence of v_i. As a sub-procedure of deCycle, mergePrevious, is called when one edge (s_{k-1}) needs to be merged with its previous edge (s_{k-2}), with different decisions being made at each merge, depending on v_k's state.

Input: S: shingle set; G: de Bruijn graph of S; i, index number of current vertex
Output: modified input (S, G, i), with updated state Ψ and Φ to reflect cycle deletion

```
1 k ← i;
2 repeat
3 |   k ← k − 1;
4 |   (S, G) ← mergePrevious(S, G, k);        /* merge s_k with s_{k−1} */;
5 until v_k = v_i;
6 delete s_k to s_{i−1} from S;
7 i ← k − 1;
8 return (S,G,i)
```

Procedure deCycle(S, G, i), deleting cycle by merging edges backwards from v_i until $\Psi(v_i)$ is merged once

Theorem 6. *The shingle set S' returned by Algorithm 2 is uniquely decodable.*

Lines 1 to 27 works in the same way as in Algorithm 1, and therefore when Algorithm 2 reaches Line 28, *UD*=**false** iff the shingle set seen so far is **NOT** uniquely decodable, and we develop the rest of the proof with the aid of Lemma 3.

Lemma 3. *When UD=**false** at Line 28 of Algorithm 2 for some index i, then*

- *1) when it next reaches Line 31, $\Phi(v_i)$ will be reduced by one, and v_i is involved in one fewer cycles;*
- *2) the next iteration of while loop (from Line 7) will restart at v_i;*
- *3) by the next time UD=**true** at Line 28 of Algorithm 2, the intruded cycle will be broken.*

```
Input: S: shingle set; G: de Bruijn graph of S; k, index number of current
       vertex
Output: modified input (S, G)
if Φ(v_k) = 0 then                                   /* v_k is not in cycle */
    mark v_k as UNVISITED;
else if Φ(v_k) = 1 then /* v_k is in exactly one cycle, this merge will
break the cycle */
    j ← k;
    repeat
        Φ(v_j) ← Φ(v_j) − 1;
        if Φ(v_j) = 0 then
            Ψ(v_j) ← null;
        end
        j ← j − 1;
    until v_j = v_k;
else   /* v_k is in more than one indistinct cycles, this merge will
reduce the number of cycles by 1 */
    Φ(v_k) ← Φ(v_k) − 1
end
Append the l-th to the last character of s_k to s_{k−1};
return (S,G)
```

Procedure mergePrevious(S, G, k), merging s_k with s_{k-1} and maintaing relevant metadata

Proof. We prove the three statements of Lemma 3 separately.

- 1) The function of procedure deCycle is to merge all edges encountered from just before the last occurence of v_i to v_i; more precisely, to merge shingles from s_{i-1} to s_{j-1}, where j is the largest index that satisfies $j < i$ and $v_j = v_i$. By assumption, Algorithm 2 detects an intrusion on an existing cycle before calling procedure deCycle, meaning that v_i already belong to a cycle, and it must be that s_{j-1} belongs to; as otherwise an intrusion would have been detected earlier. $\Phi(v_i)$ is reduced by one in procedure mergePrevious. Note that $\Phi(v_i)$ keeps track of how many cycles is v_i involved, for it is increased by one only when a new cycle at v_i is detected, and decreased by one only when $\Psi(v_i)$ is merged once.

- 2) Procedure deCycle is called when an intrusion is detected at vertex v_i, and it keeps merging shingles until it comes back tov_k, which is the last occurence of v_i. Procedure deCycle returns an index just before k, which is incremented on line 33. Therefore the next iteration restarts at v_k and $v_k = v_i$.

- 3) Statement 1) guarantees that $\Phi(v_i)$ is reduced by one at each iteration and statement 2) proves that the next iteration restarts at the same v_i. When $\Phi(v_i)$ is decreased to zero, indicating that the number of cycles that the current vertex v_i is involved is zero, the intruded cycle is hereby broken.

By considering the three terms together we can conclude that, when *UD*=**false** at Line 28 of Algorithm 2 for some index i, Algorithm 2 goes to break the intruded cycle by merging all the parent edges of v_i. □

proof of Theorem 6. From Theorem 5, we know that Algorithm 2 takes a shingle set as input, and detects whether it is uniquely decodable, more precisely, whether there is an intrusion on existing cycle(s) at the time. By Lemma 3, if *decode*=**false**, Algorithm 2 repeatedly breaks the intruded cycle(s) and restarts the check at the same

vertex, until all the intruded cycles that the current vertex is involved are broken, at which point UD=**true**. In essence, if the input set is it is not uniquely decodable, Algorithm 2 "patches" it by merging some of the shingles together, and such merging cannot increase the number of decoding needed to reconstruct the string, for all the merges are designed to take place on existing cycles and therefore cannot introduce new cycles during the patching process. After the patching, Algorithm 2 always exits with UD=**true**, indicating that it always returns an uniquely decodable set.

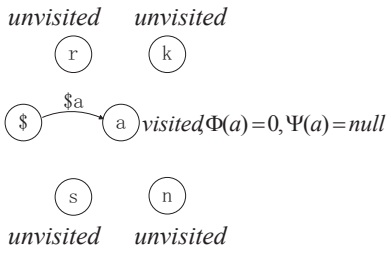
□

3.5.3 Example

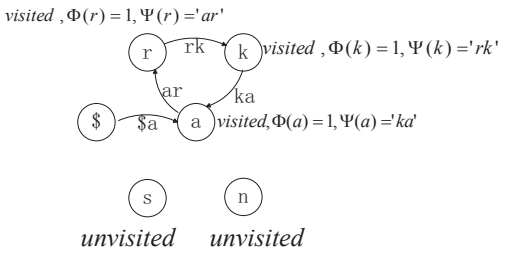
Figure 3·2 demonstrates Algorithm 2's processing flow on the string "arkansas" with shingling length $l = 2$. By design the input for Algorithm 2 is the shingle set:

$$S = \{\$a, ar, rk, ka, an, ns, sa, as, s\$\}. \tag{3.1}$$

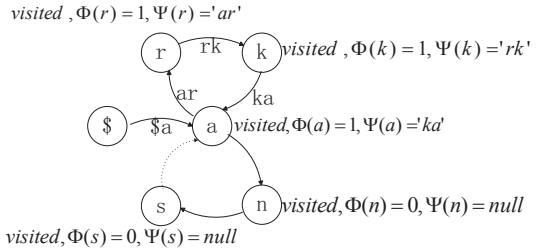
By visiting one vertex at each step, Algorithm 2 checks the unique decodability of the shingle (sub)set seen so far. If the current shingle (sub)set fails the unique decodability check, Algorithm 2 merges some of the shingles and restarts the check. Figure 3.2(a) shows the De Bruijn graph $G(S)$ when the vertex a is visited for the first time (Line 9). Figure 3.2(b) shows the state of $G(S)$ when Algorithm 2 detects a new loop at a (Line 20), note that not only the vertex a but also vertices r and k have their Φ and Ψ changed. Algorithm 2 detects an intrusion on an existing cycle in Figure 3.2(c), when it visits vertex a from an edge (sa) which is different from $\Psi(a)$ (Line 27). After the intrusion is detected, procedure deCycle and mergePrevious are called by the algorithm to merge all the shingles back to k, figure 3.2(d) show the state of $G(S)$ at Line 31, in the same iteration that the intrusion is detected. After the merge, Algorithm 2 continues to visit vertex a "again", but this time from a newly



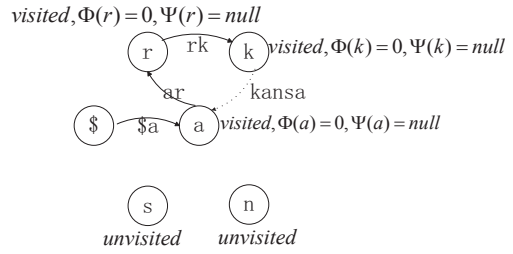
(a) string processed = 'a', $S(1 : i-1) = \{\$a\}$, $v_i = a$



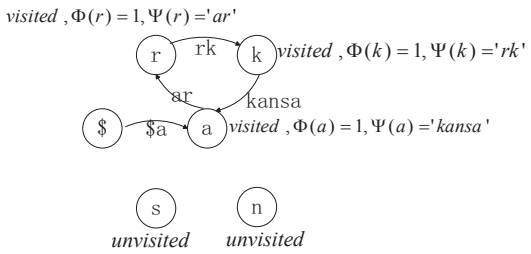
(b) string processed = 'arka', $S(1 : i-1) = \{\$a, ar, rk, ka\}$, $v_i = a$



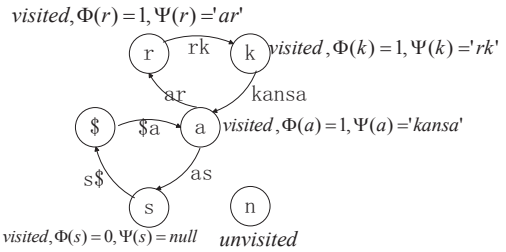
(c) string processed = 'arkansa', $S(1 : i-1) = \{\$a, ar, rk, ka, an, ns, sa\}$, $v_i = a$



(d) string processed = 'ark', $S(1 : i) = \{\$a, ar, rk, kansa\}$, $v_i = k$



(e) string processed = 'arkansa', $S(1 : i-1) = \{\$a, ar, rk, kansa\}$, $v_i = a$



(f) string processed = 'arkansas', $S = \{\$a, ar, rk, kansa, as, s\$\}$

Figure 3·2: Demonstration of Algorithm 2 on the string "arkansas" with $l = 2$ and delimiter \$

merged edge (*kansa*), the state of $G(S)$ at Line 20 is demonstrated by Figure 3.2(e). The algorithmcontinues without detecting another intrusion, finishing the De Bruijn graph $G(S)$ as shown in Figure 3.2(f).

3.6 Analysis

3.6.1 Data Structure

We can use an *array* and a double-linked *list* to store the vertex information. A two-dimensional array can be used to store the state information of all vertices. The rows of this array are indexed by vertex number, for each row representing some vertex v, it contains the state information of v such as the **VISITED** boolean, and values $\Phi(v)$ and $\Psi(v)$. The total number of rows of the array is $|\Sigma|^l$, and the number of non-zero rows is at $n - l - 1$ (excluding shingles that contain the delimiter), in practice, it is common that $n \ll |\Sigma|^l$.

Both Algorithm 1 and 2 take an *ordered* shingle set as input, and we use a double-linked *list* to store all these input shingles in order of occurence. The i-th element of the list will be denoted L_i, and by design, $L_i = s_i$.

3.6.2 Runtime Analysis: Algorithm 1

Theorem 7. *Algorithm 1 has* $\Theta(|\Sigma|)$ *preprocessing time complexity and* $\Theta(n)$ *on-line time complexity.*

Proof. We list the detailed run time nalysis as below.

- Lines 1-4. Initilization of De Bruijn graph G and its vertex set V, can be accomplished in constant time with sparse storage, with the two-dimensional array implementation described in Section 3.6.1. Note that for G only vertices need to be stored in the array while edges are essentially the input shingles, which are already kept in another list.

- Lines 6-8. Since the array containing the state information of vertices has constant time access, the time cost of this step is constant.

- Lines 9-18. All the input vertices are kept in an order *list* (see Section 3.6.1), and the iteration at lines 11-17 can then be accomplished by scanning backwards through the list. Since l is constant, an operation like comparision, searching or reading/writing can be done in constant time.

- Lines 19-25. Comparing shingles of length l takes constant time, again because l is constant.

□

3.6.3 Runtime Analysis: Algorithm 2

Theorem 8. *Algorithm 2 has linear time complexity* $\Theta(n) + |\Sigma|$ *running on string* w *of length* n*.*

Proof. Details are as followed:

- Lines 1-29. Though more metadata need to be maintained compared to Algorithm 1, all the operations can still be accomplished in constant time.

- Procedure deCycle. The cost of merging two shingles with length-l overlap is constant, because l is assumed constant. In the worst-case, all the shingles are merged together and the output shingle set S is uniquely decodable by itself, the sequence of $n - l + 1$ merges takes $\Theta(n)$ time in *aggregation*. Therefore, the *amortized* cost per call of procedure deCycle is $\Theta(n)/n$.

□

3.7 Experiments

Though it is hard to give precise bounds on the number of shingles that needed to be merged for transforming a set S into uniquely decodable. the work in (Agarwal et al., 2006) provides some hints in estimating the "safe" length of shingling random bit-strings into uniquely decodable set without additional merges. Specifically, for length-n Bernoulli string (strings of n random bits in which each bit is 0 with probability $p > 0.5$), it can expected that each node in the corresponding De Bruijn graph of length l shingles to have only one outgoing edge if

$$l \geq n + 1 + \frac{W(-\ln(p)p^{-n})}{\ln p} \tag{3.2}$$

where $W(\cdot)$ is the Lambert W function (Corless et al., 1996) (also called the Omega function or product logarithm). When n goes to infinity, then (3.2) is $O(logn)$, implying that the minimum length of shingling needs to be sized logarithmically with the string length in order to avoid high-frequency merges.

3.7.1 Bernoulli String

We next consider Bernoulli String, which are random collections of bits, where the probability of seeing 1 at any given index is p. Figure 3·3 shows the average number of merges needed for transforming a length 1000 Bernoulli string with bit probability p into a unique decodable shingle set using Algorithm 2. The corresponding vertical dotted line is the "safe" length given by (3.2), shingling a Bernoulli string in question using a length larger than the safe length is expected to get a unique decodable set naturally (without any additional merge).

Figure 3·4 and Figure 3·5 demonstrate the same trend as in Figure 3·3. An intersting fact can be found that when the length used for shingling the original string is multiple of the length of certain substring appearing with high frequency, (in

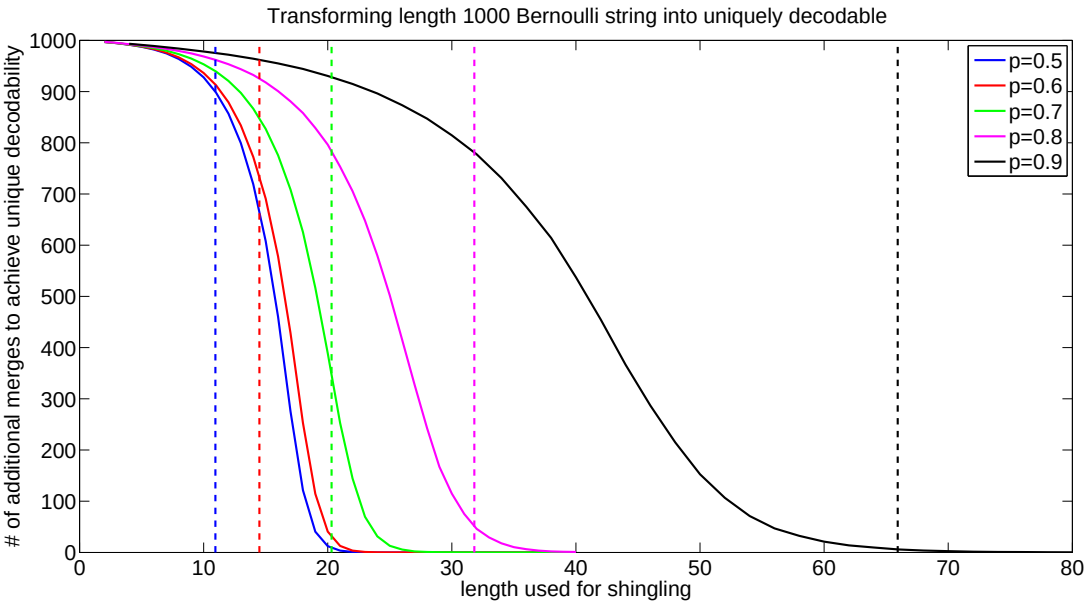


Figure 3·3: Average number of merges needed to transform a length 1000 Bernoulli string to unique decodable shingle set, on varying shingle length.

this case, when the original shingle length is multiple of 10), the generated shingle set tends to be more difficult to be uniquely decoded. This result may provide us a good reference on choosing appropriate length for shingling the original string.

3.7.2 English Text

Figure 3·6 illustrates the average number of merges needed for transforming length 2000 English text into unique decodable shingle set using algorithm 2, together with the cumulative distribution function for natrually unique decoding on shingle length l. Compared with results got from Bernoulli string, apprently determing unique decodability of English texts is more complicated than that of bitstrings, for English text has a much larger cardinality of its alphabet. The 2000-character English texts we use in our experiment are all produced by spliting popular English books which are publically available at (http://www.gutenberg.org) into paragraphs of 2000 consecutive

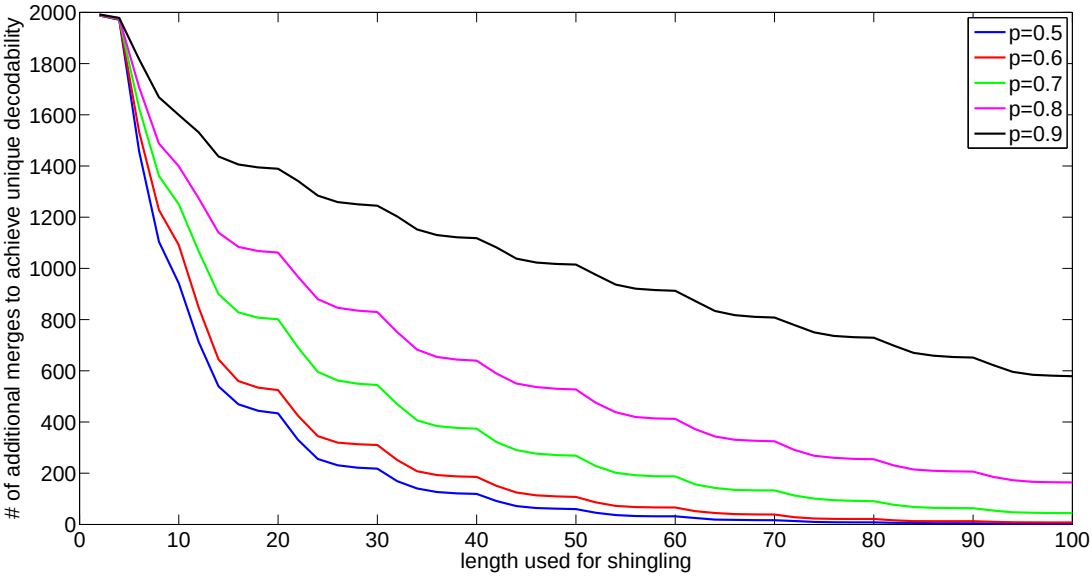


Figure 3·4: Average number of merges needed to uniquely decode a length 1000 Markovian bit string with a length-10 substring appear with probability p.

characters, the total number of such paragraphs is around 20,000.

Figure 3·7 gives a sense of how users of our string reconciliation protocol can benefit from properly choosing a length for shingling the set for transmission. While the English text result is based on the same testbed as in Figure 3·6, and the Code files to be reconciled are two lateset versions of Ubuntus operation system (v12.10 and v12.04). By design, the communication complexity of our string reconciliation protocol grows linearly to the length used for shingling when a reconciliation is carried on between with empty set, whose result seems to be dominated by the overall size of the shingle set, the lower half implies an intersting fact that when the two sets to be reconciled do not differ too much (in this case, 5%), the benifit brought by increasing the length used for shingling to save times of merges may outweight the expense it introduced to the overall communication complexity.

Figure 3·8 illustrates the experiment results on a much broader testbed (100000

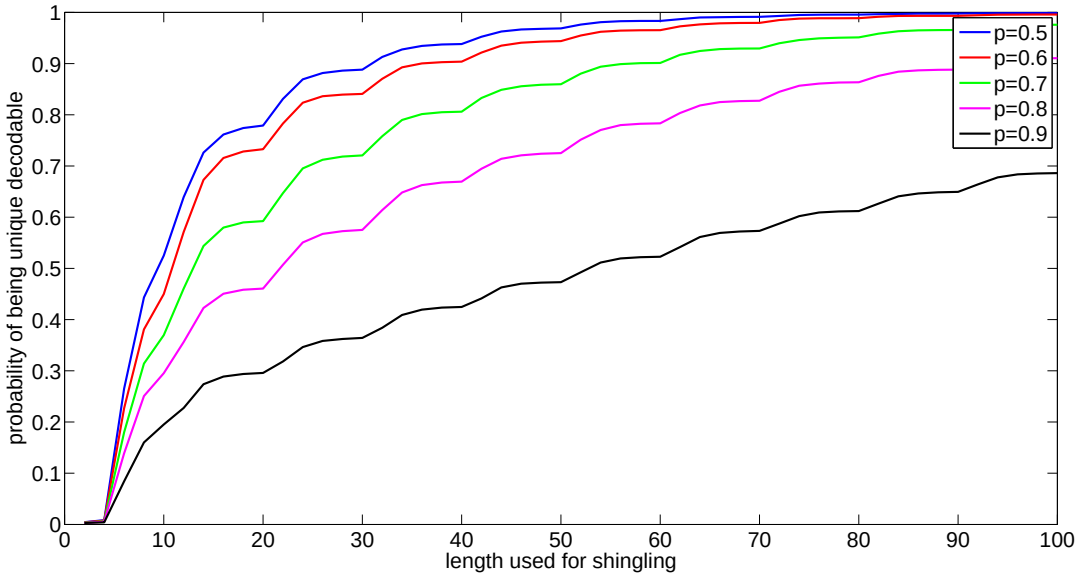


Figure 3·5: Cumulative distribution for a length 1000 markovian bit string with a length-10 substring appear with probability p being naturally unique decodable, on varying shingle lednth.

consecutive characters taken from English text), from which we can tell the same trend as shown in Figure 3·6 and 3·7. The number of merges needed for unique decoding still decreases most dramatically at a relatively small shingling length (before the long tail), correspondingly, the communication complexity of the protocol stays low within that range.

Figure 3·9 demonstrates the online time used for running Algorithm 2 on fixed-length strings with varying shingling length, and on varying-length strings with fixed-length of shingling. In the experiments the online time of Algorithm 2 grows linearly in the length of shingling (above) and also in the length of string (below), the experiment data is collected by averaging at least 5 samples at each point. The results support our analysis made in section 3.6.3.

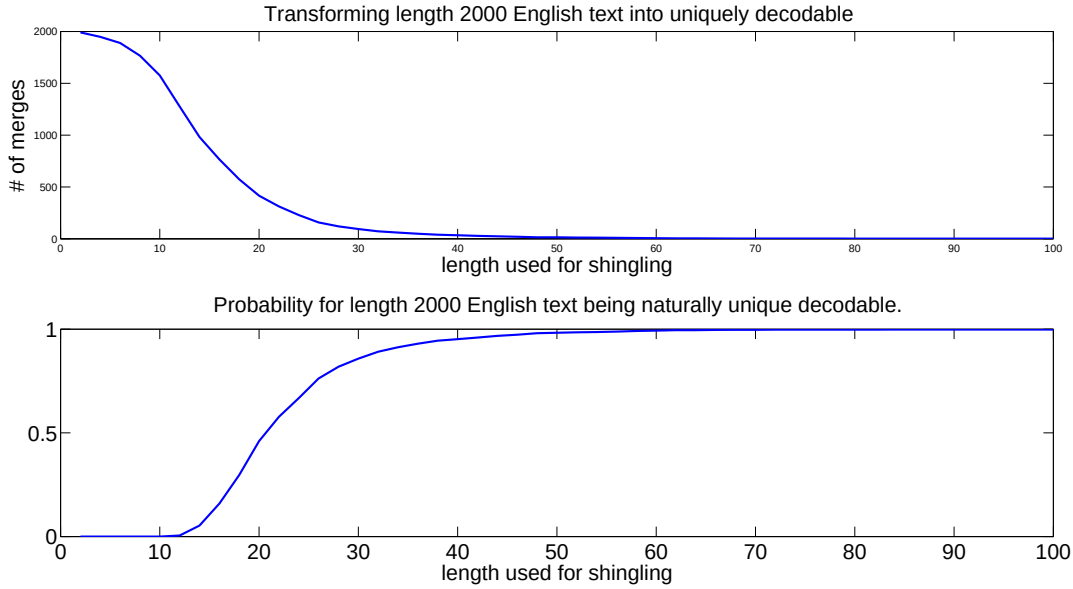


Figure 3·6: Average number of merges needed to transform 2000 consecutive characters taken from English texts to unique decodable, on varying shingle lengthes, along with the amount of communication bits.

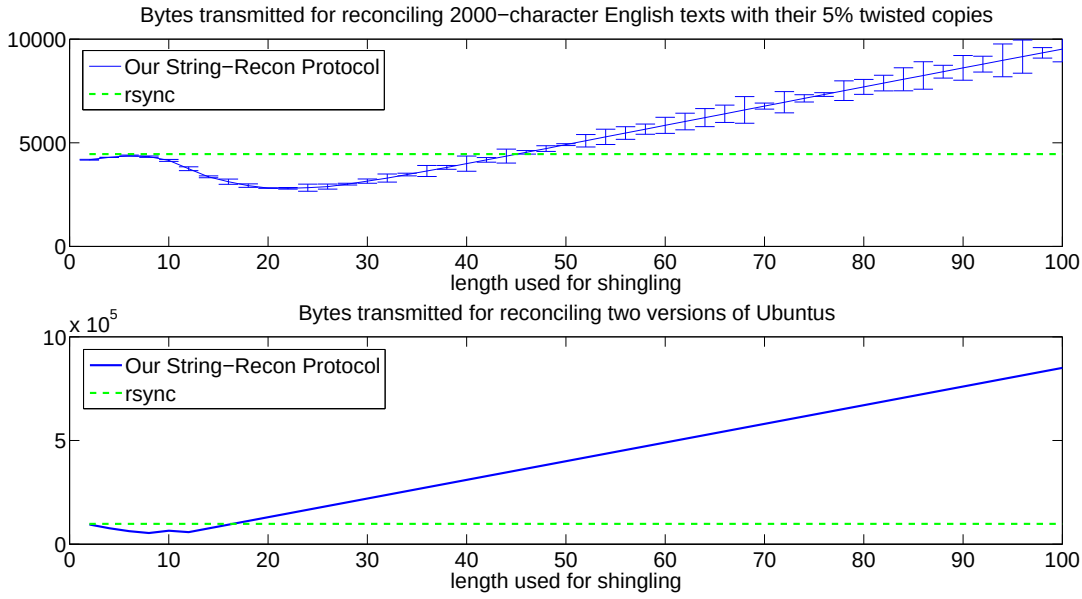


Figure 3·7: Comparison with rsync of reconciliation tasks on English text and file.

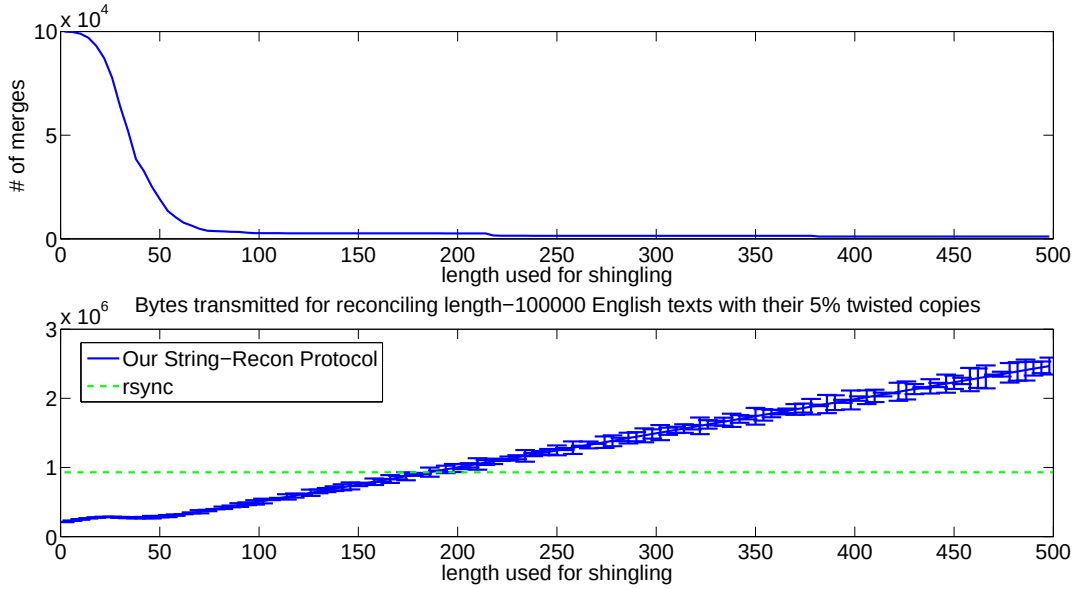


Figure 3·8: Reconciling length 100000 English texts.

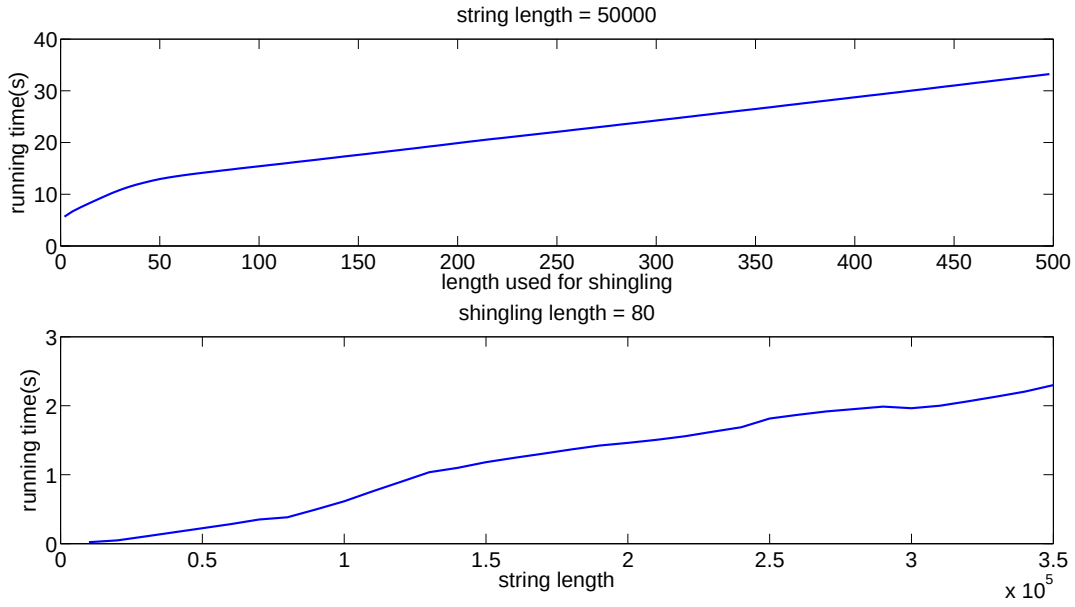


Figure 3·9: Computation complexity: Online time (in seconds) of Algorithm 2 on varying shingle/string length.

Chapter 4

An application to Delay Tolerant Networks

In this chapter, we present numerical results illustrating the performance of P-CPI into an existing DTN routing protocol RAPID (Balasubramanian et al., 2007) and report the improvements observed from simulations. We use RAPID as a proof of concept because it is DARPA-supported and has been shown to compare favorably to several other DTN routing protocols.

4.1 The RAPID Routing Protocol

The RAPID protocol routes a packet from a source to its destination by opportunistically replicating it until a copy reaches the destination. During each peer-to-peer communication, peers first exchange the metadata for all packets. Thus, for each packet i, RAPID maintains a list of nodes that carry the replica of i and, for each replica, an estimated time for direct delivery. Based on the metadata information, the nodes decide on whether or not to replicate a packet. Specifically, in the decision-making process, a node estimates the "expected benefit" of replicating a packet against not doing so, and different policies are used with regard to different benefits, such as minimizing average delay or maximizing delivery rate. Packets are then sent in decreasing order of estimated benefit.

The protocol executes symmetrically when two nodes X, Y are within radio range and have discovered one another. Below, we provide a pseudo-code for the protocol,

as run on node X.

Protocol RAPID(X, Y) (Balasubramanian et al., 2007):

1. *Synchronization:* Synchronize metadata with Y about packets in local buffer and metadata collected over past meetings.
2. *Direct delivery:* Deliver packets destined to Y.
3. *Replication:* For each packet i in node X's buffer
   (a) if i is already in Y's buffer (as determined from the metadata), ignore i.
   (b) Estimate benefit, δU_i, of replicating i to Y.
   (c) Replicate packets in decreasing order of δU_i
4. *Termination*: End transfer when out of radio range or all packets replicated.

In RAPID protocol, at the beginning of each peer-to-peer communication, two nodes exchange metadata about the packets by sync, including information about delivered packets, locally stored packets and other packets (that may "heard" from other nodes during past communications). Then, each node makes dynamic decisions on every locally stored packet, about whether to replicate it or not, based on the metadata collected so far from every other node it has communicated before. More precisely, in the decision-making process, a node estimates the expected benefit of replicating a packet against not doing so, and different policies are used with regard to different benefits, such as minimizing average delay or maximizing delivery rate.

4.2 Modified RAPID with P-CPI Deployed

As pointed in (Balasubramanian et al., 2007), RAPID requires a full synchronization of metadata ahead of any packet transmission. When the size of the metadata is large, this approach bears the risk of limiting the amount of useful information (i.e., packets) that can be exchanged at a meeting between two nodes. The authors of (Balasubramanian et al., 2007) left this problem open for future work.

P-CPI provides a good strategy to address this problem. Using P-CPI, metadata and packets with high priority are sent first. Then, if the link is still available,

lower priority information is carried over the link. Therefore, we propose a modified version of RAPID with P-CPI deployed. In the modified protocol RAPID-PCPI, the estimated benefit of replicating a certain packet is calculated by the same metadata-based utility function as in RAPID and the high or low priority of a packet is then determined by a threshold of expected benefit. Packets with higher expected benefit evaluations are assigned high priority while the rest are arranged to go with low priority. In our implementation, the numbers of low and high priority packets are set equal. However, the fraction of high priority packets can also be made application dependent and dynamic, if desired. We also note that a packet and its replicas could be assigned different priorities at different nodes. However, a higher level protocol can detect this when metadata for the packet are reconciled and avoid the need of transmitting the packet itself.

Protocol RAPID-PCPI(X, Y):

0. *Initialization:* Establish the partition tree of packets with binary priorities at node X and Y.
1. *Synchronization (high priority):* Synchronize metadata of packets of high priority with Y.
2. *Packet Delivery (high priority):* Deliver/Replicate packets of high priority.
3. *Synchronization (low priority):* Synchronize metadata of packets of low priority with Y.
4. *Packet Delivery (low priority):* Deliver/Replicate packets of low priority.
5. *Termination*: End transfer when out of radio range or all packets replicated.

4.3 Performance Evaluation

We next illustrate through simulation the performance of RAPID, with and without P-CPI being deployed. As a baseline, we assume that the original RAPID protocol uses an "oracle" (which is unfeasible in practice) to determine in advance the list of

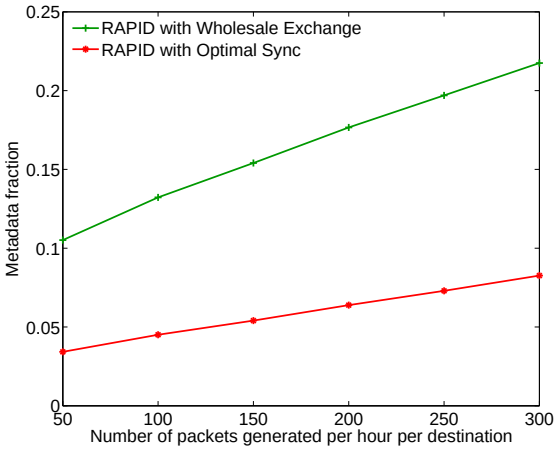


Figure 4·1: Fraction of metadata versus packet generation rate

differing metadata entries. We refer to this approach as *Optimal Sync*. As a baseline, we also consider the case where RAPID uses *Wholesale Exchange*, i.e., exchanges metadata in wholesale, during each synchronization. The simulations are conducted with the RAPID simulator developed by the authors of (Balasubramanian et al., 2007), the mobility model in this paper is also named DieselNet.

The default parameters for the simulation are listed in Table 4.1, most of which are the same with simulation setup in (Balasubramanian et al., 2007).

Number of nodes	max of 40
Buffer size	400 MB
Average transfer opp. size	given by real transfers among buses
Duration	19 hours each trace
Size of a packet	1 KB
Average bandwidth	400 Kb/s
Packet generation interval	1 hour
Optimization metric	average delay
Number of priority levels	2

Table 4.1: Experiment parameters

The metrics we use to evaluate the system performance include average delay, delivery rate, metadata fraction and communication overhead. More specifically, the

delay of a delivered packet is defined to be the duration from its generation to delivery and the delay of undelivered packet is the duration from its generation to the end of simulation.

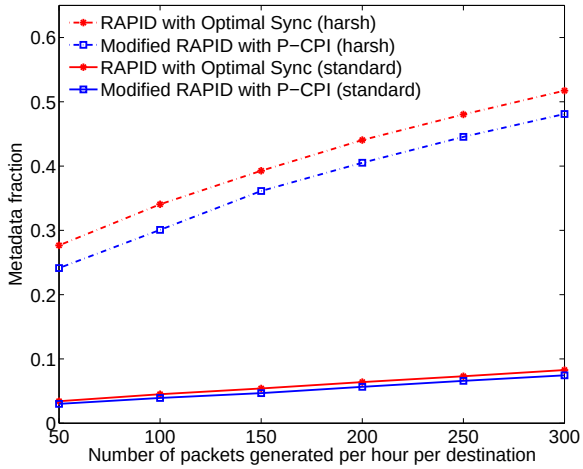


Figure 4·2: Fraction of exchanged metadata to all data sent in the network

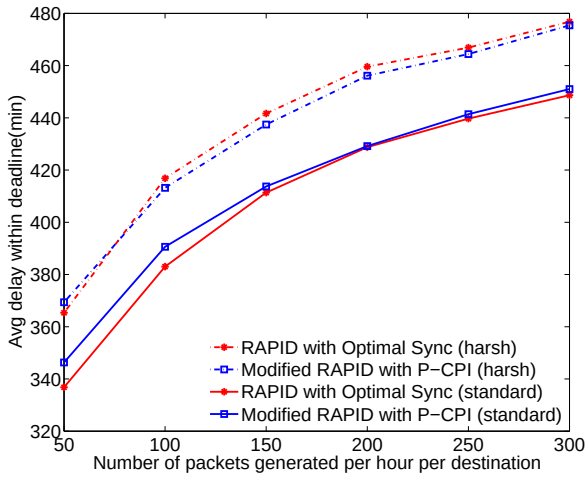


Figure 4·3: Average packet delivery time: delay from packet generation till delivery or end of simulation, no delivery deadline

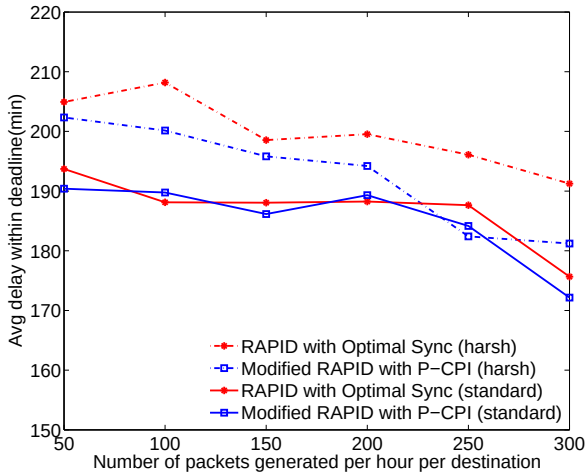


Figure 4·4: Average packet delivery time: delay from packet generation till delivery or expiration with delivery deadline of 10000 s.

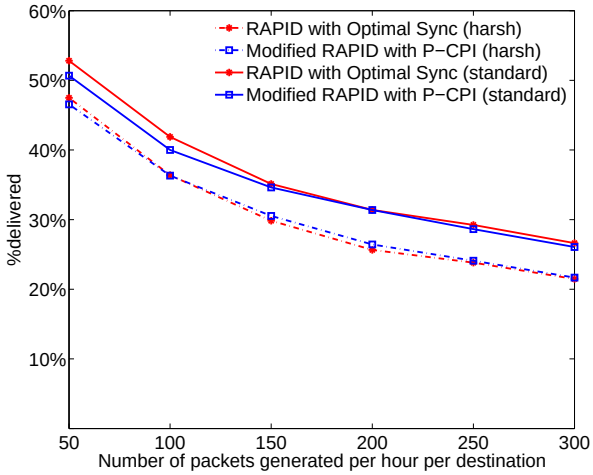


Figure 4·5: Percentage of packets delivered before expiration with delivery deadline of 10000 s

We first run simulations using the default settings shown in Table 4.1. Figure 4·1 shows the metadata fraction, i.e., the ratio of exchanged metadata to all exchanged data in the network, as a function of the packet generation rate. The figure illustrates the problem left open by the authors of (Balasubramanian et al., 2007), exchange of

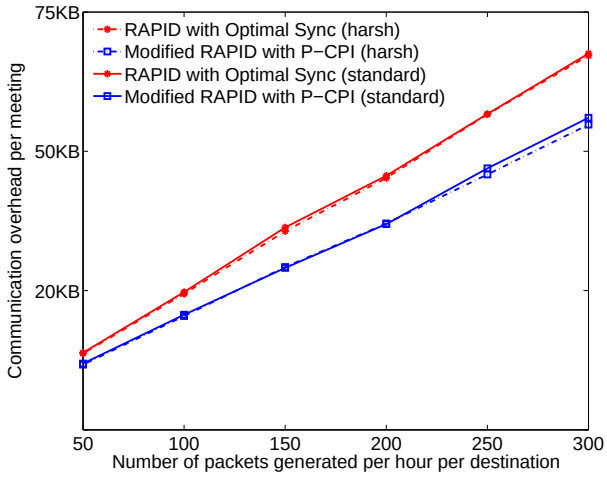


Figure 4·6: Average communication overhead per meeting

metadata could potentially jam the network traffic (and an optimal synchronization protocol can effectively reduce this effect). In fact, at high packet generation rate, RAPID with P-CPI (which is proved to be nearly-optimal in communication) yields about a three-fold reduction in the metadata fraction.

We present P-CPI as a practical synchronization middleware in DTN routing protocols such like RAPID. We next show the detailed comparison between RAPID using Optimal Sync and P-CPI and the additional benefits brought by prioritization. We run simulations in two different scenarios, a *standard* scenario which inherits the default settings in Table 4.1, and another *harsh* scenario with the average packet size set to 100 bytes and the average bandwidth set to 40 Kb/s, the result is shown in Figure 4·2 to 4·6.

Figure 4·2 shows that the metadata overhead becomes much more significant in such situations. As a result, the metadata reduction achieved with P-CPI results in major performance gains. The reason P-CPI performs even better than Optimal Sync is that Optimal Sync initially performs an optimal synchronization of the entire metadata, while P-CPI first synchronizes high priority metadata and packets and then, if

time is still available, synchronizes low priority metadata and packets. This reduction in the metadata fraction can lead to an improvement of about 5% in the percentage of packets delivered. Thus, Figure 4·3 shows that P-CPI stays closed to optimal with standard settings in the average delivery time (delay), and further reduces it and outperforms the Optimal Sync in harsh scenario. Figure 4·4 shows that the the average delay of delivered packets within deadline of P-CPI is better than Optimal Sync, which is more apparent under resource-constrained conditions. Similarly, Figure 4·5 indicates corresponding improvement for the percentage of packets delivered within the given deadline. Figure 4·6 demonstrates the effectiveness of P-CPI in sense of communication overhead. Again the reason that P-CPI is better than Optimal Sync is P-CPI sends not all but only part of the metadata at first. The simulation results of Figure 4·2 to 4·6 demonstrate that the newly proposed modified RAPID with P-CPI performs nearly the same as the original RAPID with Optimal Sync with standard settings and even better in harsh scenario.

Chapter 5

Conclusion

5.1 Conclusion

In this thesis, we have introduced, analyzed, and simulated a new synchronization algorithm, called Priority-based Characteristic Polynomial Interpolation (P-CPI), together with its applications for DTNs and string reconciliation protocol. In the algorithmic part, we have provided novel worst-case and high-probability performance analysis of the computation and communication complexity of P-CPI. Our simulations demonstrated that the computation and communication complexity of P-CPI grows close to linearly with the number of differences, depending only weakly (i.e., logarithmically) on the number of elements in the sets. These complexities also scale proportionally to the desired priority ratio. We have also proven that P-CPI can be stopped and quickly restarted at any time without incurring any extra memory overhead, a feature that is particularly useful in networks with a large number of nodes.

In addition to the analysis, we have demonstrated the practical benefit of using P-CPI in a DTN setting by implementing it as a synchronization conduit for the well-established *RAPID* DTN routing protocol. Simulations obtained using the original RAPID simulator show that P-CPI leads to significant reduction in the communication overhead of metadata compared to a simple wholesale transfer approach, hence solving a problem left open by the authors of RAPID. As a practical approach to synchronization, P-CPI yields near optimal performance in the average delivery time

of packets and other related metrics. We expect that P-CPI could serve as an effective synchronization middleware for other DTN routing protocols besides RAPID, an interesting area left for future work.

We have also considered string reconciliation as another possible application of our algorithm, in this thesis, we have developed a new string reconciliation protocol using shingling with our algorithm implemented. In the analysis we have shown that our algorithm for testing and patching the unique decodability of the string has a time complexity linear to the length of the string. Our experiment results also show that our string reconciliation protocol can potentially outperforms existing synchronization tools such like rsync in some cases.

For future work, it would be interesting to obtain the theoretical analysis for the string reconciliation protocol proposed in Chapter 3, and a further data collection can be done by applying P-CPI as synchronization middleware in routing protocols other than RAPID in Chapter 4.

References

Abdel-Ghaffar, K. and Abbadi, A. E. (1994). An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, 5:87–93.

Agarwal, S., Chauhan, V., and Trachtenberg, A. (2006). Bandwidth efficient string reconciliation using puzzles. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1217–1225.

Agarwal, S., Starobinski, D., and Trachtenberg, A. (2002). On the scalability of data synchronization protocols for pdas and mobile devices. *IEEE Network*, 16:22–28.

Allavena, A., Demers, A., and Hopcroft, J. E. (2005). Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 292–301, New York, NY, USA. ACM.

Balasubramanian, A., Levine, B. N., and Venkataramani, A. (2007). DTN routing as a resource allocation problem. In *Proceedings, ACM SIGCOMM*, pages 373–384.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426.

Broder, A. Z. (1997). On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, pages 21–29. IEEE Computer Society.

Chaisson, M., Pevzner, P., and Tang, H. (2004). Fragment assembly with short reads. *Bioinformatics*, 20:2067–2074.

Corless, R., Gonnet, G., Hare, D., Jeffrey, D., and Knuth, D. (1996). On the lambert w function. *Advances in Computational Mathematics*, 5:329–359.

Cormode, G., Paterson, M., Sahinalp, S. C., and Vishkin, U. (2000). Communication complexity of document exchange. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 197–206, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Dodis, Y., Ostrovsky, R., Reyzin, L., and Smith, A. (2008). Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139.

Dubhashi, D. P. and Panconesi, A. (2009). *Concentration of Measure for the Analysis of Randomised Algorithms*. Cambridge University Press. pp. 46-47.

Eppstein, D., Goodrich, M. T., Uyeda, F., and Varghese, G. (2011). What's the differences? efficient set reconciliation without prior context. In *SIGCOMM'11*, Toronto, Ontario, Canada.

Eshel, M., Haskin, R., Hildebrand, D., Naik, M., Schmuck, F., and Tewari, R. (2010). Panache: a parallel file system cache for global file access. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 12–12, Berkeley, CA, USA. USENIX Association.

Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34.

hong Lim, B. and Agarwal, A. (1991). Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11:253–294.

Jain, S., Demmer, M., Patra, R., and Fall, K. (2005). Using redundancy to cope with failures in a delay tolerant network. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 109–120.

Jain, S., Fall, K., and Patra, R. (2004). Routing in a delay tolerant network. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 145–158.

Kontorovich, A. and Trachtenberg, A. (2012). Efficiently decoding strings from their shingles. *CoRR*, abs/1204.3293.

Kontorovich, L. (2004). Uniquely decodable n-gram embeddings. *Theoretical Computer Science*, 329(13):271 – 284.

Levenshtein., V. (1965). Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1(1):8–17.

Li, N., Zhang, Y., Hu, J., and Nie, Z. (2011). Synchronization for general complex dynamical networks with sampled-data. *Neurocomputing*, 74:805–811.

Lindgren, A., Doria, A., and Schelén, O. (2003). Probabilistic routing in intermittently connected networks. *SIGMOBILE Mobile Computing and Communications Review*, 7:19–20.

Lindholm, T., Kangasharju, J., and Tarkoma, S. (2009). Syxaw: Data synchronization middleware for the mobile web. *Mobile Networks and Applications*, 14:661–676.

Ma, N. and Ishwar, P. (2008). Two-terminal distributed source coding with alternating messages for function computation. In *IEEE International Symposium on Information Theory, 2008.*, pages 51 –55.

MacDonald, J. P. (2000). File system support for delta compression. Master's thesis, University of California at Berkeley.

Menouar, H., Filali, F., and Lenardi, M. (2006). A survey and qualitative analysis of mac protocols for vehicular adhoc networks. *IEEE Internet Computing*, 13(5):30–35.

Minsky, Y. and Trachtenberg, A. (2002). Scalable set reconciliation. In *40th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL.

Minsky, Y., Trachtenberg, A., and Zippel, R. (2003). Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49.

Orlitsky, A. (1991). Interactive communication: balanced distributions, correlated files, and average-case complexity. In *Proceedings, 32nd Annual Symposium on Foundations of Computer Science, 1991.*, pages 228 –238.

Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G. (1981). Locus a network transparent, high reliability distributed system. *SIGOPS Operating Systems Review*, 15(5):169–177.

Ramanathan, R., Hansen, R., Basu, P., Rosales-Hain, R., and Krishnan, R. (2007). Prioritized epidemic routing for opportunistic networks. In *Proceedings of the 1st international MobiSys workshop on Mobile opportunistic networking*, MobiOpp '07, pages 62–66, New York, NY, USA. ACM.

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459.

Schwarz, T., Bowdidge, R., and Burkhard, W. (1990). Low cost comparisons of file copies. In *10th International Conference on Distributed Computing Systems, 1990.*, pages 196 –202.

Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185.

Shi, X., Xie, H., Zhang, S., and Hao, B. (2007). Decomposition and reconstruction of protein sequences: The problem of uniqueness and factorizable langauge. *Korean Physical Society*, 50(1):118–123.

Starobinski, D., Trachtenberg, A., and Agarwal, S. (2003). Efficient pda synchronization. *IEEE Transactions on Mobile Computing*, 2(1):41–50.

Suel, T. and Memon, N. Algorithms for delta compression and remote file synchronization. In Sayood, K., editor, *Lossless Compression Handbook*.

Teodosiu, D., Bjorner, N., Gurevich, Y., Manasse, M., and Porkka, J. (2006). Optimizing file replication over limited-bandwidth networks using remote differential compression. Technical report, Microsoft Research.

Tridgell, A. (2000). *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University.

Yang, H., Yang, P., Lu, P., and Wang, Z. (2008). A syncml middleware-based solution for pervasive relational data synchronization. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 308–319, Berlin, Heidelberg. Springer-Verlag.

JIAXI JIN

No. 7 XiangGangDong Rd. 13#2-401
Qingdao, Shandong, 266071 CHINA
Mobile: +1(617) 513-2209
Email: jin@bu.edu
Year of Birth: 1987

EDUCATION

Boston University, Electrical & Computer Engineering Department

Master of Science, Computer Engineering.
Boston, USA Sep. 09 - Jan. 13

Donghua University, Department of Electrical & Computer Engineering

Bachelor of Engineering, Electrical Engineering. **Graduate of Honor**
Shanghai, China Aug. 05 - July 09

Academic Scholarships:
DHU Superior Scholarship (Top 5%) 2006 - 2008
Qian Zhiguang Scholarship (Top 1%) 2007

EXPERIENCE

Research Assistant of NISlab at BU Boston, USA Apr. 10 - present
Analyze and optimize data synchronization algorithm for PDA and PCs;
Implement synchronization software with high proficiency of C++/Java.

Teaching Fellow of BU Boston, USA Sep. 09 - Apr. 10
Run discussions and hold lab/office 20h/week for Software Engineering.

Research Assistant of RFIC lab at DHU Shanghai, China Sep. 07`- Apr. 09
Take active part in weekly discussion and lab routine.

Publications

Jiaxi Jin, Wei Si, David Starobinski, Ari Trachtenberg. *Prioritized Data Synchronization for Disruption Tolerant Networks*, Accepted, MILCOM 2012, Orlando, FL, USA

ACTIVITIES AND AWARDS

Basketball: Champion of Songjiang Higher Education Mega League
Mar. 2007

Student Representative at both Opening Ceremony and Commencement Day
Sep. 2005, June 2009

Grade President of Electrical & Electronic Engineering; Sep. 2006 – July 2008

DHU Student Leader of Excellence (1%); Sep. 2006, Sep. 2008

DHU Student of Excellence (5%) Sep. 2007

Volunteer Interpreter in DHU International Cooperation Office
Nov. 2005 – Mar. 2007