InterLocal: Integrity and Replication Guaranteed Locality-based Peer-to-Peer Storage System

by

Adilet Kachkeev

A Thesis Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

July 25, 2013

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Adilet Kachkeev

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Öznur Özkasap (Advisor)

_____

Assist. Prof. Alptekin Küpçü (Advisor)

_____

Prof. Özgür Barış Akan

_____

Assoc. Prof. Mine Çağlar

_____

Assoc. Prof. Yücel Yemez

Date: _____

To my family.

ABSTRACT

Trend in computer storage flows from possessing data locally to data outsourcing. Although users tend to store data at cloud or peer-to-peer storage systems, they also require guarantees about the security of data. A key requirement is the ability to check integrity of the files without downloading them and make necessary updates. In case of peer-to-peer storage systems, it is also desirable to place files at the nodes physically close to the data owner for minimal response time and efficient access.

In the first part of this thesis, we implement and examine a system based on Dynamic Provable Data Possession (DPDP) model. We present an optimized data structure based on skip lists called FlexList and its advantages over other data structures. We then propose FlexDPDP: a complete dynamic provable data possession system employing FlexList. Furthermore, we develop optimized algorithms for FlexDPDP operations and analyze the efficiency gains in terms of time, size and energy.

In the second part of this thesis, we propose and evaluate **InterLocal**, a novel **inte**grity and **r**eplication guaranteed **local**ity-based peer-to-peer storage system. We employ a skip graph as the underlying overlay structure, and use landmark multidimensional scaling for peer locality calculation, on the top of FlexDPDP at each node to provide data integrity. We implement both a regular skip graph based storage system and InterLocal, and evaluate their performance on the PlanetLab under various scenarios. We obtain 3x speed up in terms of file access by providing InterLocal, and a gradual performance decrease in case of replica failures, having a worst-case performance that is equal to that of a regular skip graph based storage system.

ÖZETÇE

Bilgisayar sistemlerinde veri depolama eğilimi veriyi lokal olarak tutmaktan dış kaynak kullanımına kaymıştır. Kullanıcıların veriyi bulut veya görevdeş ağlarda tutma eğilimi yanında, önemli bir gereksinim de bilginin güvenliğidir. Başlıca güvenlik gereksinimlerinden birisi dosyaların tamamını kullanıcı tarafına almadan tutarlılığının sağlanması ve güncellemelerin yapılabilmesidir. Görevdeş depolama sistemlerinde sağlanması önemli diğer bir özellik tepki süresi ve hızlı erişim açısından dosyaların veri sahibine yakın düğümlerde tutulmasıdır. Tez çalışmasının birinci kısmında, dinamik ispatlanabilir veri saklama adlı model önerilip, başarım analizi yapılmıştır. Bu modelde, atlamalı liste yapılı optimize edilmiş FlexList adlı bir veri yapısı sunup, bu yapıyı temel alan FlexDPDP adlı bütün dinamik ispatlanabilir veri saklama sistemi önerilmiştir. Ayrıca, FlexDPDP işlemleri için optimize algoritmalar önerilip, bunların zaman, enerji ve depolama boyutu bakımından kazanımları analiz edilmiştir. İkinci kısımda ise, InterLocal adlı yeni bir tutarlılık ve replikasyon garantili yerel görevdeş depolama sistemi onerilmektedir. InterLocal, her düğümde bilgi tutarlılığını sağlayabilmek amacıyla atlamalı grafik veri yapısı tabanlı FlexDPDP kullanıp, dönüm noktalı çok boyutlu ölçeklenebilir algoritmalar ile düğüm yer hesaplaması yapmaktadır. Hem normal atlamalı grafik tabanlı depolama sistemi hem de InterLocal depolama sisteminin gerçekleştirimi yapılmış ve başarımları çeşitli ağ senaryolarında PlanetLab ortamındaki deneylerde karşılaştırılmıştır. Dosya erişim süresinde tutarlılığı sağlama koşulu ile üç kat kadar hızlanma elde edilmiş ve en kötü senaryoda bile normal atlamalı grafik depolama sisteminin erişim süresinin sağlandığı gözlenmiştir.

ACKNOWLEDGMENTS

First of all, I offer my sincerest gratitude to my advisors Assoc. Prof. Öznur Özkasap and Assist. Prof. Alptekin Küpçü for their invaluable assistance, family like support, patient guidance, great motivation, knowledge and insights throughout my research. Without them this thesis would not have been completed or written. They have been more than my advisors to me.

In particular, I am grateful to Assoc. Prof. Yücel Yemez for enlightening and very useful discussions. I would like to thank the thesis committee members Prof. Özgür Barış Akan and Assoc. Prof. Mine Çağlar for providing their valuable time, effort and remarks.

Special thanks to my fellow and colleague Ertem Esiner. We have worked together on the first 8 months of the project and built our first system prototype. I feel very lucky to meet him and grateful for his brilliant ideas, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last two years. We worked together till Chapter 4.4 inclusive, prepared and submitted the first 5 chapters as a journal paper. The rest of this work, starting from Chapter 5, is going to be submitted to a conference.

I thank my fellow friends in Networks and Distributed Systems Lab: Seyhan Uçar, Nabeel Akhtar, Hüseyin Güler and many others. Also I thank my labmates in Crypto Lab: Mohammad Etemad, Handan Kılınç, Ozan Okumuşoğlu and Samuel Braunfeld.

I offer my regards to all my friends from the Fatih University, who have kept in touch and supported me throughout my studies.

The financial support of the Scientific and Technological Research Council of Turkey (TÜBİTAK) and Türk Telekom are also sincerely acknowledged.

I wish to express my most sincerely gratitude to my family. My mother, Kadyrbu Djetigenova is the kindest person in the world who supported me with all her heart, motivated as if I am the best person and guided me with her experience throughout my life. My father, Jalalkan Kachkeev, has taught to be passionate about my dreams and made me stronger in every aspect of the life. My sister, Ayjan Tuleeva is the person whom I idolize and thank for all support and guidance she has given in all these years.

TABLE OF CONTENTS

List of Tables **xii**

List of Figures **xiii**

Chapter 1: Introduction **1**

1.1 Contributions 5

1.1.1 FlexList and FlexDPDP contributions 5

1.1.2 InterLocal contributions 6

1.2 Overview 7

Chapter 2: Related Work **8**

2.1 Skip Lists and Related Data Structures 8

2.2 State of the Art in Cloud Storage 9

2.3 Peer-to-Peer Storage Related Work 12

Chapter 3: Flexlist: Flexible Length-Based Authenticated Skip List **16**

3.1 Basic Definitions 16

3.2 FlexList Structure 19

3.2.1 Preliminaries 20

3.2.2 FlexList Methods 24

3.2.3 Novel FlexList Build Function 28

3.3 FlexList Evaluation 30

3.3.1 Analysis of Core FlexList Algorithms 31

Chapter 4: FlexDPDP: Flexible Dynamic Provable Data Possession 34

4.1 Preliminaries 35

4.2 Managing Multiple Challenges at Once 39

4.2.1 Proof Generation 40

4.2.2 Proof Verification 42

4.3 Verifiable Variable-size Updates 46

4.3.1 Update Execution 46

4.3.2 Update Verification 47

4.4 Performance Analysis 51

4.5 Energy-Efficiency Analysis 56

Chapter 5: InterLocal: Integrity and Replication Guaranteed Locality-based Peer-to-Peer Storage System 60

5.1 Skip Graph 60

5.2 Landmark Multidimensional Scaling 63

5.3 InterLocal construction 64

5.3.1 Locality-based skip graph 64

5.3.2 Search in a locality-based skip graph 65

5.3.3 File operations 68

5.3.4 Locality-based Replication 68

Chapter 6: Performance Analysis on the PlanetLab 72

6.1 Evaluation settings 72

6.2 Evaluation of the Skip Graph Operations 73

6.3 Evaluation of the Provable Integrity Operations 75

6.4 Evaluation of the Replication Operations 77

Chapter 7: Conclusions **79**

Bibliography **81**

Vita **87**

LIST OF TABLES

2.1 Data structure and idea comparison. 8

2.2 Comparison table of various peer-to-peer storage systems 15

3.1 Symbol descriptions of skip list algorithms. 21

4.1 Symbols used in our algorithms. 36

LIST OF FIGURES

3.1 Regular skip list example 17

3.2 Skip list of Figure 3.1 without unnecessary links and nodes. 17

3.3 Skip list alterations depending on an update request. 18

3.4 A FlexList example with 2 sub skip lists indicated. 21

3.5 Insert at index 450, level 4 (FlexList). 26

3.6 Remove block at index 450(FlexList). 27

3.7 buildFlexList example. 30

3.8 Optimizations performed on the links and nodes 31

3.9 Time ratio on buildFlexList algorithm against insertions. 33

4.1 Client Server interactions in FlexDPDP. 35

4.2 Proof path for challenged index 450 in a FlexList. 38

4.3 Multiple blocks are challenged in a FlexList. 41

4.4 Proof vector for Figure 4.3 example. 42

4.5 Verifiable insert example. 49

4.6 Verifiable remove example. 50

4.7 Multiple block challenge for different file sizes 52

4.8 Performance gain graph 53

4.9 Time ratio on *genMultiProof* algorithm. 54

4.10 Performance evaluation of FlexList methods and their verifiable versions. 56

4.11 Time and energy ratios on buildFlexList algorithm against insertions. 57

4.12 Time and energy ratios on *genMultiProof* algorithm. 58

5.1 An example of a skip graph. 60

5.2 An example of search algorithm using numerical ID. 62

5.3 A node joining a distributed system using LMDS. 65

5.4 A map with the location information of peers. 66

5.5 A locality-based skip graph. 67

6.1 Search in a skip graph based system and InterLocal. 73

6.2 Upload time graph 74

6.3 Proof receipt time graph 75

6.4 Update operation experiment with a fixed update size of 125 Kb. . . 76

6.5 Update operation test with a fixed file size of 200 Mb. 77

6.6 Replication performance when nodes are leaving a system. 78

Chapter 1

INTRODUCTION

Peer-to-peer storage systems have been deeply investigated for the past decade [Stoica et al., 2001, Rowstron and Druschel, 2001a, Zhao et al., 2001, Ratnasamy et al., 2001, Aspnes, 2003, Harvey et al., 2003, Saroiu et al., 2002, Martins et al., 2006]. Many storage systems provide some security measures against malicious nodes in the network, such as a secure search, provable content retrieval or user anonymity, where provable data retrieval deals with the verification of the claims that the data items were put in the DHT by the data owner and were not modified by malicious nodes. But all of the systems lack, proofs on the data integrity. In this work, we aim to provide a locality-based peer-to-peer storage system with integrity and replication guarantees.

However, efficient data integrity checking for a file can be provided only by means of a proof of possession, a set of values related with the file. The cloud storage systems have substantial research works [Ateniese et al., 2007, Ateniese et al., 2008, Cash et al., 2013, Dodis et al., 2009, Erway et al., 2009, Juels and Kaliski., 2007, Shacham and Waters, 2008, Stanton et al., 2010] on data integrity, therefore client-server based scheme for checking data integrity should be developed and then applied to our peer-to-peer storage system. A client in cloud storage system outsources her data to the third party data storage provider (server), which is supposed to keep data intact and make it available to her. The problem is that the server may be malicious, and even if the server is trustworthy, hardware/software failures may cause data cor-

ruption. The client should be able to efficiently and securely check the integrity of her data without downloading the entire data from the server [Ateniese et al., 2007].

One such model proposed by Ateniese et al. is **Provable Data Possession** (PDP) [Ateniese et al., 2007] for provable data integrity. In this mode, the client can challenge the server on random blocks and verify the data integrity through a proof sent by the server. PDP and related static schemes [Ateniese et al., 2007, Ateniese et al., 2009, Dodis et al., 2009, Juels and Kaliski., 2007, Shacham and Waters, 2008] show poor performance for blockwise update operations (insertion, removal, modification). While the static scenario can be applicable to some systems (e.g., archival storage at the libraries), for many applications it is important to take into consideration the dynamic scenario, where the client keeps interacting with the outsourced data in a read/write manner, while maintaining the data possession guarantees. Ateniese et al. [Ateniese et al., 2008] proposed Scalable PDP, which overcomes this problem with some limitations (only a pre-determined number of operations are possible within a limited set of operations). Erway et al. [Erway et al., 2009] proposed a solution called *Dynamic Provable Data Possession* (DPDP), which extends the PDP model and provides a dynamic storage scheme. Implementation of the DPDP scheme requires an underlying authenticated data structure based on a skip list [Pugh, 1990b].

Authenticated **skip lists** were presented by Goodrich and Tamassia [Goodrich and Tamassia, 2001], where skip lists and commutative hashing are employed in a data structure for authenticated dictionaries. A skip list is a key-value store whose leaves are sorted by keys. Each node stores a hash value calculated with the use of its own fields and the hash values of its neighboring nodes. The hash value of the root is the authentication information (meta data) that the client stores in order to verify responses from the server. To insert a new block into an authenticated skip list, one must decide on a key value for insertion since the skip list is sorted according

to the key values. This is very useful if one, for example, inserts files into directories, since each file will have a unique name within the directory, and searching by this key is enough. However, when one considers blocks of a file to be inserted into a skip list, the blocks do not have unique names; they have indices. Unfortunately, in a dynamic scenario, an insertion/deletion would necessitate incrementing/decrementing the keys of all the blocks till the end of the file, resulting in degraded performance. DPDP [Erway et al., 2009] employs **Rank-based Authenticated Skip List** (RBASL) to overcome this limitation. Instead of providing *key* values in the process of insertion, the *index* value where the new block should be inserted is given. These indices are imaginary and no node stores any information about the indices. Thus, an insertion/deletion does not propagate to other blocks.

Theoretically, an RBASL provides dynamic updates with $\mathrm{O}(\log n)$ complexity, assuming the updates are multiples of the fixed block size. Unfortunately, a variable size update leads to the propagation of changes to other blocks, making RBASL inefficient in practice. Therefore, one variable size update may affect $O(n)$ other blocks. We discuss the problem in detail in Chapter 3. We propose **FlexList** to overcome the problem in DPDP. With our FlexList, we use the same idea but instead of the indices of blocks, indices of bytes of data are used, enabling searching, inserting, removing, modifying, or challenging a specific block containing the byte at a specific index of data. Since in practice a data alteration occurs starting from an index of the file, not necessarily an index of a block of the file, our DPDP with FlexList (**FlexDPDP**) performs much faster than the original DPDP with RBASL. Even though Erway et al. [Erway et al., 2009] presents the idea where the client makes updates on a range of bytes instead of blocks, but a naive implementation of the idea leads to a security gap in the storage system [Esiner et al., 2013]. Our optimizations result in an efficient dynamic cloud storage system, and its security directly follows from the DPDP security proof and the security of authenticated skip lists.

Another aspect of our peer-to-peer storage system is **locality**-based distributed data structure and replication mechanism. The efficiency of the storage system is measured by a query processing time and response. The most familiar query is search, which has aim to find particular data item in the network and send it to the owner of the query. Most of the systems use distributed hash tables (DHTs), such as CAN [Ratnasamy et al., 2001], Chord [Stoica et al., 2001], Pastry [Rowstron and Druschel, 2001a], Tapestry [Zhao et al., 2001], skip graph [Aspnes, 2003] and etc. All of these DHTs provide at their best O(log N) search query processing time, where N is the number of nodes in the system. Therefore, the minimal time with security enhancements can be at least with complexity of search query processing time O(log N).

Efficient **replication** system i.e. locality-based replica placement is one of the components of an efficient storage system. The client wishes to store her data at the physically closest nodes in the system, since it will save power and time on network communication. In early works, replication was performed on the nodes requesting the file only like in the systems like Gnutella [Gnutella,] and Napster [napster,]. Freenet [Clarke et al., 2001] employs a trivial replication method of creating replicas on the search or insert path in the system. Another peer-to-peer storage system called Oceanstore [Kubiatowicz et al., 2000] monitors query loads for particular files, when needed creates new replicas to overwhelmed zones and adjusts them afterwards. Past [Rowstron and Druschel, 2001b], which is based on Pastry [Rowstron and Druschel, 2001a], has locality-based replication method. It makes use of latency information in the selection of replica holders. Neighbors in terms of latency are chosen to hold replicas.

Currently proposed schemes provide one at a time, either a efficient replication mechanism or a data integrity security measure. Peer-to-peer storage systems, such as Freenet [Clarke et al., 2001], CFS [Dabek et al., 2001], Past

[Druschel and Rowstron, 2001] and Wuala [Mager et al.,], provide static file storage capabilities, therefore no block-wise update operations are possible. A storage system based on paper by Tamassia and Triandopoulos et al. exploits a new data structure distributed Merkle tree [Tamassia and Tri, 2007], it gives the user opportunity to verify the correctness of data and path verification, however the system has single point of failure (root of the distributed Merkle tree). This problem is solved in the paper by Goodrich et al. [Goodrich et al., 2009], where a scheme with employment of skip graphs and directed acyclic graphs is presented. The advantage of skip graph, which has multiple points of entry for the queries, has been exploited.

A skip graph using LMDS to represent its locality information through membership vectors is our novel locality-based peer-to-peer storage system (InterLocal). A client in such a system wishes to spend less power and time on network communication. Therefore, it is substantial to store the client's data at physically close by neighbors. Our goals are to provide a peer-to-peer storage system with the security features of data integrity and ability for the file manipulations using the nearby replicas. We develop and adopt a FlexDPDP scheme [Esiner et al., 2013] for data integrity in our project. A FlexDPDP scheme provides dynamic solution for the cloud storage systems with variable block size updates.

1.1 Contributions

1.1.1 FlexList and FlexDPDP contributions

- Our implementation uses the *optimal* number of links and nodes; we created optimized algorithms for basic operations (i.e., insertion, deletion). These optimizations are applicable to all skip list types (skip list, authenticated skip list, rank-based authenticated skip list, and FlexList).
- Our FlexList translates a variable-sized update to $O(u)$ insertions, removals, or

modifications, where u is the size of the update divided by the block size, while an RBASL requires $O(n)$ block updates.

- We provide multi-prove and multi-verify capabilities in cases where the client challenges the server for multiple blocks using authenticated skip lists, rank-based authenticated skip lists and FlexLists. Our algorithms provide an *optimal* proof, without any repeated items. The experimental results show efficiency gains of 35%, 35%, 40% in terms of proof time, energy, and size, respectively.
- We provide a novel algorithm to build a FlexList from scratch in $O(n)$ time instead of $O(n \log n)$ (time for n insertions). Our algorithm assumes the original data is already sorted, which is the case when a FlexList is constructed on top of a file in secure cloud storage.

1.1.2 InterLocal contributions

- We propose InterLocal: a novel integrity and replication guaranteed peer-to-peer storage system. It provides storage for client's files at physically close peers and efficient file access. Furthermore, we present an algorithm for search by locality information, which is useful for replica search in InterLocal.
- We deploy InterLocal and regular skip graph based system on the network testbed PlanetLab. We test time efficiency of both systems for skip graph, provable integrity and replication operations. Replication system of InterLocal provides up to 3x faster operations on the file than regular skip graph based solutions. Moreover, a worst-case performance of InterLocal in case of gradual replica failures is equal to that of a regular skip graph based storage system.

1.2 Overview

This thesis is organized as follows. In Chapter 2, we discuss related work on cloud based and peer-to-peer storage systems and data structures related to their construction. One of the most important parts of InterLocal is provable data integrity checking scheme. Therefore, we first present an efficient authenticated data structure to be used for that scheme in Chapter 3. Then, in Chapter 4 we propose our complete dynamic provable data possession scheme called FlexDPDP, its optimized operations and evaluate their performance. In Chapter 5, we propose a novel locality-based peer-to-peer storage system with integrity and replication guarantees called InterLocal, where FlexDPDP guarantees data integrity. In Chapter 6, we evaluate the performance of InterLocal and its operations. Chapter 7 concludes and states future directions.

Chapter 2

RELATED WORK

In this chapter, we discuss the literature review. First, skip list and related data structures for cloud storage systems are discussed. Then, the state of art for cloud storage systems is presented with comparison between them. At last, related work about peer-to-peer storage systems is discussed.

2.1 Skip Lists and Related Data Structures

	Storage (client)	Proof Complexity (time and size)	Dynamic (insert, remove, modify)
Hash Map (whole file)	$O(1)$	$O(n)$	-
Hash Map (block by block)	$O(n)$	$O(1)$	-
PDP [Ateniese et al., 2007]	$O(1)$	$O(1)$	-
Merkle Tree [Wang et al., 2009]	$O(1)$	$O(\log n)$	-
Balanced Tree (2-3 Tree) [Zheng and Xu, 2011]	$O(1)$	$O(\log n)$	+ (balancing issues)
RBASL [Erway et al., 2009]	$O(1)$	$O(\log n)$	+ (fixed block size)
FlexList	$O(1)$	$O(\log n)$	+

Table 2.1: Complexity and capability table of various data structures and ideas for provable cloud storage. n: number of blocks

Table 2.1 provides an overview of different data structures proposed for the secure cloud storage setting. Among the structures that enable dynamic operations, the advantage of skip list is that it keeps itself balanced probabilistically, without the need for complex operations [Pugh, 1990b]. It offers search, modify, insert, and remove operations with *logarithmic* complexity with high probability

[Pugh, 1990a]. Skip lists have been extensively studied [Anagnostopoulos et al., 2001, Battista and Palazzi, 2007, Crosby and Wallach, 2011, Erway et al., 2009, Goodrich et al., 2001, Maniatis and Baker, 2003, Polivy and Tamassia, 2002]. They are used as authenticated data structures in two-party protocols [Papamanthou and Tamassia, 2007], in outsourced network storage [Goodrich et al., 2001], with authenticated relational tables for database management systems [Battista and Palazzi, 2007], in timestamping systems [Blibech and Gabillon, 2005, Blibech and Gabillon, 2006], in outsourced data storages [Erway et al., 2009, Goodrich et al., 2008], and for authenticating queries for distributed data of web services [Polivy and Tamassia, 2002].

In a skip list, not every edge or node is used during a search or update operation; therefore those unnecessary edges and nodes can be omitted. Similar optimizations for authenticated skip lists were tested in [Goodrich et al., 2007]. Furthermore, as observed in DPDP [Erway et al., 2009] for an RBASL, some corner nodes can be eliminated to decrease the overall number of nodes. Our FlexList contains all these optimizations, and many more, analyzed both formally and experimentally.

A binary tree-like data structure called rope is similar to our FlexList [Boehm et al., 1995]. It was originally developed as alternative to the strings, bytes can be used instead of the strings as in our scheme. Since a rope is tree-like structure, it requires rebalancing operations. Moreover, a rope needs further structure optimizations to eliminate unnecessary nodes.

2.2 State of the Art in Cloud Storage

PDP was one of the first proposals for provable cloud storage [Ateniese et al., 2007]. PDP does not employ a data structure for the authentication of blocks, and is applicable to only static storage. A later variant called Scalable PDP [Ateniese et al., 2008]

allows a limited number of updates. Wang et al. [Wang et al., 2009] proposed the usage of Merkle tree [Merkle, 1987] which works perfectly for the static scenario, but has balancing problems in a dynamic setting. For the dynamic case we would need an authenticated balanced tree such as the data structure proposed by Zheng and Xu [Zheng and Xu, 2011], called range-based 2-3 tree. Yet, there is no algorithm that has been presented for rebalancing either a Merkle tree or a range-based 2-3 tree while efficient updating and maintaining authentication information. Nevertheless, such algorithms have been studied in detail for the authenticated skip list [Papamanthou and Tamassia, 2007]. Table 2.1 summarizes this comparison.

For dynamic provable data possession (DPDP) in a cloud storage setting, Erway et al. [Erway et al., 2009] were the first to introduce the new data structure *rank-based authenticated skip list (RBASL)* which is a special type of the authenticated skip list [Goodrich et al., 2001]. In the DPDP model, there is a client who wants to outsource her file and a server that takes the responsibility for the storage of the file. The client pre-processes the file and maintains meta data to verify the proofs from the server. Then she sends the file to the server. When the client needs to check whether her data is intact or not, she challenges some random blocks. Upon receipt of the request, the server generates the proof for the challenges and sends it back. The client then verifies the data integrity of the file using this proof. Many other static and dynamic schemes have been proposed [Juels and Kaliski., 2007, Shacham and Waters, 2008, Dodis et al., 2009, Cash et al., 2013] including multi-server optimizations on them [Bowers et al., 2009, Curtmola et al., 2008, Etemad and Küpçü, 2013].

An RBASL, unlike an authenticated skip list, allows a search with indices of the blocks. This gives the opportunity to efficiently check the data integrity using block indices as proof and update query parameters in DPDP. To employ indices of the blocks as search keys, Erway et al. proposed using authenticated ranks. Each node in the RBASL has a *rank*, indicating the number of the leaf-level nodes that are reachable

from that particular node. Leaf-level nodes having no $after$ links have a rank of 1, meaning they can be used to reach themselves only. Ranks in an RBASL handle the problem with block numbers in PDP [Ateniese et al., 2007], and thus result in a dynamic system.

Nevertheless, in a realistic scenario, the client may wish to change a part of a block, not the whole block. This can be problematic to handle in an RBASL. To partially modify a particular block in an RBASL, we not only modify a specified block but also may have to change all following blocks. This means the number of modifications is $O(n)$ in the worst case scenario for DPDP as well.

Another dynamic provable data possession scheme was presented by Zhang et al. [Zhang and Blanton, 2013]. They employ a new data structure called a balanced update tree, whose size grows with the number of the updates performed on the data blocks. Due to this property, they require extra rebalancing operations. The scheme uses message authentication codes (MAC) to protect the data integrity. Unfortunately, since the MAC values contain indices of data blocks, they need to be recalculated with insertions or deletions. The data integrity checking can also be costly, since the server needs to send all the challenged blocks with their MAC values, because the MAC scheme is not homomorphic (see [Ateniese et al., 2009]). In our scheme we send only tags and a block sum, which is approximately of a single block size. At the client side, there is an overhead for keeping the update tree.

Our proposed data structure FlexList, based on an authenticated skip list, performs dynamic operations (modify, insert, remove) for cloud data storage, having efficient variable block size updates.

2.3 Peer-to-Peer Storage Related Work

A client-oriented efficient peer-to-peer storage system should be implemented exploiting efficient distributed data structure and locality-based replica placement. Researchers in this field has advanced and presented a lot of works on data replication [Martins et al., 2006]. Most of them exploit benefits of distributed hash tables (DHTs) [Stoica et al., 2001, Rowstron and Druschel, 2001a, Zhao et al., 2001, Ratnasamy et al., 2001, Aspnes, 2003, Harvey et al., 2003]. A skip graph is a distributed data structure, based on skip lists [Pugh, 1990b], that provides the full functionality of a balanced tree in a distributed system where resources are stored in separate nodes that may fail at any time. Skip graph is designed for use in searching peer-to-peer systems, and by providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash table functionality. Furthermore, since a query can start at any node in the system, it has no single point of failure. Another advantage is that the links between neighbors are based on prefix similarities of membership vectors.

Another aspect of peer-to-peer storage system is data management. Popular peer-to-peer storage systems like Freenet [Clarke et al., 2001], CFS [Dabek et al., 2001], Past [Druschel and Rowstron, 2001] and Wuala [Mager et al.,] have different structures and lookup algorithms but all of them static, since no block-wise updates are possible. Oceanstore [Kubiatowicz et al., 2000] that employs super nodes (cloud servers) as a backbone for the system availability. It has capability to read/write but it will require to update the whole file, since it uses erasure coding on the files. Past [Rowstron and Druschel, 2001b] has a similar replication method to ours. Its idea is to place file replicas near to the client according the node identification number. In the search operation most of client's requests received for file operations are processed by one of the replicas [Rowstron and Druschel, 2001b]. However, in our system we

also place replicas using the hash function on the file name (file owner), similarly as in other DHTs. Therefore, the diversity of the replica placement prevents possible availability problems with the files.

Tamassia and Triandopoulos et al. [Tamassia and Tri, 2007] presented a new model for data authentication in peer-to-peer network and their construction of a distributed Merkle tree (DMT). DMT is a authentication tree distributed over peer-to-peer network, which allows users to verify the integrity of the data objects received from the network and give the data owner ability to verify the integrity of updates executed by the network. There are two type of nodes: network node and DMT node. Each DMT node is represented by one of the network nodes. A query is processed by authenticated distributed hash table (ADHT), which is secure extension of DHT. When the query is delivered to the node holding data object, then the node is the leaf-level node in DMT. Starting from it, the proof for the data object is generated. The proof traverses backward from the leaf-level node to the top of the tree (root) and on its way collects auxiliary information (level, position in the tree) and hash values. At the end, the user can verify the proof by comparing hash value of the root and hash value computed using proof, as well as check validity of the signature of the hash value of root. Unfortunately, this model lacks load balancing and there is single point of failure, which the root of the DMT.

The extension of the previous paper is presented by [Goodrich et al., 2009]. They propose two hashing-based scheme for reliable resource location and content retrieval queries. In peer-to-peer setting, there can be nodes that can act maliciously. These type of nodes can redirect queries to wrong nodes or resources, change requests, or even redirect to not only to outdated file but to virus infected. Their idea is to limit the ability of adversarial nodes to carry out attacks. Their first authenticated scheme is called skip-DHT, is based on skip graphs [Aspnes, 2003]. Its construction authenticates all possible search paths that can be used for locating the resource. The

second authentication scheme is a middleware component that can be exploited by any DHT to verify put/get operations on a data set. And each of the search path is linked to the verifiable hash value signed by the data owner. Scheme basically addresses data integrity issues at the distributed data structure level, but we perform that checks at the node level, therefore a single node affected by these operations.

For a more detailed comparison of the storage systems see Table 2.2. There a number of criteria that are used to compare those systems. One of them is decentralization, which is "Full" if a system consists of regular peers with same duties and "Hybrid" if a system employs some super nodes that perform special duties.

For data integrity we use FlexDPDP scheme [Esiner et al., 2013]. It is a Dynamic Provable Data Possession (DPDP) scheme [Erway et al., 2009] using FlexList (underlying data structure) proposed by [Esiner et al., 2013]. FlexList is a skip list like authenticated data structure with support for variable block size updates. It is an enhanced version of rank-based authenticated skip lists [Erway et al., 2009]. which . Earlier works include a variety of static [Ateniese et al., 2007, Juels and Kaliski., 2007, Shacham and Waters, 2008, Wang et al., 2009, Zheng and Xu, 2011, Dodis et al., 2009] and dynamic [Erway et al., 2009, Esiner et al., 2013, Zhang and Blanton, 2013, Ateniese et al., 2008] solutions. However, even dynamic schemes have their limitations. In a scalable PDP [Ateniese et al., 2008], the client can have a predetermined number of updates. A scheme proposed by Zhang et al. [Zhang and Blanton, 2013] has rebalancing issues with update trees and has to perform extra calculations with each update. A FlexDPDP scheme [Esiner et al., 2013], which allows variable block size updates, is a modified version of DPDP [Erway et al., 2009].

System name	Data locating	Decentralization	Replication	Data Update	Proof of Possession	Locality
Past [Druschel and Rowstron, 2001]	Pastry	F	Replicate to k numerically closest nodes	R	No	No
Freenet [Clarke et al., 2001]	Probabilistic routing	F	Create replicas on search path	R	No	No
CFS [Dabek et al., 2001]	Chord	F	Replicate blocks of a file and cache them if needed	R	No	No
Oceanstore [Kubiatowicz et al., 2000]	Tapestry and Probabilistic routing	H	Replicate at hot spots and caching	R	No	No
Wuala [Mager et al.,]	Chord / Tapestry	H	Random replica placement	R	No	No
DMT [Tamassia and Tri, 2007]	Distributed Merkle tree	F	No replication	RW	Authentication of search path at distributed data structure	No
skip-DHT [Goodrich et al., 2009]	Skip graph	F	No replication	RW	Authentication of search path at distributed data structure	No
InterLocal	Skip graph	F	Random and locality-based replication	RW	Authenticated search path of data structure at the peer	Yes

Table 2.2: Comparison table of various peer-to-peer storage systems. R: Read-only RW: Read/Write F: Full H: Hybrid

Chapter 3

FLEXLIST: FLEXIBLE LENGTH-BASED AUTHENTICATED SKIP LIST

In this chapter, we present the underlying authenticated data structure for dynamic cloud storage system called FlexList. It starts with the definitions and comparison of FlexList with RBASL. Afterwards, basic helper functions and main methods of FlexList are presented and detailed with examples. Then, a novel function to build a FlexList from a scratch is presented. Finally, experimental results for both FlexList main functions and the novel build function are evaluated.

3.1 Basic Definitions

Skip List is a probabilistic data structure presented as an alternative to balanced trees [Pugh, 1990b]. It is easy to implement without complex balancing and restructuring operations such as those in AVL or Red-Black trees [Anagnostopoulos et al., 2001, Foster, 1973]. A skip list keeps its nodes ordered by their *key* values. We call a leaf-level node and all nodes directly above it at the same index a *tower*.

Figure 3.1 demonstrates a search on a skip list. The search path for the node with key 24 is highlighted. In a basic skip list, the nodes include *key*, *level*, and data (only at leaf level nodes) information, and *below* and $after$ links (e.g., $v_2.below = v_3$ and $v_2.after = v_4$). To perform the search for 24, we start from the root (v_1) and follow the link to v_2, since v_1's $after$ link leads it to a node which has a greater key value

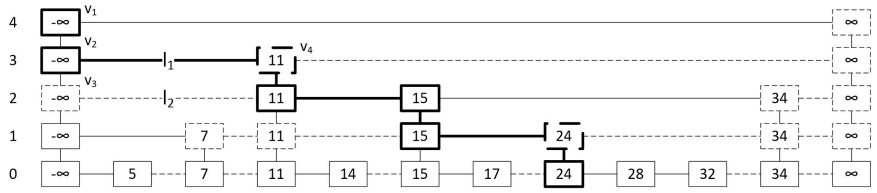

Figure 3.1: Regular skip list with search path of node with key 24 highlighted. Numbers on the left represent levels. Numbers inside nodes are key values. Dashed lines indicate unnecessary links and nodes.

than the key we are searching for ($\infty > 24$). Then, from v_2 we follow link l_1 to v_4, since the key value of v_4 is smaller than (or equal to) the searched key. In general, if the key of the node where $after$ link leads is smaller or equal to the key of the searched node, we follow that link, otherwise we follow the *below* link. Using the same decision mechanism, we follow the highlighted links until the searched node is found at the leaf level (if it does not exist, then the node with key immediately before the searched node is returned).

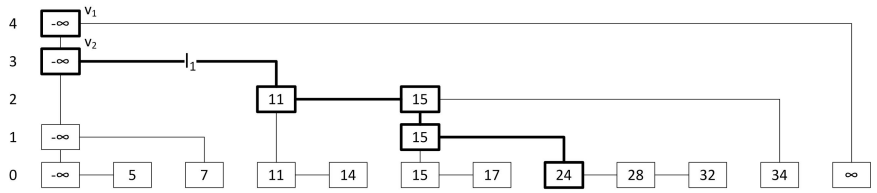

Figure 3.2: Skip list of Figure 3.1 without unnecessary links and nodes.

We observe that some of the links are never used in the skip list, such as l_2, since any search operation with key greater or equal to 11 will definitely follow link l_1, and a search for a smaller key would never advance through l_2. Thus, we say links that are not present on any search path, such as l_2, are unnecessary. When we remove unnecessary links, we observe that some nodes, which are left without $after$ links (e.g., v_3), are also unnecessary since they do not provide any new dependencies in

the skip list. Although it does not change the asymptotic complexity, it is beneficial not to include them for time and space efficiency. An optimized version of the skip list from Figure 3.1 can be seen in Figure 3.2 with the same search path highlighted. Formally:

- A **link** is **necessary** if and only if it is on any search path.
- A **node** is **necessary** if and only if it is at the leaf level or has a necessary $after$ link.

Assuming existence of a collision-resistant hash function family H, we randomly pick a hash function h from H and let $||$ denote concatenation. Throughout our study we will use: $hash(x_1, x_2, ..., x_m)$ to mean $H(x_1||x_2||...||x_m)$.

An **authenticated skip list** is constructed with the use of a collision-resistant hash function and keeps a hash value in each node. Nodes at level 0 keep links to file blocks (may link to different structures e.g., files, directories, anything to be kept intact) [Goodrich et al., 2001]. A hash value is calculated with the following inputs: *level* and *key* of the node, and the hash values of the node $after$ and the node *below*. Through the inputs to the hash function, all nodes are dependent on their *after* and *below* neighbors. Thus, the root node is dependent on every leaf node, and due to the collision resistance of the hash function, knowing the hash value of the root is sufficient for later integrity checking. Note that if there is no node

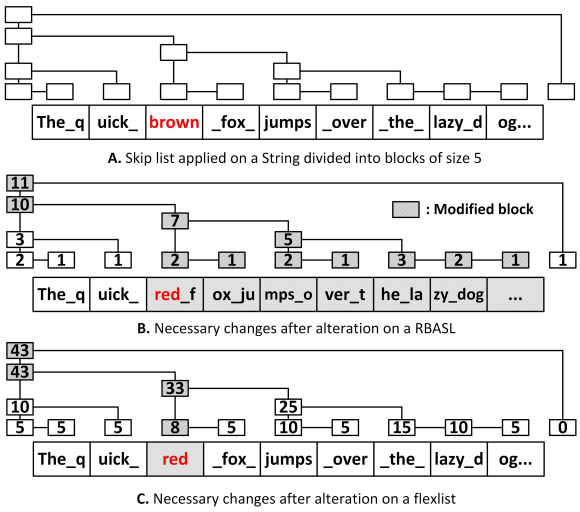

Figure 3.3: Skip list alterations depending on an update request.

below, data or a function of data (which we will call *tag* in the following sections) is used instead of the hash of the *below* neighbor. If there is no $after$ neighbor, then a dummy value (e.g., null) is used in the hash calculation.

A **rank-based authenticated skip list (RBASL)** is different from an authenticated skip list by means of how it indexes data [Erway et al., 2009]. An RBASL has *rank* information (used in hashing instead of the *key* value), meaning how many nodes are reachable from that node. An RBASL is capable of performing all operations that an authenticated skip list can in the cloud storage context.

3.2 FlexList Structure

A FlexList supports variable-sized blocks whereas an RBASL is meant to be used with fixed block size since a search (consequently insert, remove, modify) by index of data is not possible with the rank information of an RBASL. For example, Figure 3.3-A represents an outsourced file divided into blocks of fixed size.

In our example, the client wants to change "brown" in the file composed of the text "The quick brown fox jumps over the lazy dog..." with "red" and the diff algorithm returns [delete from index 11 to 15] and [insert "red" from index 11 to 13]. Apparently, a modification to the 3^{rd} block will occur. With a rank-based skip list, to continue functioning properly, a series of updates is required as shown in Figure 3.3-B which asymptotically corresponds to $O(n)$ alterations. Otherwise, the beginning and the ending indices of each block will be complicated to compute, requiring $O(n)$ time to translate a diff algorithm output to block modifications at the server side. It also leaves the client unable to verify that the index she challenged is the same as the index of the proof by the server (this issue is explained in Section 4.2 with the *verifyMultiProof* algorithm). Therefore, for instance a FlexList having 500000 leaf-level nodes needs an expected 250000 update operations for a single variable-sized update.

Besides the modify operations and related hash calculations, this also corresponds to 250000 new tag calculations either on the server side, where the private key (order of the RSA group) is unknown (thus computation is very slow) or at the client side, where the new tags should go through the network. Furthermore, a verification process for the new blocks is also required (that means a huge proof, including half of the data structure used, sent by the server and the verified by the client, where she needs to compute an expected 375000 hash values). With our FlexList, only one modification suffices as indicated in Figure 3.3-C.

Due to the lack of providing variable block sized operations with an RBASL, we present FlexList which overcomes this problem and serves our purposes in the cloud data storage setting. A FlexList stores, at each node, how many *bytes* can be reached from that node, instead of how many *blocks* are reachable. The *rank* of each leaf-level node is computed as the sum of the *length* of its data and the *rank* of the $after$ node (0 if *null*). The *length* information of each data block is added as a parameter to the hash calculation of that particular block. Note that when the length of data at each leaf is considered as a unit, the FlexList reduces to an RBASL (thus, ranks only count the number of reachable blocks). Therefore all our optimizations are also applicable to RBASL, which is indeed a special case of FlexList.

3.2.1 Preliminaries

Algorithm 3.2.1: nextPos Algorithm

Input: pn, cn, i, $level$, npi
Output: pn, cn, i, $\sqcup_n$

```
1  ⊔_n = new empty Stack
2  while cn can go below OR after do
3      if canGoBelow(cn, i) AND cn.below.level ≥ level AND npi then
4          cn = cn.below
5      else if canGoAfter(cn, i) AND cn.after.level ≥ level then
6          i = i - cn.below.r; cn = cn.after
7      add cn to ⊔_n
```

In this section, we introduce the helper methods required to traverse the skip list,

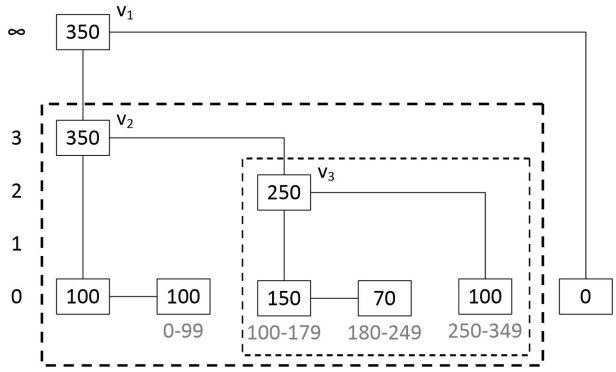

Figure 3.4: A FlexList example with 2 sub skip lists indicated.

Symbol	Description
cn	current node
pn	previous node, indicates the last node that current node moved from
mn	missing node, created when there is no node at the point where a node has to be linked
nn	new node
dn	node to be deleted
$after$	the after neighbor of a node
$below$	the below neighbor of a node
r	rank value of a node
i	index of a byte
npi	a boolean which is always true except in the inner loop of *insert* algorithm
$\sqcup_n$	stack (initially empty), filled with all visited nodes during *search*, $modify$, *insert* or *remove* algorithms

Table 3.1: Symbol descriptions of skip list algorithms.

create missing nodes, delete unnecessary nodes, delete nodes, and decide on the level to insert at, to be used in the essential algorithms (*search, modify, insert, remove*). Note that all algorithms are designed to fill a stack $\sqcup_n$ where we store nodes which may need a recalculation of hash values if authenticated, and rank values if using FlexList. All algorithms that move the current node immediately push the new current node to the stack $\sqcup_n$ as well. Further notations are shown in Table 3.1.

We first define a concept called **sub skip list** to make our FlexList algorithms easier to understand. An example is illustrated in Figure 3.4. Let the search index be

250 and the current node start at the root (v_1). The current node follows its *below* link to v_2 and enters a sub skip list (big dashed rectangle). Now, v_2 is the root of this sub skip list and the searched node is still at index 250. In order to reach the searched node, the current node moves to v_3, which is the root of another sub skip list (small dashed rectangle). Now, the searched byte is at index 150 in this sub skip list. Therefore the searched index is updated accordingly. The amount to be reduced from the search index is equal to the difference between the rank values of v_2 and v_3, which is equal to the rank of *below* of v_2. Whenever the current node follows an $after$ link, the search index should be updated. To finish the search, the current node follows the $after$ link of v_3 to reach the node containing index 150 in the sub skip list with root v_3.

nextPos (Algorithm 3.2.1): The *nextPos* method moves the current node *cn* repetitively until the desired position according to the method (*search*, *insert*, *remove*) from which it is called. There are 4 cases for *nextPos*:

- *insert* - moves current node *cn* until the closest node to the insertion point.
- *remove* or *search* - moves current node *cn* until it finds the searched node's tower.
- loop in *insert* - moves *cn* until it finds the next insertion point for a new node.
- loop in *remove* - moves current node *cn* until it encounters the next node to delete.

Algorithm 3.2.2: createMissingNode Algorithm

Input: *pn*, *cn*, *i*, *level*
Output: *pn*, *cn*, *i*, $\sqcup_n$

```
1  ⊔n = new empty Stack
2  mn = new node is created using level //Note that rank value for missing
   node is given ∞
3  if canGoBelow(cn,i) then
4      mn.below = cn.below; cn.below = mn
5  else
6      mn.below = cn.after; cn.after = mn
7      i = i - cn.below.r //Since current node is going after, i value should
       be updated
8  pn = cn; cn = mn; then cn is added to ⊔n
```

createMissingNode (Algorithm 3.2.2) is used in both the *insert* and *remove* algorithms. Since in a FlexList there are only necessary nodes, when a new node needs to be connected, this algorithm creates any missing node to make the connection.

deleteUNode (Algorithm 3.2.3) is employed in the *remove* and *insert* algorithms to delete an unnecessary node (this occurs when a node loses its *after* node) and maintain the links. It takes the previous node and current node as inputs, where the current node is unnecessary and meant to be deleted. The purpose is to preserve connections between necessary nodes after the removal of the unnecessary one. This involves deletion of the current node if it is not at the leaf level. It sets the previous node's $after$ or *below* to the current node's *below*. As the last operation of deletion, we remove the top node from the stack $\sqcup_n$, as its rank and hash values no longer need to be updated.

Algorithm 3.2.3: deleteUNode Algorithm

Input: *pn*, *cn*
Output: *pn*, *cn*, $\sqcup_n$

```
1  ⊔n = new empty Stack
2  if cn.level == 0 then
3      cn.after = NIL
4  else
5      if pn.below == cn then
6          pn.below = cn.below
7      else
8          pn.after = cn.below
9      ⊔n.pop(); cn = pn
```

deleteNode method, employed in the *remove* algorithm, takes two consecutive nodes, the previous node and the current node. By setting $after$ pointer of the previous node to current node's $after$, it detaches the current node from the FlexList.

tossCoins: Probabilistically determines the level value for a new node tower. A coin is tossed until it comes up heads. The output is the number of consecutive tails.

3.2.2 FlexList Methods

FlexList is a particular way of organizing data for secure cloud storage systems. Some basic functions must be available, such as search, modify, insert and remove. These functions are employed in the verifiable updates. All algorithms are designed to fill a stack for the possibly affected nodes. This stack is used to recalculate of rank and hash values accordingly.

search (Algorithm 3.2.4) is the algorithm used to find a particular byte. It takes the index i as the input, and outputs the node at index i with the stack $\sqcup_n$ filled with the nodes on the search path. Any value between 0 and the file size in bytes is valid to be searched. It is not possible for a valid index not to be found in a FlexList. A search path, which is the basic idea of a proof path, is visible in the stack in the basic algorithms.

Algorithm 3.2.4: search Algorithm

Input: i
Output: cn, $\sqcup_n$

1 $\sqcup_n$ = new empty Stack
2 $cn = root$
`// cn moves until cn.after is a tower node of the searched node`
3 call $nextPos$
4 $cn = cn.after$ then cn is added to $\sqcup_n$
`// cn is moved below until the node at the leaf level, which has data`
5 **while** $cn.level \neq 0$ **do**
6 $cn = cn.below$ then cn is added to $\sqcup_n$

In algorithm 3.2.4, the current node cn starts at the root. The $nextPos$ method moves cn to the position just before the top of the tower of the searched node. Then cn is taken to the searched node's tower and moved all the way down to the leaf level.

modify: By taking index i and new data, we make use of the *search* algorithm for the node, which includes the byte at index i, and update its data. It then we recalculate hash values along the search path. The input of this algorithm contains the index i and new data. The outputs are the modified node and stack $\sqcup_n$ filled with nodes on the search path.

Algorithm 3.2.5: insert Algorithm

Input: i, data
Output: nn,$\sqcup_n$

```
1   ⊔_n = new empty Stack
2   pn = root; cn = root; level = tossCoins()
3   call nextPos// cn moves until it finds a missing node or cn.after is
    where nn is to be inserted
    // Check if there is a node where new node will be linked.  if not,
    create one.
4   if !CanGoBelow(cn, i) or cn.level ≠ level then
5       call createMissingNode;
    // Create new node and insert after the current node.
6   nn = new node is created using level
7   nn.after = cn.after; cn.after = nn and nn is added to ⊔_n
    // Create insertion tower until the leaf level is reached.
8   while cn.below ≠ null do
9       if nn already has a non-empty after link then
10          a new node is created to the below of nn; nn = nn.below and nn is added
            to ⊔_n
11      call nextPos // Current node moves until we reach an after link that
        passes through the tower.  That is the insertion point for the
        new node.
        // Create next node of the insertion tower.
12      nn.after = cn.after; nn.level = cn.level
        // cn becomes unnecessary as it looses its after link, therefore
        it is deleted
13      deteletUNode(pn, cn);
    // Done inserting, put data and return this last node.
14  nn.data = data
    // For a FlexList, call calculateHash and calculateRank on the nodes
    in the ⊔_n to compute their (possibly) updated values.
```

insert (Algorithm 3.2.5) is run to add a new node to the FlexList with a random level by adding new nodes along the insertion path. The inputs are the index i and data. The algorithm generates a random *level* by tossing coins, then creates the new node with given data and attaches it to index i, along with the necessary nodes until the *level*. Note that this index should be the beginning index of an existing node, since inserting a new block inside a block makes no sense.[1] As output, the algorithm returns the stack $\sqcup_n$ filled with nodes on the search path of the new block.

Figure 3.5 demonstrates the insertion of a new node at index 450 with *level* 4. *nextPos* brings the current node to the closest node to the insertion point with level

[1]In case of an addition inside a block we can do the following: search for the block including the byte where the insertion will take place, add our data in between the first and second part of data found to obtain new data and employ *modify* algorithm (if new data is long, we can divide it into parts and send it as one modify and a series of inserts).

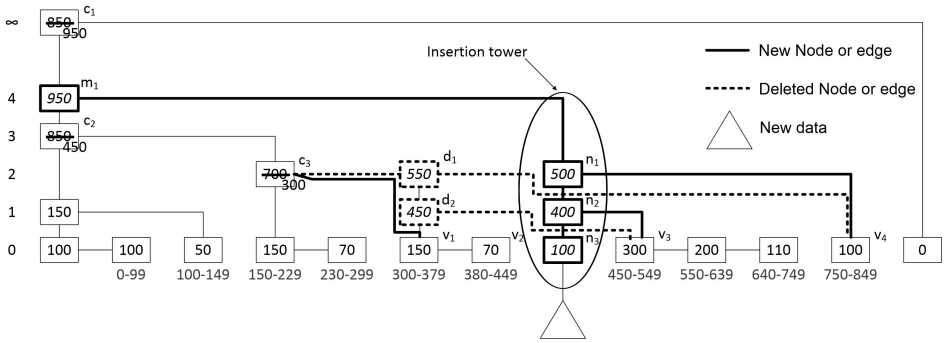

Figure 3.5: Insert at index 450, level 4 (FlexList).

greater than or equal to the insertion level (c_1 in Figure 3.5). Lines 3-4 create any missing node at the *level*, if there was no node to connect the new node to (e.g., m_1 is created to connect n_1 to). Within the while loop, during the first iteration, n_1 is inserted to level 2 since nodes at levels 3 and 4 are unnecessary in the insertion tower. Inserting n_1 makes d_1 unnecessary, since n_1 stole its after link. Likewise, the next iteration results in n_2 being inserted at level 1 and d_2 being removed. Note that removal of d_1 and d_2 results in c_3 getting connected to v_1. The last iteration inserts n_3, and places data. Since this is a FlexList, hashes and ranks of all the nodes in the stack will be recalculated ($c_1, m_1, n_1, c_2, c_3, n_2, n_3, v_1, v_2$). Those are the only nodes whose hash and rank values might have changed.

remove (Algorithm 3.2.6) is run to remove the node which starts with the byte at index i. As input, it takes the index i. The algorithm detaches the node to be removed and all other nodes above it while preserving connections between the remaining nodes. As output, the algorithm returns the stack $\sqcup_n$ filled with the nodes on the search path of the left neighbor of the node removed.

Algorithm 3.2.6: remove Algorithm

Input: i
Output: dn,$\sqcup_n$

```
1   ⊔n = new empty Stack pn = root; cn = root
2   call nextPos // Current node moves until after of the current node is
    the node at the top of deletion tower
3   dn = cn.after
    // Check if current node is necessary,if so it can steal after of the
    node to delete, otherwise delete current node
4   if cn.level = dn.level then
5       deleteNode(cn, dn); dn = dn.below; // unless at leaf level
6   else
7       deleteUNode(pn, cn);
    // Delete whole deletion tower until the leaf level is reached
8   while cn.below ≠ null do
9       call nextPos// Current node moves until it finds a missing node
        // Create the missing node unless at leaf level and steal the
        after link of the node to delete
10      call createMissingNode; deleteNode(cn, dn)
11      dn = dn.below // move dn to the next node in the deletion tower
        unless at leaf level
    // For a FlexList, call calculateHash and calculateRank on the nodes
    in the ⊔n to compute their (possibly) updated values.
```

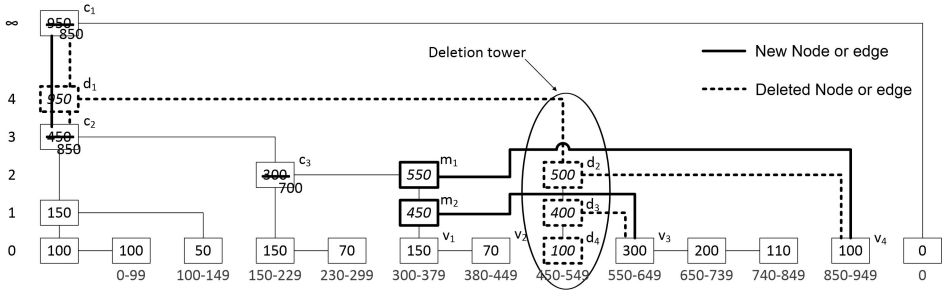

Figure 3.6: Remove block at index 450(FlexList).

Figure 3.6 demonstrates removal of the node having the byte with index 450. The algorithm starts at the root c_1, and the first *nextPos* call on line 2 returns d_1. Lines 4-7 check if d_1 is necessary. If d_1 is necessary, d_2 is deleted and we continue deleting from d_3. Otherwise, if d_1 is unnecessary, then d_1 is deleted, and we continue searching from c_1. In our example, d_1 is unnecessary, so we continue from c_1 to delete d_2. Within the while loop, the first call of *nextPos* brings the current node to c_3. The goal is to delete d_2, but this requires creating of a missing necessary node m_1. Note

that, m_1 is created at the same level as d_2. Once m_1 is created and d_2 is deleted, the while loop continues its next iteration starting from m_1 to delete d_3. This next iteration creates m_2 and deletes d_3. The last iteration moves the current node to v_2 and deletes d_4 without creating any new nodes, since we are at the leaf level. The output stack contains nodes $(c_1, c_2, c_3, m_1, m_2, v_1, v_2)$. Rank and hash values of those nodes could have changed, those values will be recalculated.

3.2.3 Novel FlexList Build Function

Algorithm 3.2.7: buildFlexList Algorithm

Input: B, L, T
Output: *root*

```
    // H will keep pointers to tower heads
1   H = new vector is created of size L_0 + 1
    // Loop will iterate for each block
2   for i = B.size − 1 to 0 do
3       pn= null
4       for j = 0 to L_i+1 do
            // Enter only if at level 0 or H_j has an element
5           if H_j ≠ null or j = 0 then
6               nn = new node is created with level j //if j is 0, B_i,T_i are
                included to the creation of nn
7               nn.below = pn; nn.after = H_j // Connect tower head at H_j as
                an after link
8               call calculateRank and calculateHash on nn
9               pn = nn; H_j = null
        // Add a tower head to H at H_{L_i}
10      H_{L_i} = pn
11  root = H_{L_0} //which is equal to pn
12  root.level = ∞; call calculateHash on root
13  return root
```

The usual way to build a skip list (or FlexList) is to perform n insertions (one for each item). When original data is already sorted, one may insert them in increasing or decreasing order. Such an approach will result in $O(n \log n)$ total time complexity. But, when data is sorted as in the secure cloud storage scenario (where blocks of a file are already sorted), a much more efficient algorithm can be developed. Observe that a skip list contains $2n$ nodes in total, in expectation [Pugh, 1990b]. This is an $O(n)$ value, and thus spending $O(n \log n)$ time for creating $O(n)$ nodes is an overkill,

since creation of nodes take a constant time only. We present our novel algorithm for building a FlexList from scratch in just $O(n)$ time. To the best of our knowledge, such an efficient build algorithm did not exist before.

buildFlexList (Algorithm 3.2.7) is an algorithm that generates a FlexList over a set of sorted data in time complexity $O(n)$. It has the small space complexity of $O(l)$ where l is number of levels in the FlexList ($l = O(\log n)$ with high probability). As the inputs, the algorithm takes blocks B on which the FlexList will be generated, corresponding (randomly generated) levels L and tags T. The algorithm assumes data is already sorted. In cloud storage, the blocks of a file are already sorted according to their block indices, and thus our optimized algorithm perfectly fits our target scenario. The algorithm attaches one link for each tower from right to left. For each leaf node generated, its tower follows in a bottom up manner. As output, the algorithm returns the root node.

Figure 3.7 demonstrates the building process of a FlexList where the insertion levels of blocks are 4, 0, 1, 3, 0, 2, 0, 1, 4, in order. Labels v_i on the nodes indicate the generation order of the nodes. Note that the blocks and the tags for the sentinel nodes are null values. The idea of the algorithm is to build towers of a given level for each block. As shown in the figure, all towers have only one link from left side to its tower head (the highest node in the tower). Therefore, we need to store the tower heads in a vector, and then make necessary connections. The algorithm starts with the creation of the vector H to hold pointers to the tower heads at line 1. At lines 6-9 for the first iteration of the inner loop, the node v_1 is created which is a leaf node, thus there is no node below. Currently, H is empty; therefore there is no node at H_0 to connect to v_1 at level 0. The hash and the rank values of v_1 are calculated. Since H is still empty, we do not create new nodes at levels 1, 2, 3, 4. At line 10, we put v_1 to H as H_4. The algorithm continues with the next block and the creation of v_2. H_0 is still empty, therefore no $after$ link for v_2 is set. The hash and the rank values of

v_2 are calculated. The next iterations of the inner loop skip the lines 6-9, because H_1 and H_2 are empty as well. At line 10, v_2 is inserted to H_2. Then, v_3 is created and its hash and rank values are calculated. There is no element at H_0 to connect to v_3. Its level is 0, therefore it is added to H as H_0. Next, we create the node v_4; it takes H_0 as its $after$. The hash and the rank values are calculated, then v_4 is added to H at index 0. The algorithm continues for all elements in the block vector. At the end of the algorithm, the root is created, connected to the top of the FlexList, then its hash and rank values are calculated.

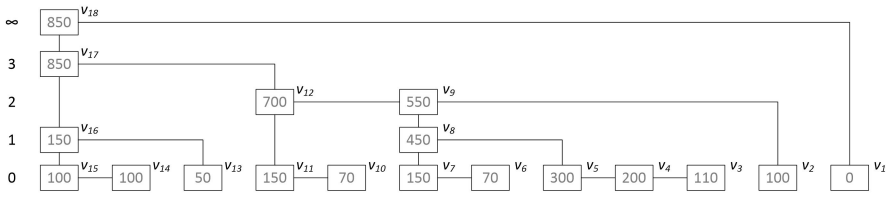

Figure 3.7: buildFlexList example.

3.3 FlexList Evaluation

We have developed a prototype implementation of an optimized FlexList (on top of our optimized skip list and authenticated skip list implementations). We used C++ and employed some methods from the *Cashlib* library [Meiklejohn et al., 2010, Brownie Points Project,]. The local experiments were conducted on a 64-bit machine with a 2.4GHz Intel 4 core CPU (only one core is active), 4GB main memory and 8MB L2 cache, running Ubuntu 12.10. As security parameters, we used 1024-bit RSA modulus, 80-bit random numbers, and SHA-1 hash function, overall resulting in an expected security of 80-bits. All our results are the average of 10 runs. The tests include I/O access time since **each block of the file is kept on the hard disk**

drive separately, unless it stated otherwise. The size of a FlexList is suitable to keep a lot of FlexLists in RAM.

3.3.1 Analysis of Core FlexList Algorithms

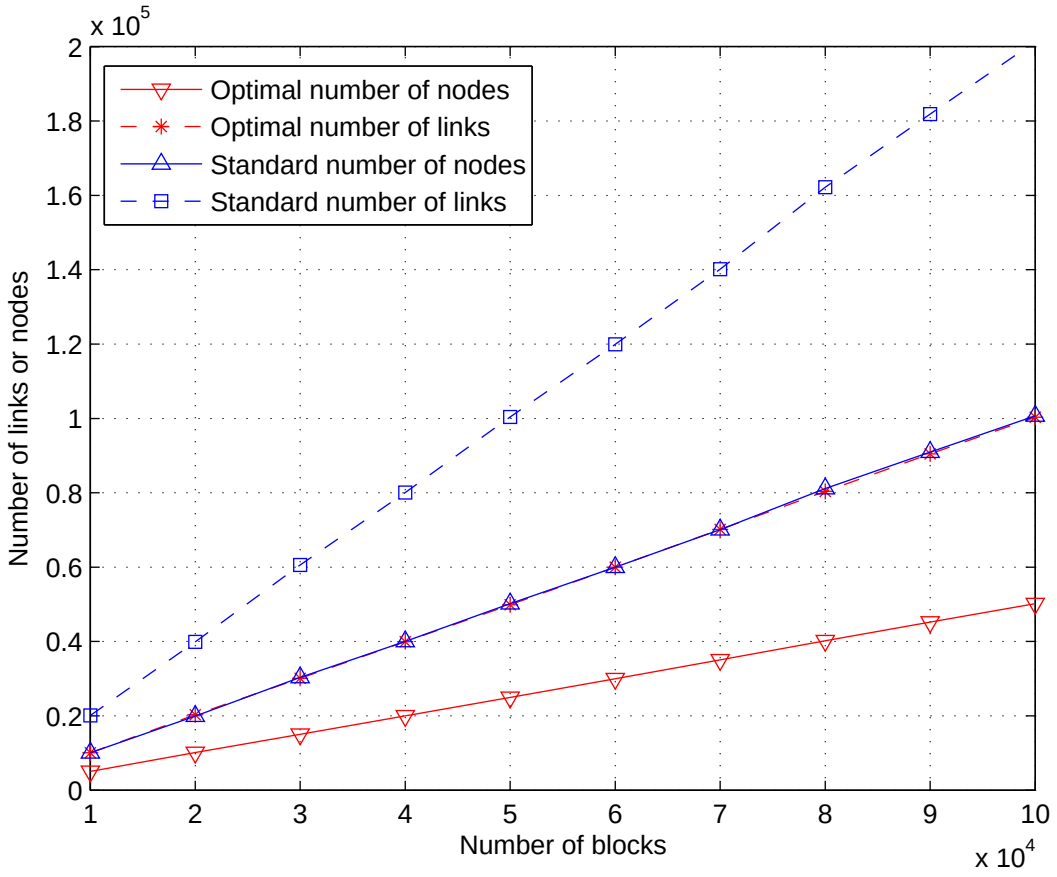

Figure 3.8: The number of nodes and links used on top of leaf level nodes, before and after optimization.

One of the core optimizations in a FlexList is done in terms of the structure. Our optimization, removing unnecessary links and nodes, ends up with 50% less nodes and links on top of the leaf nodes, which are always necessary since they keep the file blocks. Figure 3.8 shows the number of links and nodes used before and after

optimization. The expected number of nodes in a regular skip list is $2n$ [Pugh, 1990b] (where n represents the number of blocks): n leaf nodes and n non-leaf nodes. Each non-leaf node makes any left connection below its level unnecessary as described in Section 3.1. Since in a skip list, half of all nodes and links are at the leaf level in expectation, this means half of the non-leaf level links and half of the leaf level links are unnecessary, making a total on n unnecessary links. Since there are $n/2$ non-leaf unnecessary links, it means that there are $n/2$ non-leaf unnecessary nodes as well, according to unnecessary node definition (Section 3.1). Hence, there are $n - n/2 = n/2$ non-leaf necessary nodes. Since each necessary node has 2 links, in total there are $2 * n/2 = n$ necessary links above the leaf level. Therefore, in Figure 3.8, there is an overlap between the standard number of non-leaf nodes (n) and the optimal number of the non-leaf links (n). Therefore, we eliminated approximately **50**% of **all nodes and links** above the leaf level (and 25% of all).

Moreover, we presented a novel algorithm for the efficient building of a FlexList. Figure 3.9 demonstrates time ratios between the *buildFlexList* algorithm and building FlexList by means of insertion (in sorted order). The time ratio is calculated by dividing the time spent for the building FlexList using insertion method by the time needed by the *buildFlexList* algorithm. In our time ratio experiments, we do not take into account the disk access time; therefore there is no delay for I/O switching. As expected, *buildFlexList* algorithm outperforms the regular insertion method, since in the *buildFlexList* algorithm the expensive hash calculations are performed only once for each node in the FlexList. So practically, the *buildFlexList* algorithm reduced the time to build a FlexList for a **file of size 400MB** with 200000 blocks **from 12 seconds to 2.3 seconds** and for a **file of size 4GB** with 2000000 blocks **from 128 seconds to 23 seconds**.

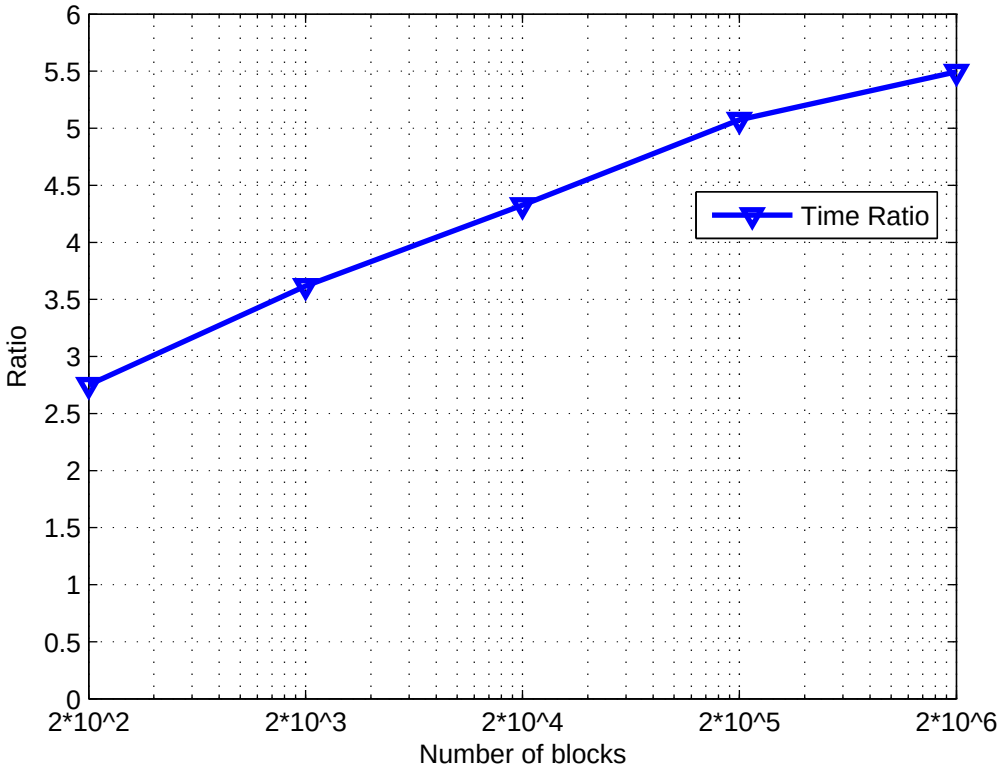

Figure 3.9: Time ratio on buildFlexList algorithm against insertions.

Chapter 4

FLEXDPDP: FLEXIBLE DYNAMIC PROVABLE DATA POSSESSION

In this section, we describe the application of our FlexList to integrity checking in secure cloud storage systems according to the DPDP model [Erway et al., 2009]. Preliminaries and basic definitions for this chapter are given in the beginning. Then, algorithms for the proof generation and verification are discussed and detailed by examples. Then, algorithms for verifiable variable-size updates are presented and detailed by examples. Then, the analysis of these algorithms under various scenarios is given. Finally, the proof generation algorithm is evaluated under energy-efficiency metric.

The DPDP model has two main parties: the client and the server. The cloud server stores a file on behalf of the client. Erway et al. showed that an RBASL can be created on top of the outsourced file to provide proofs of integrity (see Figure 4.1). The following are the algorithms used in the DPDP model for secure cloud storage [Erway et al., 2009]:

- *Challenge* is a probabilistic function run by the client to request a proof of integrity for randomly selected blocks.
- *Prove* is run by the server in response to a challenge to send the proof of possession.
- $Verify$ is a function run by the client upon receipt of the proof. A return value of accept ideally means the file is kept intact by the server.

- *prepareUpdate* is a function run by the client when she changes some part of her data. She sends the update information to the server.
- $performUpdate$ is run by the server in response to an update request to perform the update *and* prove that the update performed reliably.
- $verifyUpdate$ is run by the client upon receipt of the proof of the update. Returns accept (and updates her meta data) if the update was performed reliably.

We construct the above model with FlexList as the authenticated data structure. We provide new capabilities and efficiency gains as discussed in Section 3 and call the resulting scheme **FlexDPDP**. In this section, we describe our corresponding algorithms for each step in the DPDP model.

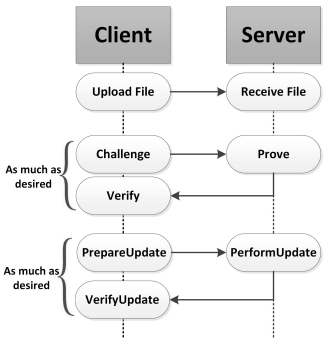

Figure 4.1: Client Server interactions in FlexDPDP.

The FlexDPDP scheme uses *homomorphic verifiable tags* (as in DPDP [Erway et al., 2009]); multiple tags can be combined to obtain a single tag that corresponds to combined blocks [Ateniese et al., 2009]. Tags are small compared to data blocks, enabling storage in memory. Authenticity of the skip list guarantees integrity of tags, and tags protect the integrity of the data blocks.

4.1 Preliminaries

Before providing optimized proof generation and verification algorithms, we introduce essential methods to be used in our algorithms to determine intersection nodes, search multiple nodes, and update rank values. Table 4.1 shows additional notation used in this section.

Symbol	Description
hash	hash value of a node
rs	rank state, indicates the byte count to the left of current node and used to recover i value when roll-back to a state is done
$state$	state, created in order to store from which node the algorithm will continue, contains a node, rank state, and last index
C	challenged indices vector, in ascending order
V	verify challenge vector, reconstructed during verification to check if the proof belongs to challenged blocks, in terms of indices
p	proof node
P	proof vector, stores proof nodes for all challenged blocks
T	tag vector of challenged blocks
M	block sum
$\sqcup_s$	intersection stack, stores states at intersections in $searchMulti$ algorithm
$\sqcup_h$	intersection hash stack, stores hash values to be used at intersections
$\sqcup_i$	index stack, stores pairs of integer values, employed in $updateRankSum$
$\sqcup_l$	changed nodes' stack, stores nodes for later hash calculation, employed in $hashMulti$
$start$	start index in $\sqcup_i$ from which $updateRankSum$ should start
end	end index in $\sqcup_i$
$first$	current index in C
$last$	end index in $\sqcup_s$

Table 4.1: Symbols used in our algorithms.

isIntersection: This function is used when *searchMulti* checks if a given node is an intersection. A node is an intersection point of proof paths of two indices when the first index can be found following the *below* link and the second index is found by following the $after$ link (the challenged indices will be in ascending order). There are two conditions for a node to be called an intersection node:

- The current node follows the *below* link according to the index we are building the proof path for.
- The current node needs to follow the $after$ link to reach the element of challenged indices at index $last$ in the vector C.

If one of the above conditions is not satisfied, then there is no intersection, and the method returns false. Otherwise, it decrements $last$ and continues trying until it finds a node which cannot be found by following the $after$ link and returns $last'$ (to be used

in the next call of *isIntersection*) and true (as the current node *cn* is an intersection point). Note that this method directly returns false if there is only one challenged index.

Algorithm 4.1.1: searchMulti Algorithm

Input: cn, C, $first$, $last$, rs, P, $\sqcup_s$
Output: cn, P, $\sqcup_s$

```
    // Index of the challenged block (key) is calculated according to the
    current sub skip list root
1   i = C_first − rs
    // Create and put proof nodes on the search path of the challenged
    block to the proof vector
2   while Until challenged node is included do
3       p = new proof node with cn.level and cn.r
        // End of this branch of the proof path is when the current node
        reaches the challenged node
4       if cn.level = 0 and i < cn.length then
5           p.setEndFlag(); p.length = cn.length
        //When an intersection is found with another branch of the proof
        path, it is saved to be continued again, this is crucial for the
        outer loop of ''multi'' algorithms
6       if isIntersection(cn, C, i, last_k, rs) then
            //note that last_k becomes last_{k+1} in isIntersection method
7           p.setInterFlag(); state(cn.after, last_k, rs+cn.below.r) is added to ⊔_s //
            Add a state for cn.after to continue from there later
        // Missing fields of the proof node are set according to the link
        current node follows
8       if (CanGoBelow(cn, i)) then
9           p.hash = cn.after.hash; p.rgtOrDwn =dwn
10          cn = cn.below //unless at the leaf level
11      else
12          p.hash = cn.below.hash; p.rgtOrDwn =rgt
            // Set index and rank state values according to how many bytes
            at leaf nodes are passed while following the after link
13          i -= cn.below.r; rs += cn.below.r; cn = cn.after
14      p is added to P
```

Proof node is the building block of a proof, used throughout this section. It contains level, data length (if level is 0), rank, hash, and three boolean values $rgtOrDwn$, end flag and intersection flag. Level and rank values belong to the node for which the proof node is generated. The hash is the hash value of the neighbor node, which is not on the proof path. There are two scenarios for setting hash and $rgtOrDwn$ values:

(1) When the current node follows *below* link, we set the hash of the proof node to the hash of the current node's $after$ and its $rgtOrDwn$ value to *dwn*.

(2) When the current node follows $after$ link, we set the hash of the proof node to the hash of the current node's *below* and its $rgtOrDwn$ value to rgt.

searchMulti (Algorithm 4.1.1): This algorithm is used in *genMultiProof* to generate the proof path for multiple nodes without unnecessary repetitions of proof nodes. Figure 4.2, where we challenge the node at the index 450, clarifies how the algorithm works. Our aim is to provide the proof path for the challenged node. We assume that in the search, the current node *cn* starts at the root (w_1 in our example). Therefore, initially the search index i is 450, the rank state rs and $first$ are zero, the proof vector P and intersection stack $\sqcup_s$ are empty.

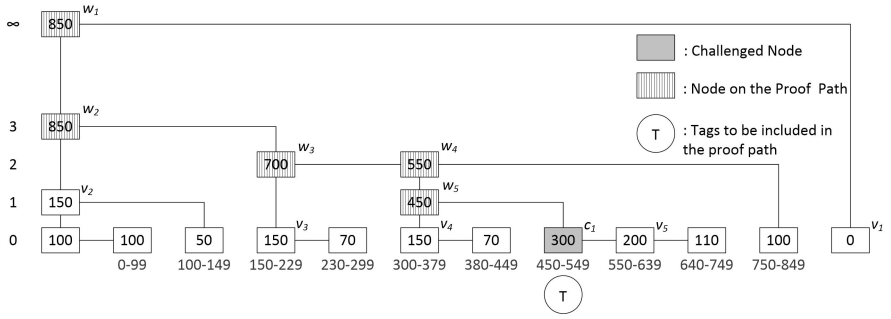

Figure 4.2: Proof path for challenged index 450 in a FlexList.

For w_1, a proof node is generated using scenario (1), where p.hash is set to v_1.hash and $p.rgtOrDwn$ is set to *dwn*. For w_2, the proof node is created as described in scenario (2) above, where p.hash is set to v_2.hash and $p.rgtOrDwn$ is set to rgt. The proof node for w_3 is created using scenario (2). For w_4 and w_5, proof nodes are generated as in scenario (1). The last node c_1 is the challenged leaf node, and the proof node for this node is also created as in scenario (1). Note that in the second, third, and fifth iterations of the while loop, the current node is moved to a sub skip

list (at line 13 in Algorithm 4.1.1). Lines 4-5 (setting the end flag and collecting the data length) and 6-7 (setting intersection flag and saving the state) in Algorithm 4.1.1 are crucial for generation of proof for multiple blocks. We discuss them later in this section.

updateRankSum: This algorithm, used in *verifyMultiProof*, is given the rank difference as input, the verify challenge vector V, and indices *start* and *end* (on V). The output is a modified version of the verify challenge vector V'. The procedure is called when there is a transition from one sub skip list to another (larger one). The method updates entries starting from index *start* to index *end* by rank difference, where rank difference is the size of the larger sub skip list minus the size of the smaller sub skip list.

Finally, tags and combined blocks will be used in our proofs. For this purpose, we use an RSA group Z_N^*, where $\mathrm{N} = pq$ is the product of two large prime numbers, and g is a high-order element in Z_N^* [Erway et al., 2009]. It is important that the server does not know p and q. The tag t of a block m is computed as $t = g^m \mod N$. The block sum is computed as $M = \sum_{i=0}^{|C|} a_i m_{C_i}$ where C is the challenge vector containing block indices and a_i is the random value for the i^{th} challenge.

4.2 Managing Multiple Challenges at Once

Client server interaction (Figure 4.1) starts with the client pre-processing her data (creating a FlexList for the file and calculating tags for each block of the file). The client sends the random seed she used for generating the FlexList to the server along with a public key, data, and the tags. Using the seed, the server constructs a FlexList over the blocks of data and assigns tags to leaf-level nodes. Note that the client may request the root value calculated by the server to verify that the server constructed the correct FlexList over the file. When the client checks and verifies that the hash

of the root value is the same as the one she had calculated, she may safely remove her data and the FlexList. She keeps the root value as meta data for later use in the proof verification mechanism.

To challenge the server, the client generates two random seeds, one for a pseudo-random generator that will generate random indices for bytes to be challenged, and another for a pseudo-random generator that will generate random coefficients to be used in the block sum. The client sends these two seeds to the server as the challenge, and keeps them for verification of the server's response.

4.2.1 Proof Generation

genMultiProof (Algorithm 4.2.1): Upon receipt of the random seeds from the client, the server generates the challenge vector C and random values A accordingly and runs the *genMultiProof* algorithm in order to get tags, file blocks, and the proof path for the challenged indices. The algorithm searches for the leaf node of each challenged index and stores all nodes across the search path in the proof vector. However, we have observed that regular searching for each particular node is inefficient. If we start from the root for each challenged block, there will be a lot of replicated proof nodes. In the example of Figure 4.2, if proofs were generated individually, w_1, w_2, and w_3 would be replicated 4 times, w_4 and w_5 3 times, and c_3 2 times. To overcome this problem we save states at each intersection node. In our *optimal* proof, only one proof node is generated for each node on any proof path. This is beneficial in terms of not only space but also time. The verification time of the client is greatly reduced since she computes less hash values.

We explain *genMultiProof* (Algorithm 4.2.1) using Figure 4.3 and notations in Table 4.1. By taking the index array of challenged nodes as input (challenge vector C generated from the random seed sent by the client contains [170, 320, 470, 660] in

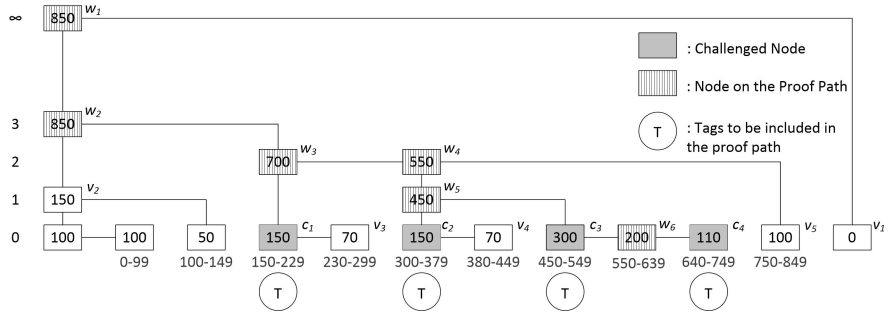

Figure 4.3: Multiple blocks are challenged in a FlexList.

the example), the *genMultiProof* algorithm generates the proof P, collects the tags into the tag vector T, calculates the block sum M at each step, and returns all three. The algorithm starts traversing from the root (w_1 in our example) by retrieving it from the intersection stack $\sqcup_s$ at line 3 of Algorithm 4.2.1. Then, in the loop, we call *searchMulti*, which returns the proof nodes for w_1, w_2, w_3 and c_1. The state of node w_4 is saved in the stack $\sqcup_s$ as it is the $after$ of an intersection node, and the *intersection* flag for proof node for w_3 is set. Note that proof nodes at the intersection points store no hash value. The second iteration starts from w_4, which is the last saved state. New proof nodes for w_4, w_5 and c_2 are added to the proof vector P, while c_3 is added to the stack $\sqcup_s$. The third iteration starts from c_3 and *searchMulti* returns P, after adding c_3 to it. Note that w_6 is added to the stack $\sqcup_s$. In the last iteration, w_6 and c_4 are added to the proof vector P. As the stack $\sqcup_s$ is empty, the loop is over. Note that all proof nodes of the challenged indices have their *end* flags and length values set (line 5 of Algorithm 4.1.1). When *genMultiProof* returns, the output proof vector should be as in Figure 4.4. At the end of the *genMultiProof* algorithm the proof and tag vectors and the block sum are sent to the client for verification.

Algorithm 4.2.1: genMultiProof Algorithm

Input: C, A
Output: T, M, P

`Let C=` $(i_0, \ldots, i_k)$ `where` i_j `is the` $(j+1)^{th}$ `challenged index;`
$A = (a_0, \ldots, a_k)$ `where` a_j `is the` $(j+1)^{th}$ `random value;`
`state`$_m = (node_m, lastIndex_m,\ rs_m)$
1 $cn = root$; $rs = 0$; $M = 0$; $\sqcup_s$, P and T are empty; state(root, k, rs) added to $\sqcup_s$
`// Call` $searchMulti$ `method for each challenged block to fill the proof vector` P
2 **for** $i = 0$ *to* k **do**
3 $state = \sqcup_s$.pop()
4 $cn = searchMulti(state.\text{node}, C, i, state.end, state.rs, P, \sqcup_s)$
`// Store tag of the challenged block and compute the block sum`
5 $cn.tag$ is added to T and M += cn.data*a$_i$

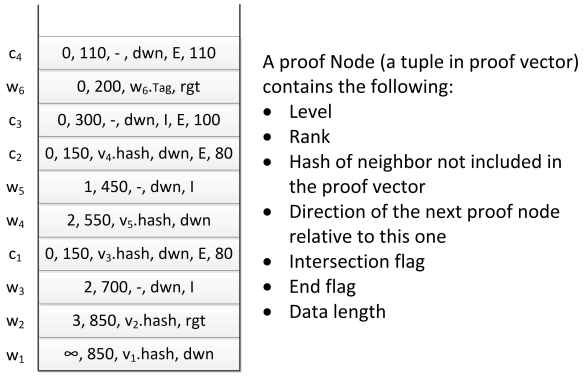

Figure 4.4: Proof vector for Figure 4.3 example.

4.2.2 Proof Verification

verifyMultiProof (Algorithm 4.2.2): Remember that the client keeps random seeds used for the challenge. She generates the challenge vector C and random values A according to these seeds. If the server is honest, these will contain the same values as the ones the server generated. There are two steps in the verification process: tag verification and FlexList verification.

Tag verification is done as follows: Upon receipt of the tag vector T and the block sum M, the client calculates $tag = \prod_{i=0}^{|C|} T_i^{a_i} \mod N$ and accepts iff $tag = g^M \mod N$.

By this, the client checks the integrity of file blocks by tags. Later, when tags are proven to be intact by FlexList verification, the file blocks will be verified. **FlexList verification** involves calculation of hashes for the proof vector P. The hash for each proof node can be calculated in different ways as described below using the example from Figure 4.3 and Figure 4.4.

The hash calculation always has the *level* and *rank* values stored in a proof node as its first two arguments.

- If a proof node is marked as *end* but *not intersection* (e.g., c_4, c_2, and c_1), this means the corresponding node was challenged (to be checked against the challenged indices later), and thus its tag must exist in the tag vector. We compute the corresponding hash value using that tag, the hash value stored in the proof node (null for c_4 since it has no $after$ neighbor, the hash value of v_4 for c_2, and the hash value of v_3 for c_1), and the corresponding length value (110 for c_4, 80 for c_2 and c_1).
- If a proof node is not marked and $rgtOrDwn = rgt$ or $level = 0$ (e.g., w_6, w_2), this means the $after$ neighbor of the node is included in the proof vector and the hash value of its *below* is included in the associated proof node (if the node is at leaf level, the tag is included instead). Therefore we compute the corresponding hash value using the hash value stored in the corresponding proof node and the previously calculated hash value (hash of c_4 is used for w_6, hash of w_3 is used for w_2).
- If a proof node is marked as *intersection* and *end* (e.g., c_3), this means the corresponding node was both challenged (thus its tag must exist in the tag vector) and is on the proof path of another challenged node; therefore, its $after$ neighbor is also included in the proof vector. We compute the corresponding hash value using the corresponding tag from the tag vector and the previously

calculated hash value (hash of w_6 for c_3).

- If a proof node is marked as *intersection* but *not end* (e.g., w_5 and w_3), this means the node was not challenged but both its $after$ and *below* are included in the proof vector. Hence, we compute the corresponding hash value using the previously calculated two hash values (the hash values calculated for c_2 and for c_3, respectively, are used for w_5, and the hash values calculated for c_1 and for w_4, respectively, are used for w_3).
- If none of the above is satisfied, this means a proof node has only $rgtOrDwn = dwn$ (e.g., w_4 and w_1), meaning the *below* neighbor of the node is included in the proof vector. Therefore we compute the corresponding hash value using the previously calculated hash value (hash of w_5 is used for w_4, and hash of w_2 is used for w_1) and the hash value stored in the corresponding proof node.

We treat the proof vector (Figure 4.4) as a stack and do necessary calculations as discussed above. The calculation of hashes is done in the reverse order of the proof generation in *genMultiProof* algorithm. Therefore, we perform the calculations in the following order: c_4, c_6, c_3, c_2, w_5, ... until the hash value for the root (the last element in the stack) is computed. Observe that to compute the hash value for w_5, the hash values for c_3 and c_2 are needed, and this reverse (top-down) ordering always satisfies these dependencies. Finally, we compute the corresponding hash values for w_2 and w_1. When the hash for the last proof node of the proof path is calculated, it is compared with the meta data that the client possesses (in line 22 of Algorithm 4.2.2).

The check above makes sure that the nodes, whose proofs were sent, are indeed in the FlexList that correspond to the meta data stored at the client. But the client also has to make sure that the server indeed proved storage of data that she challenged.

Algorithm 4.2.2: verifyMultiProof Algorithm

Input: C, P, T, *MetaData*
Output: accept or reject

`Let` $P = (A_0, \ldots, A_k)$`, where` $A_j = (level_j,\ r_j,$ `hash`$_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ `for` $j = 0, \ldots, k$`;` $T = (tag_0, \ldots, tag_n)$`, where` $tag_m =$ `tag for challenged` $block_m$ `for` $m = 0, \ldots, n$`;`

1 $start = n$; $end = n$; $t = n$; $V = 0$; hash $= 0$; hash$_{prev} = 0$; $startTemp = 0$; $\sqcup_h$ and $\sqcup_i$ are empty stacks
`// Process each proof node from the end to calculate hash of the root and indices of the challenged blocks`
2 **for** $j = k$ *to* 0 **do**
3 **if** $isEnd_j$ *and* $isInter_j$ **then**
4 hash $= hash(level_j, r_j, tag_t,$hash$_{prev}, length_j)$; decrement(t)
5 $updateRankSum(length_j, V, start, end)$; decrement($start$) `// Update index values of challenged blocks on the leaf level of current part of the proof path`
6 **else if** $isEnd_j$ **then**
7 **if** $t \neq n$ **then**
8 hash$_{prev}$ is added to $\sqcup_h$
9 ($start$, end) is added to $\sqcup_i$
10 decrement($start$); $end = start$
11 hash $= hash(level_j, r_j, tag_t,$hash$_j, length_j)$; decrement(t)
12 **else if** $isInter_j$ **then**
13 ($startTemp$,end) $= \sqcup_i$.pop()
14 $updateRankSum(r_{prev}, V, startTemp, end)$ `// Last stored indices of challenged block are updated to rank state of the current intersection`
15 hash $= hash(level_j, r_j,$ hash$_{prev}, \sqcup_h.pop())$
16 **else if** $rgtOrDwn_j = rgt$ *or* $level_j = 0$ **then**
17 hash $= hash(level_j, r_j,$hash$_j,$hash$_{prev})$
18 $updateRankSum(r_j - r_{prev}, V, start, end)$ `// Update indices of challenged blocks, which are on the current part of the proof path`
19 **else**
20 hash $= hash(level_j, r_j,$ hash$_{prev},$ hash$_j)$
21 hash$_{prev}$ = hash; $r_{prev} = r_j$
`//`$endnodes$ `is a vector of proof nodes marked as End in the order of appearance in` P
22 **if** $\forall a,\ 0 \leq a \leq n\ ,\ 0 \leq C_a - V_a < endnodes_{n-a}.length$ *OR* $hash \neq MetaData$ **then**
23 return reject
24 return accept

The server may have lost those blocks but may instead be proving storage of some other blocks at different indices. To prevent this, the verify challenge vector, which contains the start indices of the challenged nodes (150, 300, 450, and 460 in our example), is generated by the rank values included in the proof vector (in lines 5, 9, 10, 13, 14, and 18 of Algorithm 4.2.2). With the start indices and the lengths of the

challenged nodes given, we check if each challenged index is included in a node that the proof is generated for (as shown in line 22 of Algorithm 4.2.2). For instance, we know that we challenged index 170, c_1 starts from 150 and is of length 80. We check if $0 \leq 170 - 150 < 80$. Such a check is performed for each challenged index and each proof node with an *end* mark.

4.3 Verifiable Variable-size Updates

The main purpose of the insert, remove, and modify operations (update operations) of our FlexList being employed in the cloud setting is that we want the update operations to be verifiable. The purpose of the following algorithms is to verify the update operation and compute new meta data to be stored at the client through the proof sent by the server.

4.3.1 Update Execution

performUpdate is run at the server side upon receipt of an update request to the index i from the client. We consider it to have three parts: *proveModify, proveInsert, proveRemove*. The server runs *genMultiProof* algorithm to acquire a proof vector in a way that it covers the nodes which may get affected from the update. For a modify operation the modified index (i), for an insert operation the left neighbor of the insert position (i-*1*), and for a remove operation the left neighbor of the remove position and the node at the remove position (i-*1*, i) are to be used as challenged indices for *genMultiProof* Algorithm. Then the server performs the update operation as it is using the regular FlexList algorithms, and sends the new meta data to the client.

4.3.2 Update Verification

The algorithm *verifyUpdate* of the DPDP model, in our construction, not only updates her meta data but also verifies if it is correctly updated at the server by checking whether or not the calculated meta data and the received one are equal. It makes use of one of the following three algorithms due to the nature of the update, at the client side.

Algorithm 4.3.1: verifyModify Algorithm

Input: C, P, T, *tag*, *data*, *MetaData*, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$

`Let C=` (i_0) `where` i_0 `is the modified index;` $P = (A_0, \ldots, A_k)$`, where` $A_j = (\ level_j,\ r_j,\ \texttt{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ `for` $j = 0, \ldots, k$`;` $T = (tag_0)$`, where` tag_0 `is tag for` $block_0$ `before modification;` P`,` T `are the proof and tag before the modification;` *tag* `and` *data* `are the new tag and data of the modified block`

```
1  if !VerifyMultiProof(C, P, T, MetaData) then
2      return reject;
3  else
4      i = size(P) - 1
5      hash = hash(A_i.level, A_i.rank - A_i.length + data.length, tag, A_i.hash,
       data.length)
       // Calculate hash values until the root of the Flexlist
6      MetaData_new = calculateRemainingHashes( i-1, hash, data.length - A_i.length,
       P)
7      if MetaData_byServer = MetaData_new then
8          Metadata = MetaData_new
9          return accept
10     else
11         return reject
```

verifyModify (Algorithm 4.3.1) is run at the client to approve the modification. The client alters the last element of the received proof vector and calculates temp meta data accordingly. Later she checks if the new meta data provided by the server is equal to the one that the client has calculated. If they are the same, then modification is accepted, otherwise rejected.

verifyInsert (Algorithm 4.3.2) is run to verify the correct insertion of a new block to the FlexList, using the proof vector and the new meta data sent by the server. It calculates the temp meta data using the proof P as if the new node has been inserted in it. The inputs are the challenged block index, a proof, the tags, and the new block

information. The output is accept if the temp root calculated is equal to the meta data sent by the server, otherwise reject.

Algorithm 4.3.2: verifyInsert Algorithm

Input: C, P, T, *tag*, *data*, *level*, *MetaData*, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$

Let C= (i_0) where i_0 is the index of the left neighbor; P = ($A_0, \ldots, A_k$), where A_j = ($level_j$, r_j, hash_j, $rgtOrDwn_j$, $isInter_j$, $isEnd_j, length_j$) for $j = 0, \ldots, k$; T = (tag_0) where tag_0 is for precedent node of newly inserted node; P, T are the proof and tag before the insertion; *tag*, *data* and *level* are the new tag, data and level of the inserted block

1 **if** *!VerifyMultiProof(C, P, T, MetaData)* **then**
2 return reject;
3 **else**
4 i = size(P) - 1; rank = A_i.length; rankTower = A_i.rank - A_i.length + *data*.length
5 hashTower = *hash*(0, rankTower, *tag*, A_i.hash, *data*.length)
6 **if** *level*$\neq$ *0* **then**
7 hash = *hash*(0, A_i.length, tag_0, 0);
8 decrement(i)
9 **while** $A_i.level \neq level$ *or* ($A_i.level = level$ *and* $A_i.rgtOrDwn = dwn$) **do**
10 **if** $A_i.rgtOrDwn = rgt$ **then**
11 rank += A_i.rank - A_{i+1}.rank
// A_i.length is added to hash calculation if A_i.level = 0
12 hash = *hash*(A_i.level, rank, A_i.hash, hash)
13 **else**
14 rankTower += A_i.rank - A_{i+1}.rank
15 hashTower = *hash*(A_i.level, rankTower, hashTower, A_i.hash)
16 decrement(i)
17 hash = *hash*(*level*, rank + rankTower, hash, hashTower)
18 $MetaData_{new}$ = calculateRemainingHashes(i, hash, *data*.length, P)
19 **if** $MetaData_{byServer} = MetaData_{new}$ **then**
20 MetaData = $MetaData_{new}$
21 return accept
22 return reject

The algorithm is explained using Figure 4.5 as an example where a verifiable insert at index 450 occurs. The algorithm starts with the computation of the hash values for the proof node n_3 as *hashTower* at line 5 and v_2 as *hash* at line 7. Then the loop handles all proof nodes until the intersection point of the newly inserted node n_3 and the precedent node v_2. In the loop, the first iteration calculates the hash value for v_1 as *hash*. The second iteration yields a new *hashTower* using the proof node for d_2. The same happens for the third iteration but using the proof node for d_1. Then the hash value for the proof node c_3 is calculated as *hash*, and the same operation is

done for c_2. The hash value for the proof node m_1 (intersection point) is computed by taking *hash* and *hashTower*. Following this, the algorithm calculates all remaining hash values until the root. The last hash value computed is the hash of the root, which is the temp meta data. If the server's meta data for the updated FlexList is the same as the newly computed temp meta data, then the meta data stored at the client is updated with this new version.

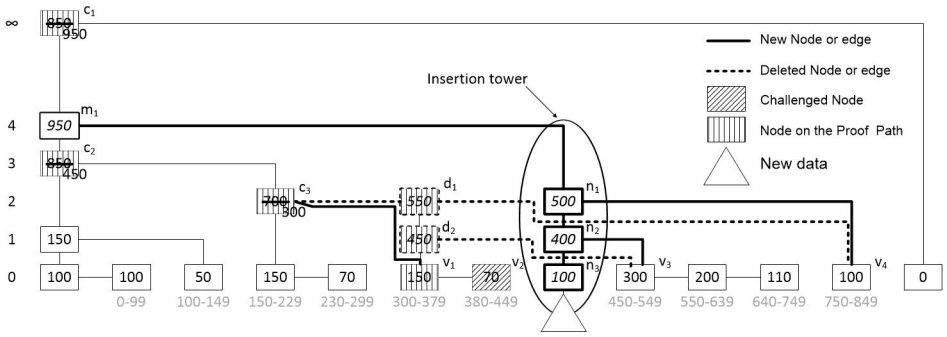

Figure 4.5: Verifiable insert example.

verifyRemove (Algorithm 4.3.3) is run to verify the correct removal of a block in the FlexList, using the proof and the new meta data by the server. Proof vector P is generated for the left neighbor and the node to be deleted. It calculates the temp meta data using the proof P as if the node has been removed. The inputs are the proof, a tag, and the new block information. The output is accept if the temp root calculated is equal to the meta data from the server, otherwise reject.

Algorithm 4.3.3: verifyRemove Algorithm

Input: C, P, T, *MetaData*, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$

```
Let C= (i0, i1) where i0, i1 are the index of the left neighbor and the
removed index respectively; P = (A0, ..., Ak), where Aj = (levelj, rj,
hashj, rgtOrDwnj, isInterj, isEndj, lengthj) for j = 0, ..., k; T
= (tag0, tag1) where tag1 is tag value for deleted node and tag0 is for
its precedent node ; P, T are the proof and tags before the removal;
```

```
1  if !VerifyMultiProof(C, P, T, MetaData) then
2      return reject
3  else
4      dn = size(P) - 1; i = size(P) - 2; last = dn
5      while !Ai.isEnd do
6          decrement(i)
7      rank = Adn.rank; hash = hash(0, rank, tag0, Adn.hash, Adn.length)
8      decrement(dn)
9      if !Adn.isEnd or !Ai.isInter then
10         decrement(i)
11     while !Adn.isEnd or !Ai.isInter do
12         if Ai.level < Adn.level or Adn.isEnd then
13             rank += Ai.rank - Ai+1.rank
               // Ai.length is added to hash calculation if Ai.level = 0
14             hash = hash(Ai.level, rank, Ai.hash, hash)
15             decrement(i)
16         else
17             rank += Adn.rank - Adn+1.rank
18             hash = hash(Adn.level, rank, hash, Adn.hash)
19             decrement(dn)
20     decrement(i)
21     MetaDatanew = calculateRemainingHashes(i, hash, Alast.length, P)
22     if MetaDatabyServer = MetaDatanew then
23         MetaData = MetaDatanew
24         return accept
25     return reject
```

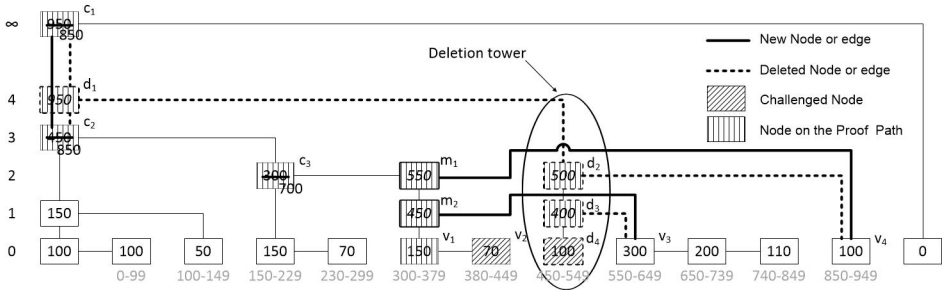

Figure 4.6: Verifiable remove example.

The algorithm will be discussed through the example in Figure 4.6, where a ver-

ifiable remove occurs at index 450. The algorithm starts by placing iterators i and dn at the position of v_2 (line 6) and d_4 (line 4), respectively. At line 7, the hash value (*hash*) for the node v_2 is computed using the hash information at d_4. dn is then updated to point at node d_3 at line 8. The loop is used to calculate the hash values for the newly added nodes in the FlexList using the hash information in the proof nodes of the deleted nodes. The hash value for v_1 is computed by using *hash* in the first iteration. The second and third iterations of the loop calculate the hash values for m_2 and m_1 by using hash values stored at the proof nodes of d_3 and d_2 respectively. Then the hash calculation is done for c_3 by using the hash of m_1. After the hash of c_2 is computed using the hash of c_3, the algorithm calculates the hashes until the root. The hash of the root is the temp meta data. If the server's meta data for the updated FlexList is verified using the newly computed temp meta data, then the meta data stored at the client is updated with this new version.

4.4 Performance Analysis

Proof Generation Performance : Figure 4.7 shows the server proof generation time for FlexDPDP as a function of the block size by fixing the file size to 16MB, 160MB, and 1600MB. As shown in the figure, with the increase in block size, the time required for the proof generation increases, since with a higher block size, the block sum generation takes more time. Interestingly though, with extremely small block sizes, the number of nodes in the FlexList become so large that it dominates the proof generation time. Since 2KB block size worked best for various file sizes, our other tests employ 2KB blocks. These 2KB blocks are kept **on the hard disk drive**, on the other hand the FlexList nodes are much smaller and subject to be **kept in RAM**. While we observed that *buildFlexList* algorithm runs faster with bigger block sizes (since there will be fewer blocks), the creation of a FlexList happens only once.

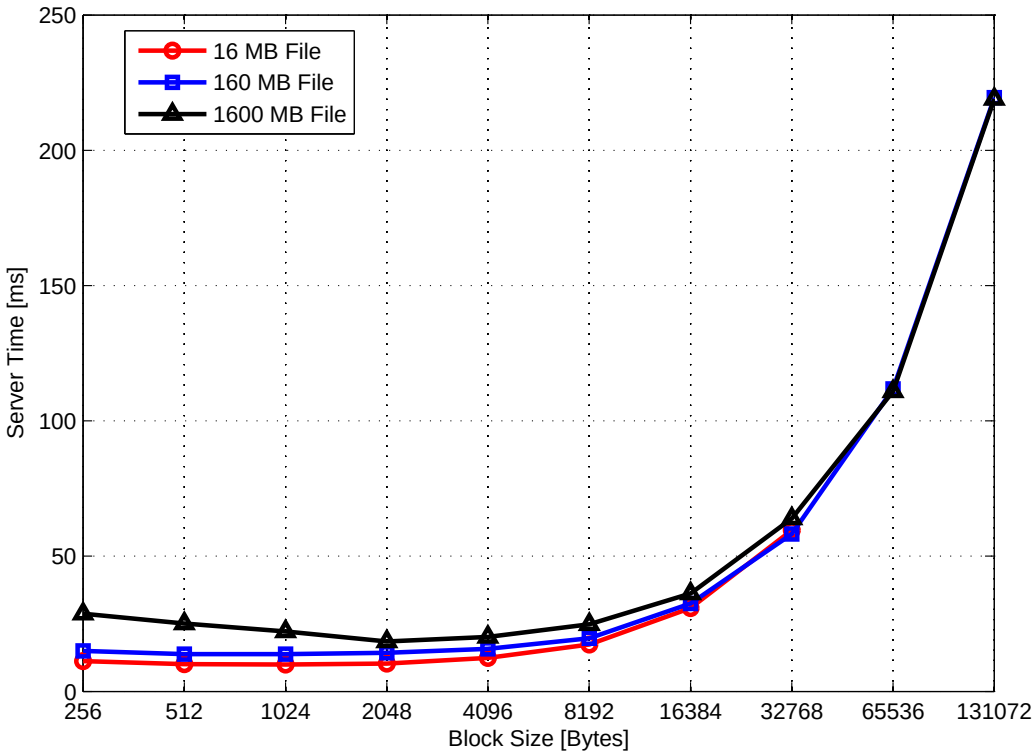

Figure 4.7: Server time for 460 random challenges as a function of block size for various file sizes.

On the other hand, the proof generation algorithm runs periodically depending on the client, therefore we chose to optimize for its running time.

The performance of our optimized implementation of the proof generation mechanism is evaluated in terms of communication and computation. We take into consideration the case where the client wishes to detect with more than 99% probability if more than a 1% of her 1GB data is corrupted by challenging 460 blocks; the same scenario as in PDP and DPDP [Ateniese et al., 2007, Erway et al., 2009]. In our experiment, we used a FlexList with 500,000 nodes, where the block size is 2KB.

In Figure 4.8 we plot the ratio of the unoptimized proofs over our optimized proofs in terms of the **FlexList proof** size and computation, as a function of the

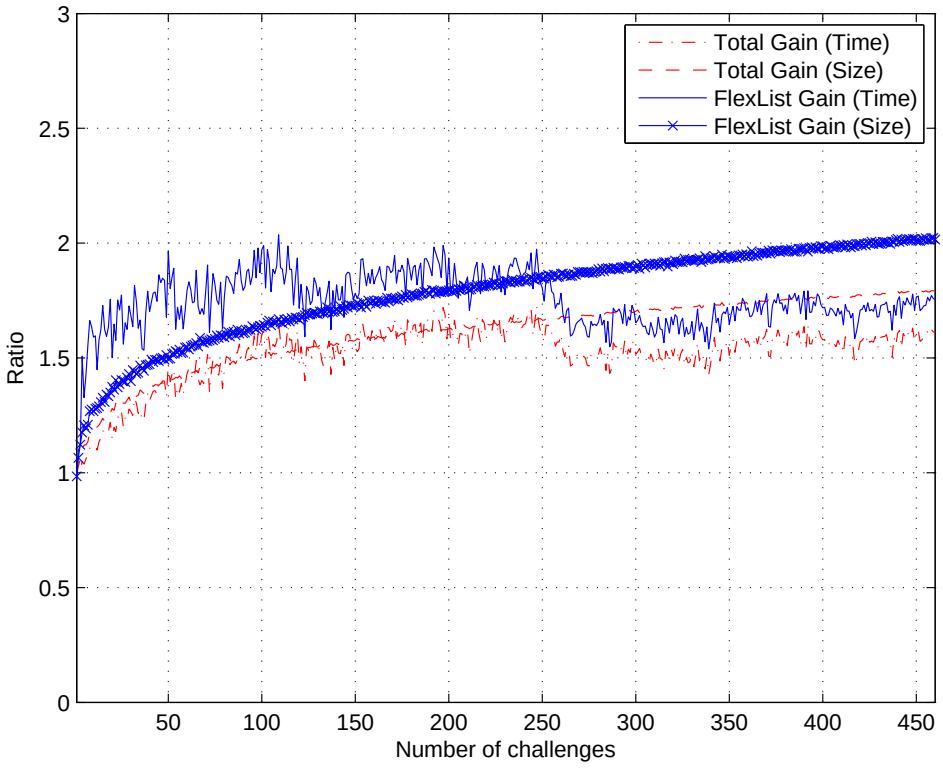

Figure 4.8: Performance gain graph ([460 single proof / 1 multi proof] for 460 challenges).

number of challenged nodes. The unoptimized proofs correspond to proving each block separately, instead of using our *genMultiProof* algorithm for all of them at once. Our multi-proof optimization results in **40**% **computation and 50**% **communication gains** for FlexList proofs. This corresponds to FlexList proofs being up to **1.75 times as fast** and **2 times as small**.

We also measure the gain in the total size of a **FlexDPDP proof** and computation done by the server in Figure 4.8. With our optimizations, we clearly see a gain of about **35**% and **40**% **for the overall computation and communication**, respectively, corresponding to proofs being up to **1.60 times as fast** and **1.75 times as small**. The whole proof roughly consists of 213KB FlexList proof, 57KB

of tags, and 2KB of block sum. Thus, for 460 challenges as suggested by PDP and DPDP [Ateniese et al., 2007, Erway et al., 2009], we obtain a decrease in total **proof size from 485KB to 272KB**, and the computation is **reduced from 19ms to 12.5ms** by employing our *genMultiProof* algorithm. We could have employed gzip to eliminate duplicates in the proof, but it does not perfectly handle the duplicates and our algorithm also provide computation (proof generation and verification) time optimization as well. Compression is still beneficial when applied on our optimal proof.

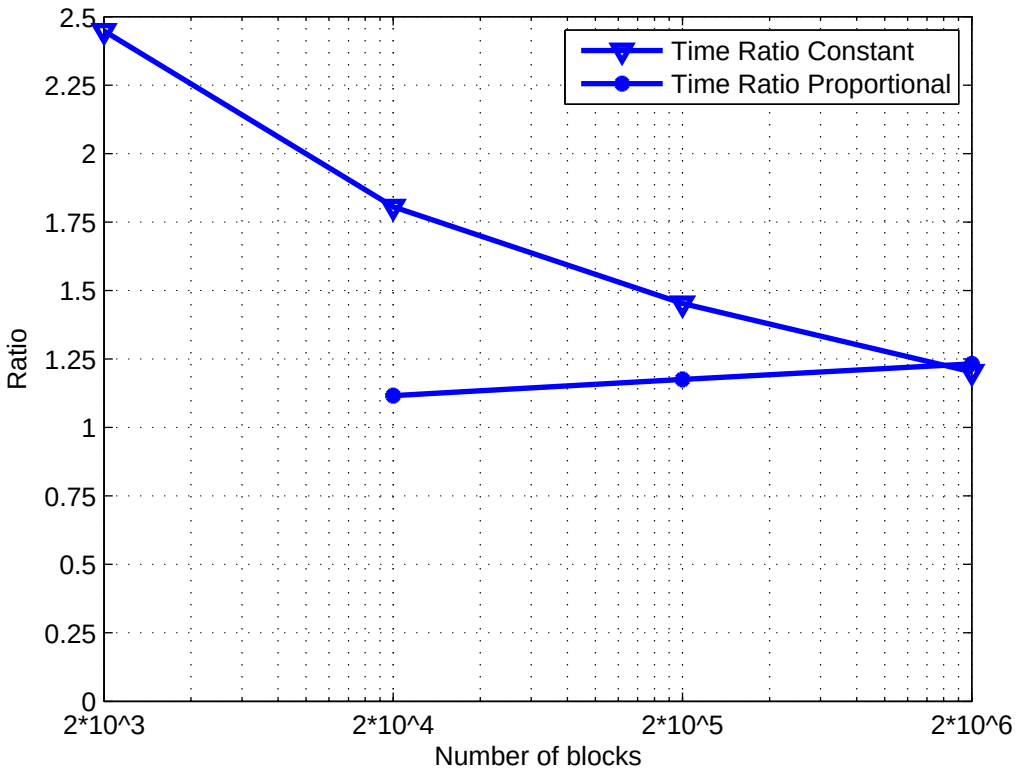

Figure 4.9: Time ratio on *genMultiProof* algorithm.

Furthermore, we tested the performance of *genMultiProof* algorithm in terms of time efficiency. The time ratio graph for the *genMultiProof* algorithm is shown in

Figure 4.9. We have tested the algorithm in different file size scenarios, starting a file size from 4MB to 4GB (where block size is 2KB, and thus the number of blocks increase with the file size). In *constant* scenario we applied the same challenge size of 460. Our results showed a relative decline in the performance of the *genMultiProof* as the number of blocks in the FlexList increases. This is caused by the number of challenges being constant. Because as the number of blocks in the FlexList grows, the number of repeated proof nodes in the proof decreases. In *proportional* scenario, we have the time ratio for 5, 46 and 460 challenges for the block number of 20000, 200000 and 2000000 respectively. The graph shows a relative incline in the performance of *genMultiProof* for the proportional number of challenges to the number of blocks in a file. The algorithm has a clear efficiency gain in the computation time in comparison to the generating each proof individually.

Provable Update Performance: In FlexDPDP, we have optimized algorithms for verifiable update operations. The results for the basic functions of the FlexList ($insert, modify, remove$) against their verifiable versions are shown in Figure 4.10. The regular $insert$ method takes more time than any other method, since it needs extra time for the memory allocations and I/O delay. The $remove$ method takes less time than the $modify$ method, because there is no I/O delay and at the end of the $remove$ algorithm there are less nodes that need recalculation of the hash and rank values. As expected, the complexity of the FlexList operations increase logarithmically. The verifiable versions of the functions require an average overhead of 0.05 ms for a single run. For a **single verifiable insert**, the server **needs less than 0.4ms** to produce a proof in a FlexList with 2 million blocks (corresponding to a 4GB file). These results show that the verifiable versions of the updates can be employed with only little overhead.

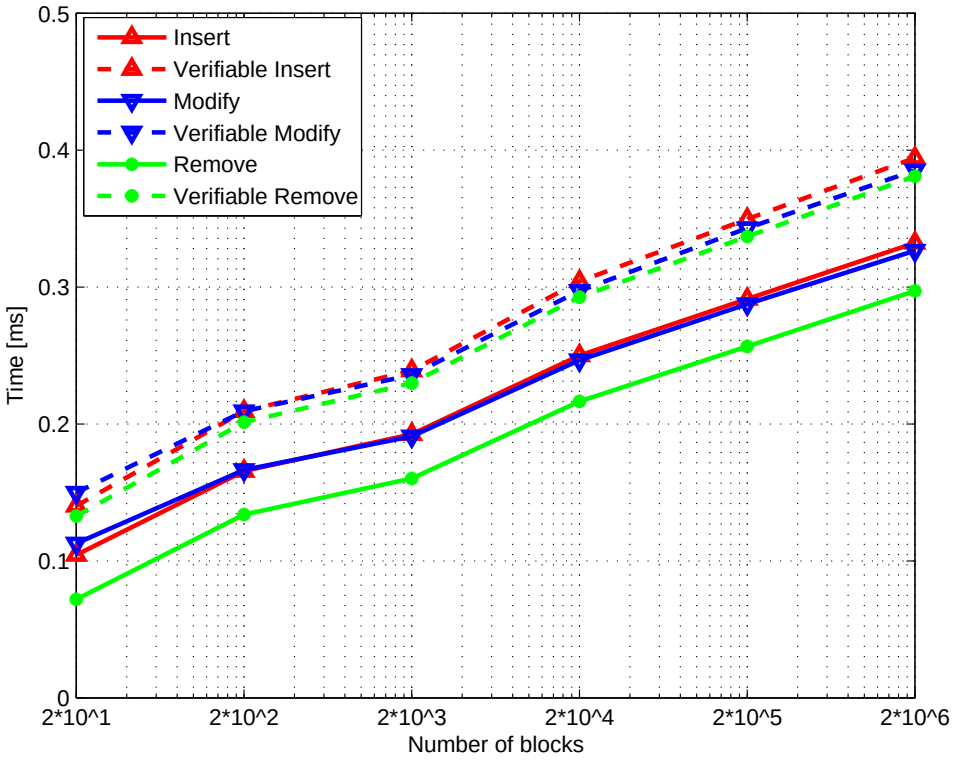

Figure 4.10: Performance evaluation of FlexList methods and their verifiable versions.

4.5 Energy-Efficiency Analysis

For energy efficiency tests (Figure 4.11 and 4.12), we used Watts up Pro meter. It measures the total energy consumption of the connected device. We conducted the tests and took their energy consumption measurements. Then, we measured the average energy cost for the idle time when no tests were taking place. The difference between these two measurements were used in the calculation of the results. Energy consumption and time (CPU) gain results are close for both graphs. For the energy efficiency tests we do not take I/O delay into account. Therefore, we argue that energy efficiency of our algorithms is directly impacted by the CPU usage time of them. Therefore, our algorithms that optimize operations in the FlexList creation

and multi-challenge proof generation are efficient in terms of both time and energy.

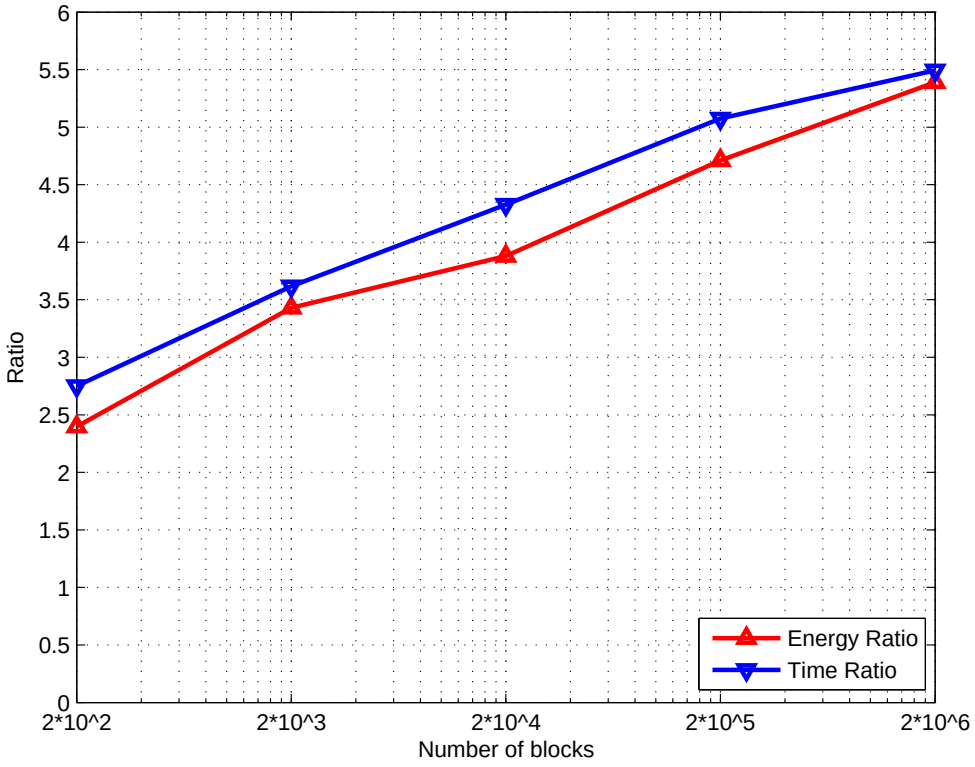

Figure 4.11: Time and energy ratios on buildFlexList algorithm against insertions.

We presented a novel algorithm for the efficient building of a FlexList. Figure 3.9 demonstrates energy consumption and time ratios between the *buildFlexList* algorithm and building FlexList by means of insertion (in sorted order). The time ratio is calculated by dividing the time spent for the building FlexList using insertion method by the time needed by the *buildFlexList* algorithm. The same ratio equation is applied to the energy consumption ratio calculation. In our energy-time ratio experiments, we do not take into account the disk access time; therefore there is no delay for I/O switching. The energy and time ratio values are close to each other because of the same reason: the more time, the algorithm executes, the more energy is spent.

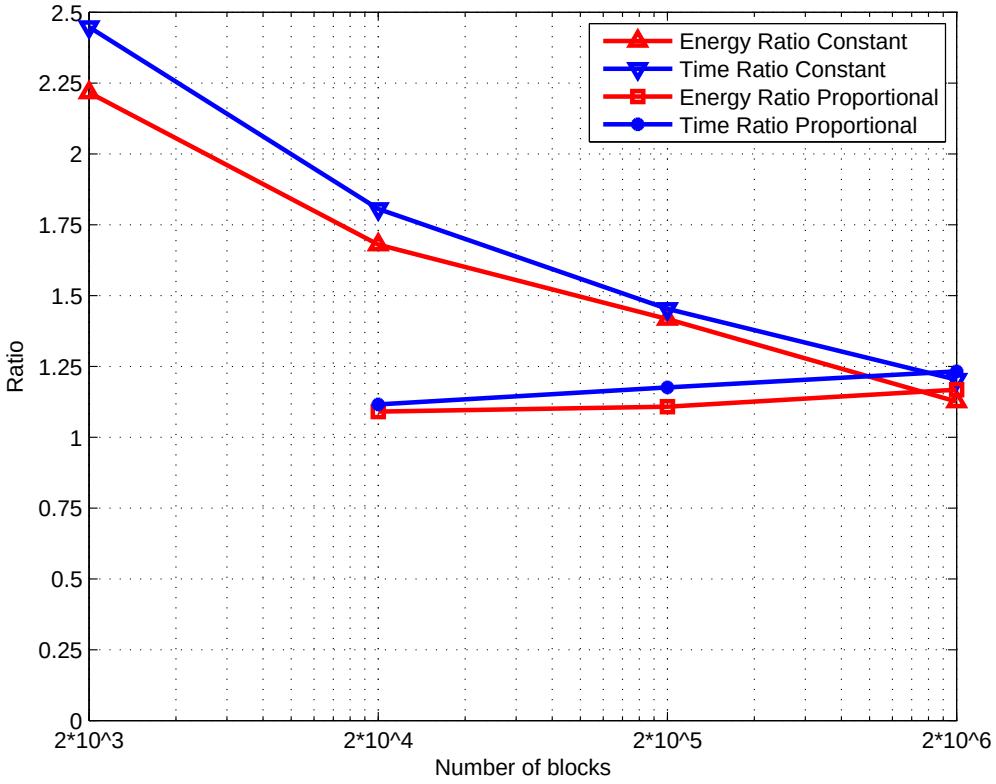

Figure 4.12: Time and energy ratios on *genMultiProof* algorithm.

Furthermore, we tested the performance of *genMultiProof* algorithm in terms of energy efficiency [Kachkeev et al., 2013]. The time and energy ratio graph for the *genMultiProof* algorithm is shown in Figure 4.9. We have tested the algorithm in different file size scenarios, starting a file size from 4MB to 4GB (where block size is 2KB, and thus the number of blocks increase with the file size). In *constant* scenario we applied the same challenge size of 460. Our results showed a relative decline in the performance of the *genMultiProof* as the number of blocks in the FlexList increases. This is caused by the number of challenges being constant. Because as the number of blocks in the FlexList grows, the number of repeated proof nodes in the proof decreases. In *proportional* scenario, we have the time and energy ratio for 5, 46 and

460 challenges for the block number of 20000, 200000 and 2000000 respectively.

Chapter 5

INTERLOCAL: INTEGRITY AND REPLICATION GUARANTEED LOCALITY-BASED PEER-TO-PEER STORAGE SYSTEM

In this chapter, we propose a novel locality-based peer-to-peer storage system with integrity and replication guarantees called InterLocal. First, preliminaries such as a skip graph and landmark multidimensional scaling (LMDS) are discussed. Then, locality-based skip graph with search algorithm employing membership vectors is presented. Finally, the replication mechanism that places client's files to the physically closest neighbor is presented.

5.1 Skip Graph

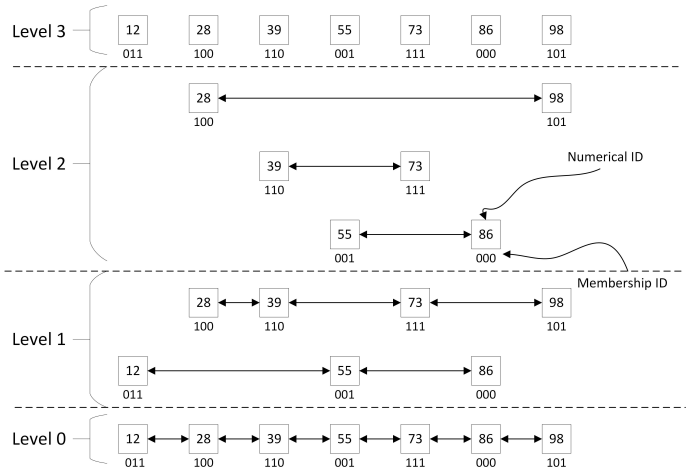

Figure 5.1: An example of a skip graph.

In this section, we give a short description of skip graph, which is a backbone of the InterLocal. Skip graph [Aspnes, 2003] (Figure 5.1) consists of sparse and sorted doubly-linked lists divided by levels, with starting level at 0. On level 0, all nodes are connected in a sequence. Skip graph can be viewed as distributed version of skip lists [Pugh, 1990b]. The difference between them is that the skip graph may have many lists at level i, and each node participates only in one of them, until the nodes are divided into singletons after $O(logn)$ levels on average, where n is the number of nodes. A skip graph supports regular DHT operations, such as *put* and *get*, as well as skip list operations like search, insert and delete. An operation get(x) is done using numerical ID on a skip graph on a set K of keys is returning the greatest key x' $\in$ K such that x' $\leq$ x. Another value stored at each node is name ID called membership vector of the node. Membership of a given key y $\in$ set K' of keys in the list at some level i is determined by the first i bits in the possibly infinite sequence of random bits related to y, which is called *membership vector* of y as denoted as $m(y)$. The first i bits of the membership vector $m(y)$ is therefore denoted by $m(y)|i$. Two keys y and y' appear in the same list at level i if and only if their first i bits of membership vectors $m(y)$ and $m(y')$ are identical i.e. $m(y)|i = m(y')|i$ [Aspnes, 2003].

There are two possible ways to search in a skip graph. It can be done by searching using numerical ID and name ID. Regular get(x) operation as in all popular DHT systems is done employing numerical ID. The search algorithm initially starts from the top level of the node received request, then at every other node it continues passing request to the next neighbor with the closest key. If the received key x is greater than its own key then it checks for the neighbor with key k such that k $\leq$ x. The node checks for appropriate neighbors at level i if no found checks at level below, as soon as an appropriate neighbor is found, the request is continued at that node. At the level 0 when the node with the request has no node to send the request, it sends a reply to the owner of the request pointing itself as the searched node. An example of

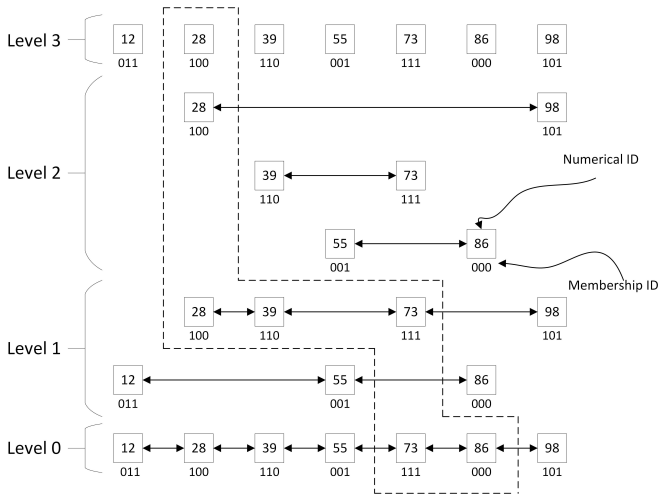

Figure 5.2: An example of search algorithm using numerical ID.

search algorithm is shown in Figure 5.2. The numerical ID (key x) given as input is 86. The key of the node received the request is 28. Since 28 is less than 86 we check right neighbors for the keys less than 86. We find it at the level 1 with the key 39. Then we check its right neighbors as well and move to the node with the key 73 at the same level 1. Lastly, we see that the right neighbor at the level 0 has key less or equal to our searched key x. We pass the request to that node, and that node replies to the request owner.

Another search algorithm uses the name ID (membership vector) and initially starts from the level 0 (bottom of a skip graph). As an input algorithm receives a name ID. The idea of the search is to find the node which has the biggest common prefix in name ID with the given one. The node received request checks its first i bits of the membership vector at level i ($m(x)|i$) with its right or left neighbors. If at least one of them has *i same prefix bits* as $m(x)$, then the node makes the same operation at the level i+1. Otherwise it forwards the reguest to one of the neighbors (cannot send to the neighbor it received request from). Then the neighbor continues from the same level i, and searches for appropriate neighbors. The node which gets to the top

level, sends a reply to the owner of the request.

5.2 Landmark Multidimensional Scaling

Multidimensional scaling (MDS) is a set of techniques for data analysis where the structure of distance-like data is displayed as a geometrical picture [Wang, 2011]. There are different types of MDS, however the original and best-known approach is *classical MDS*. In a classical MDS there is one similarity matrix and the Euclidean distance applied to calculate the similarities. It is based on an efficient matrix algorithm that finds a Euclidean embedding which exactly preserves the metric given on the input data. The complexity of the algorithm is approximately $O(kN^2)$, where N is the number of data points and k is the dimension of the embedding [Wang, 2011]. However, when the number of the points becomes too large, the run of the algorithm becomes too expensive as well. Therefore, in InterLocal we employ *Landmark Multidimensional Scaling (LMDS)* [de Silva and Tenenbaum, 2004], which require a set of landmark points and their distance matrix. In the distance matrix we put the average latency values between the landmark nodes and the other nodes. Assume that we are given a set of N nodes and we want to embed them in the Euclidean space R^k. Landmark MDS consists of the following steps [de Silva and Tenenbaum, 2004] :

1. Define a set of n landmark nodes
2. Employ classical MDS to compute k x n matrix L which represents an embedding of the n landmark nodes in R^k. n x n distance matrix D_n should be used as an input.
3. Distance-based triangulation should be applied to find a k x N matrix X representing an embedding of the N nodes in R^k. An input to that calculation is the n x N matrix $D_{n,N}$ of latencies between landmark nodes and other nodes. The new computer coordinates are derived from the squared distances by an affine

linear transformation

5.3 InterLocal construction

InterLocal is a locality-based peer-to-peer storage system with integrity and replication guarantees. The core part of the system is a locality-based skip graph, which enables efficient and local placement of replicas and as well as content retrieval (search) operation. The replication mechanism plays pivotal role in the client's efficient access to the files. File operations are the basic methods that are performed as a new file is received, a proof of possession is requested or a file update is sent.

5.3.1 Locality-based skip graph

Locality-based skip graph construction consists of two phases of construction as in Figure 5.3. First phase starts with the selection of n landmark nodes. Afterwards, each of the chosen nodes measures latency values (l_1, l_2, l_3) to every other landmark node, all combining to the distance matrix n x n. We employ classical MDS algorithm as in [Wang, 2011] to receive a 1 x n matrix with locality information for each of the landmark nodes. Then, we build a skip graph using the locality information as membership vectors for the nodes. Second phase starts with a new node joining the system as in Figure 5.3. The new node measures latencies (l_4, l_5, l_6) to the landmark nodes and prepares 1 x n distance matrix. Then node uses LDMS algorithm to compute its own relative locality information in order to join the skip graph.

To visualize our locality-based skip graph, we present a map of peers and their locality information in Figure 5.4. We use locality information of these peers and build a locality-based skip graph, where some of the peers are landmark nodes. To use these locality information in a skip graph, we transform them to bits as shown in Figure 5.4. Obviously, peers that are further away from each other have a larger

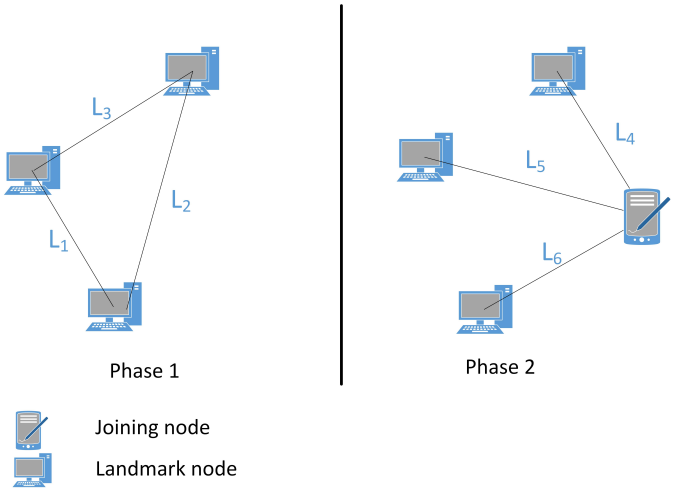

Figure 5.3: A node joining a distributed system using LMDS.

difference between their numbers.

5.3.2 Search in a locality-based skip graph

Algorithm 5.3.1: Search by nameID Algorithm

Input: givenNameID, length, dir, checkbit, startNode
Output:

```
1  while length < max + 1 do
2      if m(givenNameID)|length ≠ m(this.nameID)|length then
3          decrement(length)
4          exit
5      increment(length)
6  if this.neighbors[dir][length] ≠ NULL then
7      send to this.neighbors[dir][length] SearchByName (givenNameID, length, dir,
       checkbit, startNode)
8  else if checkbit is true then
9      checkbit set to false
10     send to this.neighbors[!dir][length] SearchByName (givenNameID, length, dir,
       checkbit, startNode)
11 else
12     send to startNode FoundByName (this.nameID, length, dir, checkbit,
       this.address)
```

Search is one of the most basic and crucial operations in peer-to-peer system.

Figure 5.4: A map with the location information of peers.

Search by name id was first mentioned as an idea to find neighbors for a random skip graph [Aspnes, 2003]. In InterLocal, we have a search by name id algorithm, which performs search using membership vectors (name ids). An input to the algorithm are a name id, length of common bits, direction *dir*, bit for changing direction *checkbit* and address of the initiator of the search as *startNode*. As an output, information for the node with the most common bits is sent to the initiator. A sample search is presented in Figure 5.3.1 to give a more detailed explanation of this operation. The search can start from any node, in our case it started from node with the name id = 1110 (14) and the given name id = 0100 (4). Since even the first bits of these two membership vectors are not equal, we send the request right (direction) neighbor

with name id = 0111 since it has longer common prefix than the left neighbor. Then, we compare the membership vector of that node with the given name id and we have first two bits in common, so we check neighbor at level 2. We have only one neighbor at left with name id = 0101, and we transfer the request to it and changing *checkbit* to false, because we have to change the direction of the search from right to left. The next neighbor its name id with the given name id and also sends it to its left neighbor with name id = 0100. And it occurs to be the searched node with the given name equal to its name id, and it replies to the initiator by sending its information. If we look at the map in Figure 5.4 and connect nodes on the search path(14, 7, 5, 4), then we see transitions from neighbor to neighbor. The idea of the search by name id is to follow the efficient path to the searched locality.

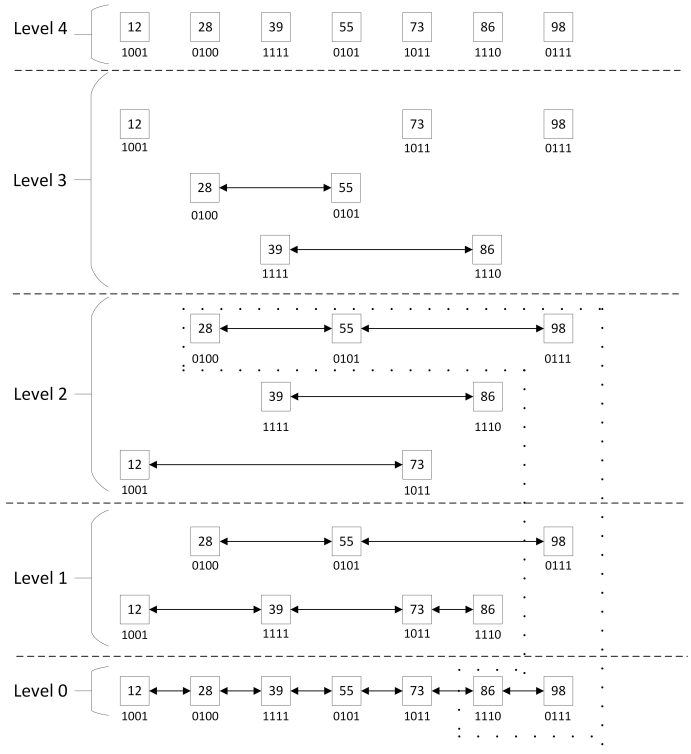

Figure 5.5: A locality-based skip graph.

5.3.3 File operations

File upload

Once a LMDS skip graph is built, clients can upload their files to the system. A client wants to place a file to the physically close neighbor but before that the client has to compute its own membership vector by getting the locality information. Then, we employ the search by name ID (membership vector) to find an appropriate peer. The peer, which receives a file from the client, becomes the main replica holder.

Proof of possession

The client already uploaded a file and wishes to check the data integrity of her file. Each node having a replica of a file also stores a FlexList for that file in memory (a FlexList is relatively small). So when a client sends a request for the proof of possession, any replica holder receiving the request can process it and send the proof of possession.

File update

In case of updates, the client prepares the update it wants to be applied at all replicas of the file and sends it using membership vector information to the nearest replica holder. The update is delivered to the main replica holder, which processes and applies update to its FlexList and the file. Afterwards, it sends the update to a subset of replicas in the system and the updates are slowly propagated to all replicas.

5.3.4 Locality-based Replication

Replication of the data items is crucial mechanism in InterLocal. Since our locality-based skip graph structure produces the advantage of finding physically closest neighbors, the replication of data items should be done near the client. The replication

system gives the client opportunity to send, replicate and update the data with the minimal network communication and time consumption.

Replica creation

The client initially uploads a file to one of the peers, which becomes a main replica holder. The duties of the main replica holder include the control over the total number of replicas for the particular file, the peer selection for new replicas, the selection of backup peers and the maintenance of replica management mechanism. Upon the receipt of a file, the main replica holder makes a selection between its neighbors for appropriate replica holders. It starts from top level, if a neighbor has enough free space above the storage threshold then it stores a new replica. The main replica holder starts from the top level, since the closest neighbors are connected from the higher levels due the connections based on the membership vectors. The number of replicas for a file depends on the availability churns of the nodes.

Backup peers

Replica placement according to the client's position is advantageous. However, we cannot place all replicas near the client since the availability of the peers in the system is subject to alterations and natural disasters may cause even small cities to loose power or internet connection. Therefore, a backup mechanism should be introduced. A backup mechanism consists of several peers that reside at some distance from the client in case if the client looses its neighbors. In a trivial implementation of a such backup mechanism, extra replicas are placed according to the hash value of file name. In case of malfunctioning of the replica holders, these backup peers will not only replicate data items to the nearest position to the client but also perform the regular file operations (upload, update). As soon as some of the neighbors of the client get the file replica, the neighbors take over all the work with the client. The backup peers

perform the maintenance among all replica peers, checking whether some peers left the system and if so new replicas should be introduced. Moreover, the number of backup peers directly depends on the churn of the network peers.

Replica maintenance

Due to the nature of peer-to-peer systems, nodes leave and join the system constantly. Therefore, a peer leaving the system should be noticed and its duties should be delegated to another peer. For each file, there are entities like main replica holder, replica holders and backup peers. Each entity has a list and periodically sends a short message to one of the peers in the list, which is chosen completely randomly. If a main replica holder for a file leaves the system, then the replica holder with the closest membership vector to it becomes new main replica holder. In case if all replica holders leave simultaneously, the backup peers recreate the replica holders at the peers with the closest position to the client. On the other hand, if the backup peers leave the system, then the replica holders delegate their duties to the other nodes. Algorithm 5.3.2 shows the steps taken during a maintenance. It takes $List$ of peers having replica of the file and $file$ as an input. It basically checks a part of $List$ for a liveness, and if needed creates new replicas or delegates replication to the nodes with the same type. There are two helper methods called FindNewHolder and DelegateReplication. The first one performs a search and places a replica, the second one delegates the replica placement to another node, which searches and places replica.

Algorithm 5.3.2: Replica Maintenance Algorithm

Input: List, file
Output:

```
1   counter = 0 and iter = size(List) * p
    p is the size of maintenance
2   while counter < iter do
3       peer = random from 0 to size(List)
4       if List[peer] is not alive then
5           if List[peer].type == this.type then
6               FindNewHolder (List, file, List[peer]);
7           else
8               send to node in List with same type DelegateReplication (List, file,
                List[peer]);
9       increment(counter)
```

Chapter 6

PERFORMANCE ANALYSIS ON THE PLANETLAB

In this chapter, we present the experimental analysis for InterLocal and a regular skip graph based storage system. First, the evaluation of the basic skip graph functions for both systems is given. Then, the results for provable integrity operations such as proof of possession and update are discussed. Lastly, the analysis of replication mechanism for both systems under scenario where nodes are gradually leaving the system is presented.

6.1 Evaluation settings

We have developed and deployed the implementations of both a skip graph and FlexDPDP on the real-world peer-to-peer test-bed PlanetLab. Both implementations are done in C++, we have employed *Cashlib* as a main library [Brownie Points Project, , Meiklejohn et al., 2010] and *Asio* library [Boost C++ Libraries,] for the network communication protocols. Security parameters for FlexDPDP are 1024-bit RSA modulus, 80-bit random numbers, and SHA-1 hash function, overall resulting in an expected security of 80-bits. In all tests, blocks are stored separately on the hard disk and therefore we include I/O access time. Moreover, the size of a Flexlist is small enough to keep it in RAM. In our experiments we used a skip graph of size from 40 to 50 nodes. The number of landmark nodes is 8. All our results are the average of 10 runs.

6.2 Evaluation of the Skip Graph Operations

In our tests, we implemented two systems based on a skip graph. First system is a regular skip graph with the random membership vectors. Second one is our system with locality information according to the physical position of the nodes (InterLocal). We have tested these two implementations for variety of scenarios. Our main purpose is to observe a substantial efficiency of InterLocal over a system based on regular skip graph. Therefore, experiments include measurements on different storage related operations like search, upload, proof receipt, update.

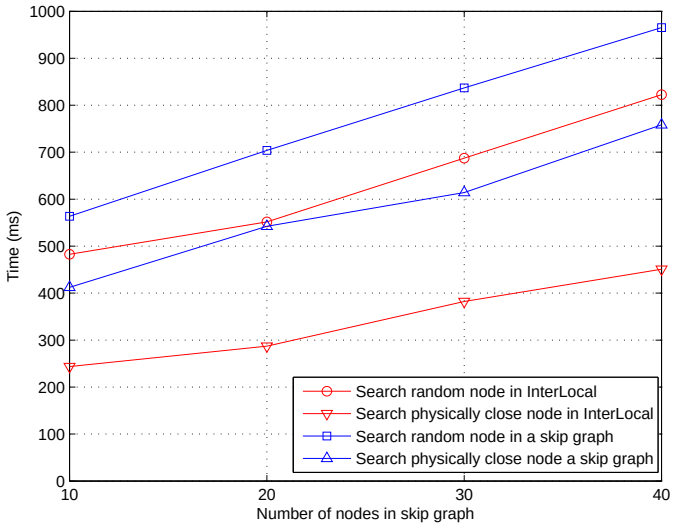

Figure 6.1: Search in a skip graph based system and InterLocal.

Search is a basic operation used in the most distributed systems. We tested search algorithm according to the physical positions of the nodes in a skip graph. The selection of peers and the searched peers are the same in both systems. So we wanted to test them in the same environment. There are two main test cases: search for a random node and search for a physically close node. In case of random node it is trivial, in other case we are looking for the node which is physically near the node,

which starts the search operation. For an example in a **skip graph of size 40**, our solution yields a search result for physically close node in **approximately 450 ms**. The random skip graph with the same size spends **around 750 ms** to get the search result. Moreover, the search for random nodes performed better than random search. Since in the random search, request may be sent from one corner of the system to the other and back again. It is caused by the random connections of the neighbor nodes. In our case, the connections are based on the locality information therefore at high levels (depends on the skip graph size) the request will be passed to relatively close nodes.

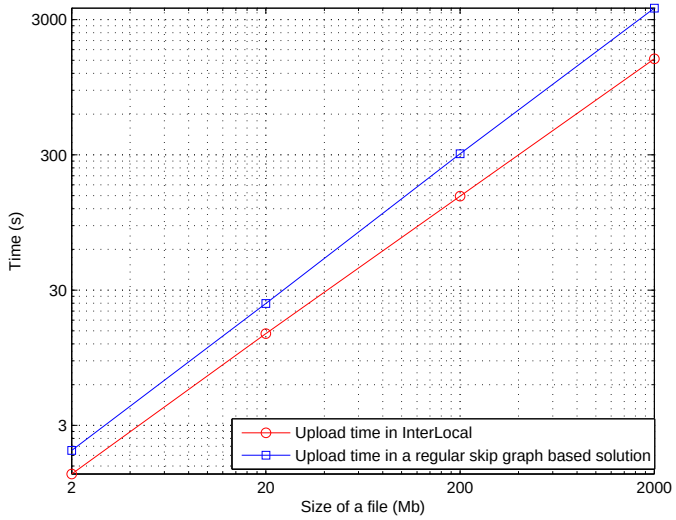

Figure 6.2: Time measurement on upload times for a regular skip graph based solution and InterLocal.

The upload time plays crucial role in the client's network communication, since the upload may take most of the time and power from the client's resources. For upload time measurement we employed file sizes of 2, 20, 200 and 2000 Mb for the realistic scenario. As the result we received time measurements for both systems, from the Figure 6.2 we observe that for a file upload InterLocal outperforms the solution with

a regular skip graph. For an example, for a **file upload of size 2000 Mb** a client in our system requires **around 1500 seconds**. Client in the other system for the same file needs more than **3500 seconds**.

6.3 Evaluation of the Provable Integrity Operations

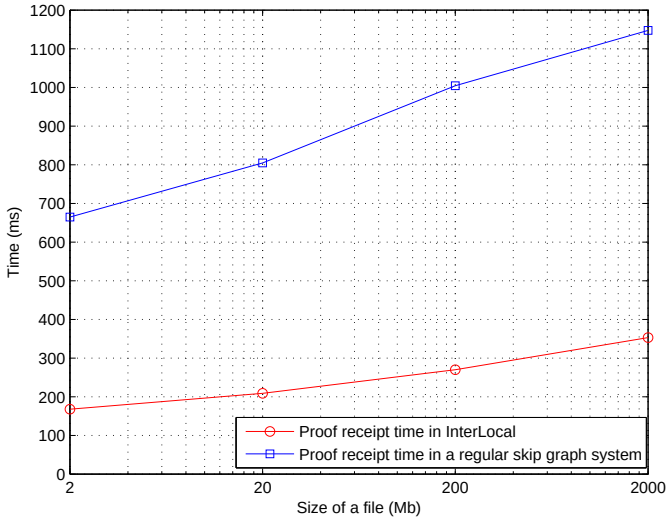

Figure 6.3: Time measurement on proof receipt for a regular skip graph based system and InterLocal.

The client wishes to be convinced that her data is intact at the peers in the system. Therefore, she periodically challenges (requests proof of possession) replicas to check data integrity of her files. We tested it for different file sizes, ranging from 2 to 2000 Mb (Figure 6.3). The client in the system using a regular skip graph receives the proof for **the file size of 2000Mb in less then 1200ms**. However, our system based on LMDS delivers the proof receipt to the client in **less than 400ms**.

Dynamic storage systems provide ability to update the data files. Therefore, the

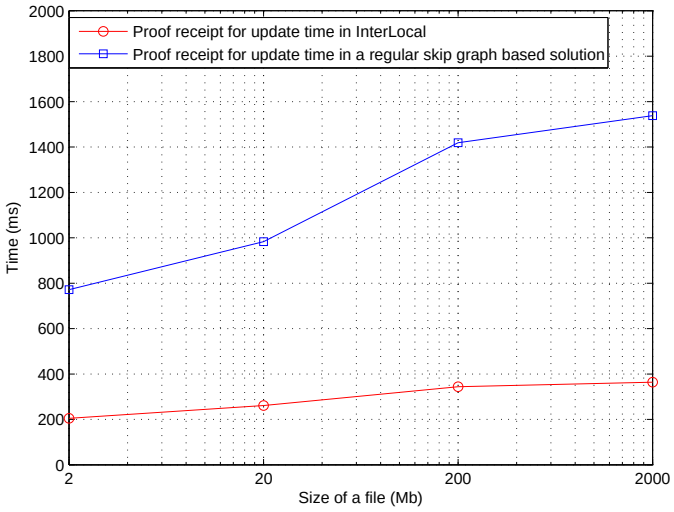

Figure 6.4: Update operation experiment with a fixed update size of 125 Kb.

update operation performance should be analysed as well. The client prepares an update for her file. In Figure 6.4, we have a constant size of an update of 125 Kb and file sizes 2, 20, 200, 2000 Mb. We can see that for a fixed sized 125 Kb update for the **file of 2000 Mb**, the client **waits 1500 ms** for an update proof in regular skip graph based system. In InterLocal, the client receives an update proof in **less than 400 ms**. We chose an update size of 125 Kb to show that even if an average size of an update is 125 Kb, still our system performs well under different file sizes. The other case is the measurement of the dynamic update sizes of 30, 60, 125, 250 and 500 Kb for the fixed 200 Mb file. An example our locality-based system requires **approximately 750 ms** to produce and deliver the proof for **an update of size 500 Kb**. On the other hand, the system based regular skip graph computes and transfers the proof **in 2250 ms**.

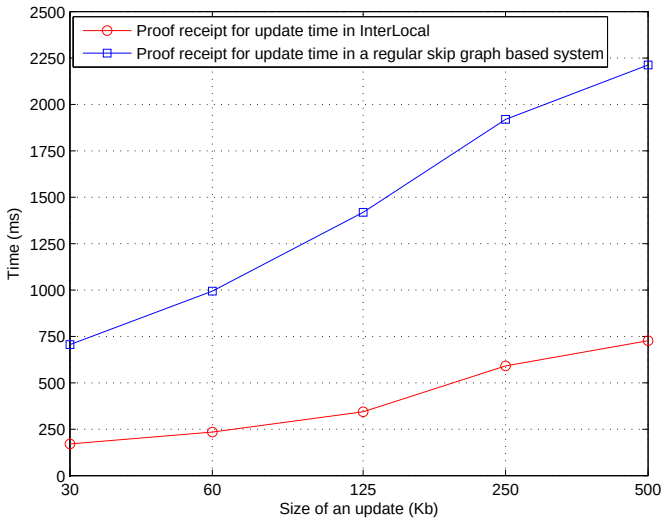

Figure 6.5: Update operation test with a fixed file size of 200 Mb.

6.4 Evaluation of the Replication Operations

Replication mechanism in InterLocal is based on locality of the peers. The client in InterLocal wishes to perform file operations like upload, receive proof of possession, or update in minimized access time. Therefore, replicas of a file should be placed to physically close neighbors of the client. In our test scenario we have a regular skip graph based system, InterLocal (60/40) with 60% replicas physically close to the client (local) and 40% replicas placed according to the hash of the file name (randomly), and InterLocal (40/60) with 40% locally and 60% randomly. We have tested the scenarios where the nodes with replicas are gradually leaving the system to measure degradation of the replication system. For that purpose, we tested systems with requests for proof of possession receipts. Note that, each system has 13 replicas and at InterLocal, the nodes with locally placed replicas leave first. Our experiment results showed that InterLocal (60/40) is **3x faster** and InterLocal (40/60) is **2x faster** than a regular skip graph solution. Moreover, a worst case performance for

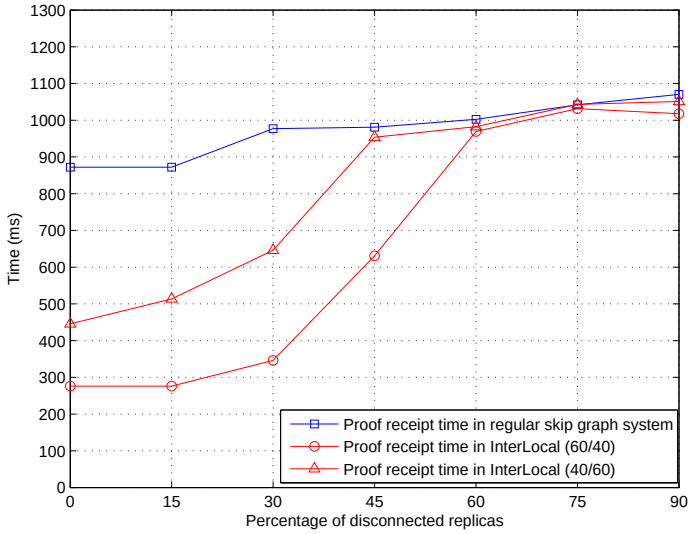

Figure 6.6: Replication performance when nodes are leaving a system.

both InterLocal systems is equal to that of a regular skip graph based system. A slight time difference between InterLocal (60/40) and InterLocal (40/60) in the beginning of the experiment when all replica holders are still alive may be caused by relatively small size of the skip graph to the replication number.

Chapter 7

CONCLUSIONS

Peer-to-peer storage systems have been developed and enhanced for the past decade. There are a number of criteria to make storage system adoptable to public such as data integrity, possibility of efficient updates and adequate replication mechanism. However, an efficient data integrity verification scheme requires a good authenticated data structure. We have checked and researched previous works for cloud storage based storage systems. Early works have shown that the static solutions with optimal complexity [Ateniese et al., 2007, Shacham and Waters, 2008], and the dynamic solutions with logarithmic complexity [Erway et al., 2009] are within reach. However, a DPDP [Erway et al., 2009] solution is not applicable to real life scenarios since it supports only fixed block size and therefore lacks flexibility on the data updates, while the real life updates are likely not of constant block size. We have extended earlier studies in several ways and provided a new data structure (FlexList) and its optimized implementation for use in the cloud data storage. A FlexList efficiently supports variable block sized dynamic provable updates, and we showed how to handle multiple proofs and updates at once, greatly improving scalability. Furthermore, we provided a novel algorithm to build FlexList from a scratch. It greatly reduces time complexity of the FlexList construction. Finally, we evaluate proof generation and build FlexList algorithms for time, size and energy efficiency.

A DPDP with FlexList is the dynamic provable data possession scheme that we use in InterLocal, a novel locality-based peer-to-peer storage systems with integrity and replication guarantees. InterLocal provides a locality-based search capabilities to

efficiently locate data items. Moreover, replication mechanism creates replica of a file physically near the data owner to minimize response time.

As a future work, we plan to further study parallelism to develop and implement algorithms for FlexDPDP operations. It can be extended to distributed and replicated servers for multiple access purpose. One of the other extensions to our work may be recovery mechanism or backup system for data loss prevention in case if data is corrupted or erased. In peer-to-peer settings, InterLocal can be further enhanced to provided load balancing and security measures against malicious nodes.

BIBLIOGRAPHY

[Anagnostopoulos et al., 2001] Anagnostopoulos, A., Goodrich, M. T., and Tamassia, R. (2001). Persistent authenticated dictionaries and their applications. In *ISC*.

[Aspnes, 2003] Aspnes, J. (2003). Skip graphs. In *in SODA*, pages 384–393.

[Ateniese et al., 2007] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., and Song, D. (2007). Provable data possession at untrusted stores. In *ACM CCS*.

[Ateniese et al., 2009] Ateniese, G., Kamara, S., and Katz, J. (2009). Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*.

[Ateniese et al., 2008] Ateniese, G., Pietro, R. D., Mancini, L. V., and Tsudik, G. (2008). Scalable and efficient provable data possession. In *SecureComm*.

[Battista and Palazzi, 2007] Battista, G. D. and Palazzi, B. (2007). Authenticated relational tables and authenticated skip lists. In *DBSec*.

[Blibech and Gabillon, 2005] Blibech, K. and Gabillon, A. (2005). Chronos: an authenticated dictionary based on skip lists for timestamping systems. In *SWS*.

[Blibech and Gabillon, 2006] Blibech, K. and Gabillon, A. (2006). A new timestamping scheme based on skip lists. In *ICCSA (3)*.

[Boehm et al., 1995] Boehm, H.-J., Atkinson, R., and Plass, M. (1995). Ropes: an alternative to strings. *Software: Practice and Experience*, 25.

[Boost C++ Libraries,] Boost C++ Libraries. Boost asio library. http://www.boost.org/doc/libs.

[Bowers et al., 2009] Bowers, K. D., Juels, A., and Oprea, A. (2009). Hail: a high-

availability and integrity layer for cloud storage. In *ACM CCS*.

[Brownie Points Project,] Brownie Points Project. Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.

[Cash et al., 2013] Cash, D., Küpçü, A., and Wichs, D. (2013). Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*.

[Clarke et al., 2001] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001). Freenet: A distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 46–66. Springer-Verlag New York, Inc.

[Crosby and Wallach, 2011] Crosby, S. A. and Wallach, D. S. (2011). Authenticated dictionaries: Real-world costs and trade-offs. *ACM TISSEC*.

[Curtmola et al., 2008] Curtmola, R., Khan, O., Burns, R., and Ateniese, G. (2008). Mr-pdp: Multiple-replica provable data possession. In *ICDCS*.

[Dabek et al., 2001] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with cfs.

[de Silva and Tenenbaum, 2004] de Silva, V. and Tenenbaum, J. (2004). Sparse multidimensional scaling using landmark points. Technical report, Stanford University.

[Dodis et al., 2009] Dodis, Y., Vadhan, S., and Wichs, D. (2009). Proofs of retrievability via hardness amplification. In *TCC*.

[Druschel and Rowstron, 2001] Druschel, P. and Rowstron, A. (2001). Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80.

[Erway et al., 2009] Erway, C., Küpçü, A., Papamanthou, C., and Tamassia, R. (2009). Dynamic provable data possession. In *ACM CCS*.

[Esiner et al., 2013] Esiner, E., Kachkeev, A., Küpçü, A., and Özkasap, Ö. (2013). Flexdpdp: Flexlist-based optimized dynamic provable data possession. *Under Sub-*

mission.

[Etemad and Küpçü, 2013] Etemad, M. and Küpçü, A. (2013). Transparent, distributed, and replicated dynamic provable data possession. In *ACNS*.

[Foster, 1973] Foster, C. C. (1973). A generalization of avl trees. *Commun. ACM*.

[Gnutella,] Gnutella. Gnutella. http://www.gnutelliums.com.

[Goodrich et al., 2007] Goodrich, M. T., Papamanthou, C., and Tamassia, R. (2007). On the cost of persistence and authentication in skip lists. In *Proceedings of the 6th international conference on Experimental algorithms*.

[Goodrich et al., 2008] Goodrich, M. T., Papamanthou, C., Tamassia, R., and Triandopoulos, N. (2008). Athos: Efficient authentication of outsourced file systems. In *ISC*.

[Goodrich et al., 2009] Goodrich, M. T., Sun, J. Z., Tamassia, R., and Triandopoulos, N. (2009). Reliable resource searching in p2p networks. In *SecureComm*, pages 437–447.

[Goodrich and Tamassia, 2001] Goodrich, M. T. and Tamassia, R. (2001). Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute.

[Goodrich et al., 2001] Goodrich, M. T., Tamassia, R., and Schwerin, A. (2001). Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA*.

[Harvey et al., 2003] Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. (2003). Skipnet: A scalable overlay network with practical locality properties.

[Juels and Kaliski., 2007] Juels, A. and Kaliski., B. S. (2007). PORs: Proofs of retrievability for large files. In *ACM CCS*.

[Kachkeev et al., 2013] Kachkeev, A., Esiner, E., Küpçü, A., and Özkasap, Ö. (2013). Energy efficiency in secure and dynamic cloud storage systems. In *EE-LSDS*.

[Kubiatowicz et al., 2000] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: An architecture for global-scale persistent storage. pages 190–201.

[Mager et al.,] Mager, T., Biersack, E., and Michiardi, P. A measurement study of the wuala on-line storage service. In *IEEE P2P 2012, Tarragona, Spain*.

[Maniatis and Baker, 2003] Maniatis, P. and Baker, M. (2003). Authenticated append-only skip lists. *Acta Mathematica*.

[Martins et al., 2006] Martins, V., Pacitti, E., and Valduriez, P. (2006). Survey of data replication in P2P systems. Technical report.

[Meiklejohn et al., 2010] Meiklejohn, S., Erway, C., Küpçü, A., Hinkle, T., and Lysyanskaya, A. (2010). Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*.

[Merkle, 1987] Merkle, R. (1987). A digital signature based on a conventional encryption function. *LNCS*.

[napster,] napster. Napster. http://www.napster.com.

[Papamanthou and Tamassia, 2007] Papamanthou, C. and Tamassia, R. (2007). Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*.

[Polivy and Tamassia, 2002] Polivy, D. J. and Tamassia, R. (2002). Authenticating distributed data using web services and xml signatures. In *In Proc. ACM Workshop on XML Security*.

[Pugh, 1990a] Pugh, W. (1990a). A skip list cookbook. Technical report.

[Pugh, 1990b] Pugh, W. (1990b). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable content-addressable network. In *IN PROC. ACM SIGCOMM 2001*, pages 161–172.

[Rowstron and Druschel, 2001a] Rowstron, A. and Druschel, P. (2001a). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.

[Rowstron and Druschel, 2001b] Rowstron, A. and Druschel, P. (2001b). Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. pages 188–201.

[Saroiu et al., 2002] Saroiu, S., Gummadi, P. K., and Gribble, S. D. (2002). A measurement study of peer-to-peer file sharing systems.

[Shacham and Waters, 2008] Shacham, H. and Waters, B. (2008). Compact proofs of retrievability. In *ASIACRYPT*.

[Stanton et al., 2010] Stanton, P. T., McKeown, B., Burns, R. C., and Ateniese, G. (2010). Fastad: an authenticated directory for billions of objects. *SIGOPS Oper. Syst. Rev.*

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160.

[Tamassia and Tri, 2007] Tamassia, R. and Tri, N. (2007). Efficient content authentication in peer-to-peer networks. In *Proc. ACNS*, pages 354–372.

[Wang, 2011] Wang, J. (2011). *Geometric Structure of High-Dimensional Data and Dimensionality Reduction*. Springer.

[Wang et al., 2009] Wang, Q., Wang, C., Li, J., Ren, K., and Lou, W. (2009). Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*.

[Zhang and Blanton, 2013] Zhang, Y. and Blanton, M. (2013). Efficient dynamic provable data possession of remote data via balanced update trees. *ASIA CCS*.

[Zhao et al., 2001] Zhao, B. Y., Kubiatowicz, J., Joseph, A. D., Zhao, B. Y., Kubiatowicz, J., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report.

[Zheng and Xu, 2011] Zheng, Q. and Xu, S. (2011). Fair and dynamic proofs of retrievability. In *CODASPY*.

VITA

Adilet Kachkeev was born in Bishkek, Kyrgyzstan on August 14, 1989. He received his B.S degree in Computer Engineering from Fatih University, Istanbul in 2011. In September 2011, he joined M.Sc. Program in Computer Science and Engineering at Koç University as a research and teaching assistant. During his studies he worked on secure cloud secure storage system and locality-based peer-to-peer storage system. He has co-authored a conference paper in EE-LSDS'13 one journal (under submission) and one conference paper (under submission).