M.Sc. Thesis

by

Ertem Esiner

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 15, 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

Ertem Esiner

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. Attila Gürsoy

_____

Assoc. Prof. Serdar Taşıran

_____

Assoc. Prof. Emre Alper Yıldırım

Date: _____

To my beloved mother.

ABSTRACT

Cloud storage systems are becoming cheaper and more available. With the increase in popularity of the cloud storage systems both in industry and our personal lives people have started to care about the security of their data on the clouds. In this thesis, we develop and test a complete system for a server able to prove integrity of the client's data without her downloading the whole data, and still letting the client interact with her data in a read/write manner. This system is called Dynamic Provable Data Possession by Erway et al. .

We first show the FlexList: Flexible Length-Based Authenticated Skip List, a data structure optimized for secure cloud storage systems, and its differences from previous data structures. Then we demonstrate its utilization on the dynamic provable data possession system and we call the new scheme FlexDPDP. We further optimize the FlexDPDP scheme using parallelization techniques, and provide optimized algorithms to reduce the time complexity of the protocol.

We provide an analysis on all of our proposals at the end of each chapter. We also deployed the optimized FlexDPDP scheme on large-scale network test-bed PlanetLab, demonstrating that FlexDPDP performs comparably to the most efficient static storage scheme (PDP), while providing dynamic data support. Finally, we demonstrate the efficiency of our proposed optimizations on multi-client scenarios according to real workloads based on real version control system traces.

ÖZETÇE

Bulut depolama sistemleri gittikçe ucuzluyor ve yaygınlaşıyor. Bu sistemlere olan ilgi arttıkça günlük yaşamlarımızda da, endüstride de insanlar verilerinin güvenliğini daha çok önemsemeye başlıyorlar. Bu tezde biz, bir sunucunun, müşterinin verisinin istediği kısımlarını yazma okuma şeklinde değiştirmesine izin verirken aynı zamanda verinin tamamını indirmeden bütünlüğünü ispatlayabileceği, kullanılabilir tam bir sistem öneriyoruz. Bu sisteme, Erway et al. 'ın ismini koyduğu dinamik ispatlanabilir veri saklama deniliyor.

Önce bulut sistemleri için en verimli duruma getirilmiş FlexList veri yapısını (Esnek Uzunluk-tabanlı doğrulanabilir atlamalı liste) ve eski veri yapılarından farklarını anlatıyoruz. Devamında bu veri yapısını dinamik ispatlanabilir veri saklama sistemindeki kullanımını gösteriyor ve yeni oluşan plana (scheme) FlexDPDP diyoruz. FlexDPDP planını paralelleştirme yöntemleri kullanarak ve verimli algoritmalar sağlayarak daha da iyi kullanılabilecek bir hale getiriyoruz.

Her bölümün sonunda o bölümde tavsiye ettiklerimizin verimlilik incelemesini sunuyoruz. Ayrıca, iyileştirilmiş FlexDPDP planını geniş çaplı ağ deneme yatağı olan PlanetLab'da çalışır hale getirdik ve FlexDPDP'nin dinamik veri özelliğini sağlamasına rağmen, en verimli statik veri saklama sistemiyle (PDP) kıyaslanabilir olduğunu gösteriyoruz. Son olarak da kurduğumuz planı gerçeğe yakın ortamlar ve gerçek dosya sürümleme sistemlerinden alınmış iş yükleriyle test ederek geliştirilmiş algoritmalarımızın verimliliğini gösteriyoruz.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors Assoc. Prof. Öznur Özkasap and Assist. Prof. Alptekin Küpçü for the continuous support of my MSc. study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me, not only academically but, they were like a family to me.

I would like to thank Koçsistem for their faith in me and their financial support.

I am grateful that in the beginning of my MSc. studies, for 8 months, I studied and did research with Adilet Kachkeev, my ingenuous colleague with whom we have established the basis of our research. Including Chapter 4.4 we have worked together (some of his parts are excluded) and the first 5 chapters we submitted as a journal paper. The rest of this work, starting from Chapter 5, is also sumbitted to a conference.

I am grateful to Mohammad Etemad, Ozan Okumuşoğlu and everyone in crypto lab at Koç, without whom the days and nights I passed in office wouldn't have been as much fun or as pleasant. I also would like to thank Emrah Çem, and many others that have been helping me one way or another.

My sincere thanks goes to Suat Ertem who guided and encouraged me to find my way through life. I owe him perfect high school years, a gorgeous college and whatever good things I have become today.

Last but not least, I would like to thank the pearls of my life. It is not possible to express the value of the family. My mother, Hale Korman is basically the reason

why I am the person I am today. I am way too grateful, myself. And my father, Kemal Esiner, was always there when I need him and most thinks that I inherited his disciplined and being hardworking approach.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Data outsourcing has become quite popular in recent years both in industry (e.g., Dropbox, box.net, Google Drive, Amazon S3, iCloud, Skydrive) and academia [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, Zhang and Blanton, 2013b, Yuan and Yu, 2013]. The most important impediment in public adoption of cloud systems is due to the lack of some security guarantees in data storage services [Jensen et al., 2009, Furht and Escalante, 2010, Wooley, 2011]. In a cloud storage system, there are two main parties, a server and a client where the client transmits her files to the cloud storage server and the server stores the files on behalf of the client. The client outsources her data to the third party data storage provider (server), which is supposed to keep data intact and make it available to her. In this work, we address the integrity of the client's data stored on the cloud storage servers. A trustworthy brand is not sufficient for the client, since hardware/software failures or malicious third parties may also cause data loss or corruption [Cachin et al., 2009]. 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 model, the client can challenge the server on random blocks and verify the data integrity through a

proof sent by the server. Solutions for the static cases (i.e., logging, archival, static file sharing) such as Provable Data Possession [Ateniese et al., 2007] were proposed [Dodis et al., 2009, Juels and Kaliski., 2007, Shacham and Waters, 2008, Ateniese et al., 2007, Ateniese et al., 2009]. These proposals show poor performance for blockwise update operations (insertion, removal, modification).

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 $O(\log n)$ 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 and modifying 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. presents the idea where the client makes updates on a range of bytes instead of blocks (in Section 7.1 of [Erway et al., 2009]) by defining rank values of the bottom level nodes to the size of its corresponding data and states that queries and proofs proceed as before. We show that a naive implementation as mentioned in Section 7.1 of [Erway et al., 2009] leads to a security gap in the storage system. We show in detail that our FlexDPDP scheme is **provably secure**.

Our optimizations result in a dynamic cloud storage system whose efficiency is comparable to the best known static system PDP. Our system takes $\sim$15% more time than an optimized version of PDP on the client side when implemented on a real

network system. Our results confirm that FlexDPDP improves throughput 11 times at the server side in comparison to existing PDP approach, at the cost of increasing response time ∼35% at the client side.

1.1 Contributions

- (Chapter 3) 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).
- (Chapter 3) 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.
- (Chapter 3) 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.
- (Chapter 4) 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% and 40% in terms of proof time and size, respectively.
- (Chapter 5) We optimize the first preprocessing phase of the FlexDPDP provable cloud storage protocol by showing that the algorithm to build a FlexList in $O(n)$ time is well parallelizable even though FlexList is an authenticated data structure that generates dependencies over the file blocks. We propose a parallelization algorithm and our experimental results show a speed up of 6 and 7.7,

with 8 and 12 cores respectively.

- (Chapter 5) We provide a multiple update algorithm for FlexDPDP. Our experiments show 60% efficiency gain at the server side compared to updating blocks independently, when the updates are on consecutive data blocks.
- (Chapter 5) We provide an algorithm to verify update operations for FlexDPDP. Our new algorithm is applicable to not only modify, insert, and remove operations but also a mixed series of multiple update operations. The experimental results show an efficiency gain of nearly 90% in terms of verification time of consecutive updates.
- (Chapter 6) We deployed the FlexDPDP implementation on the network testbed PlanetLab and also tested its applicability on a real SVN deployment. The results show that our improved scheme is practically usable in a real life scenarios with only 35KB overhead (from the server to the client) per update of size 77KB on average. Our results demonstrate that FlexDPDP performs comparable to the most efficient static storage scheme (PDP), while providing dynamic data support.

1.2 Overview

In this thesis we propose, present and test a complete dynamic provable data possession scheme where the server is able to prove integrity of the client's data without her downloading the whole data, and still letting the client interact with her data. Details of the thesis are presented in respective chapters as described here:

Chapter 2 presents the literature review. We divide the literature into two parts. In the first part we discuss the cloud storage related work, then we discuss the related work about the underlying data structure and provide comparison with the alternatives.

Chapter 3 presents the underlying data structure used in our protocol. We first start with the definitions and tell why the FlexList is important. Then, we provide definitions and fundamental functions of the data structure, such as insert, remove, modify and build. At the end of the chapter, we provide analysis of the proposed algorithms.

Chapter 4 presents the scheme we propose and shows the algorithms employed in the scheme both for challenge-verify and verifiable update mechanisms. In this chapter, we also provide analysis for the functions provided and we compare our system with one of the most efficient static provable data possession scheme. Last, we prove our scheme secure under the same assumptions with the previous work.

Chapter 5 presents further optimizations on the flexDPDP scheme by using parallel techniques and providing more efficient algorithms. First, we show parallelization of the build algorithm of the FlexList. Second, we show how to handle multiple updates in one query. Third, we show the multiple verification mechanism (at the client side), without which multiple update operation can not be performed (at the server side).

Chapter 6 presents analysis on the proposed efficient algorithms and their implementation on the large-scale network test bed PlanetLab. In this chapter, we use real work loads under realistic conditions.

Chapter 2

RELATED WORK

2.1 Cloud Storage Related Work

2.1.1 Static 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 improving data integrity on the cloud, some protocols such as [Chockler et al., 2009, Hendricks et al., 2007, Cachin and Tessaro, 2006, Liskov and Rodrigues, 2006, Goodson et al., 2004, Malkhi and Reiter, 1998] provide Byzantium fault-tolerant storage services based on some server labor. There also exist protocols using quorum techniques, which do not consider the server-client system but works on local systems such as hard disk drives or local storage [Jayanti et al., 1998,

Gafni and Lamport, 2003, Abraham et al., 2006, Chockler and Malkhi, 2002]. A recent protocol using quorum techniques [Bessani et al., 2011] replicates the data on several storage providers to improve integrity of data stored on the cloud, yet the protocol only considers static data.

2.1.2 Dynamic Cloud Storage

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 preprocesses 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. Some distributed versions of the idea have been studied as well [Curtmola et al., 2008, Etemad and Küpçü, 2013]. There are other studies showing that a client's file is kept intact in the sense that client can retrieve (recover) it fully whenever she wishes [Juels and Kaliski., 2007, Shacham and Waters, 2008, Dodis et al., 2009, Cash et al., 2013, Bowers et al., 2009].

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, 2013a]. 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.

	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

2.2 Skip Lists and Other Data Structures

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.

Chapter 3

FLEXLIST: FLEXIBLE LENGTH-BASED AUTHENTICATED SKIP LIST

In this chapter, we provide the data structure FlexList to be used as the underlying data structure in the later chapters. To the best of our knowledge, the data structure we provide in this chapter is the first optimized implementation of its kind. It has no unnecessary links and nodes (we clearly define necessary and unnecessary) and it can be constructed from scratch in $O(n)$. The implementation also covers Rank-based skip list, the data structure proposed in [Erway et al., 2009]. We first start with the primitive version of a skip list and show its evolution to our FlexList.

3.1 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

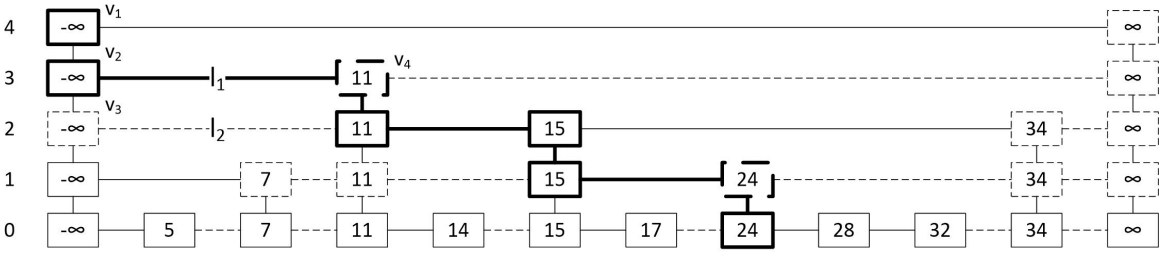

Figure 3.1: Regular skip list with search path of node with key 24 highlighted. Numbers on the left represent levels. Numbers inside nodes are key values. Dashed lines indicate unnecessary links and nodes.

the link to v_2, since v_1's $after$ link leads it to a node which has a greater key value 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).

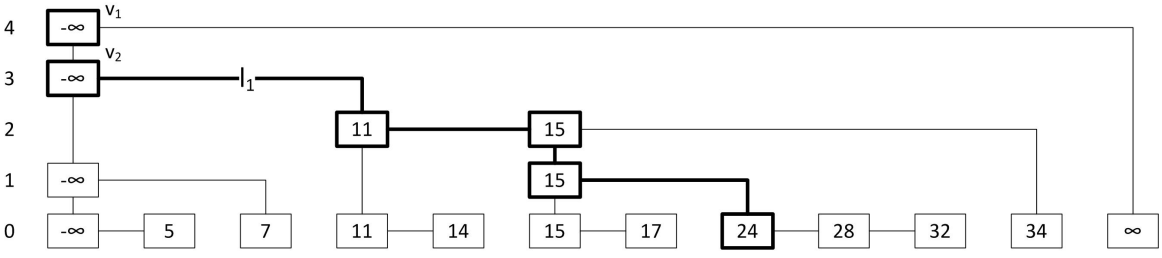

Figure 3.2: Skip list of Figure 3.1 without unnecessary links and nodes.

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 at least one 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 at least one 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 *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

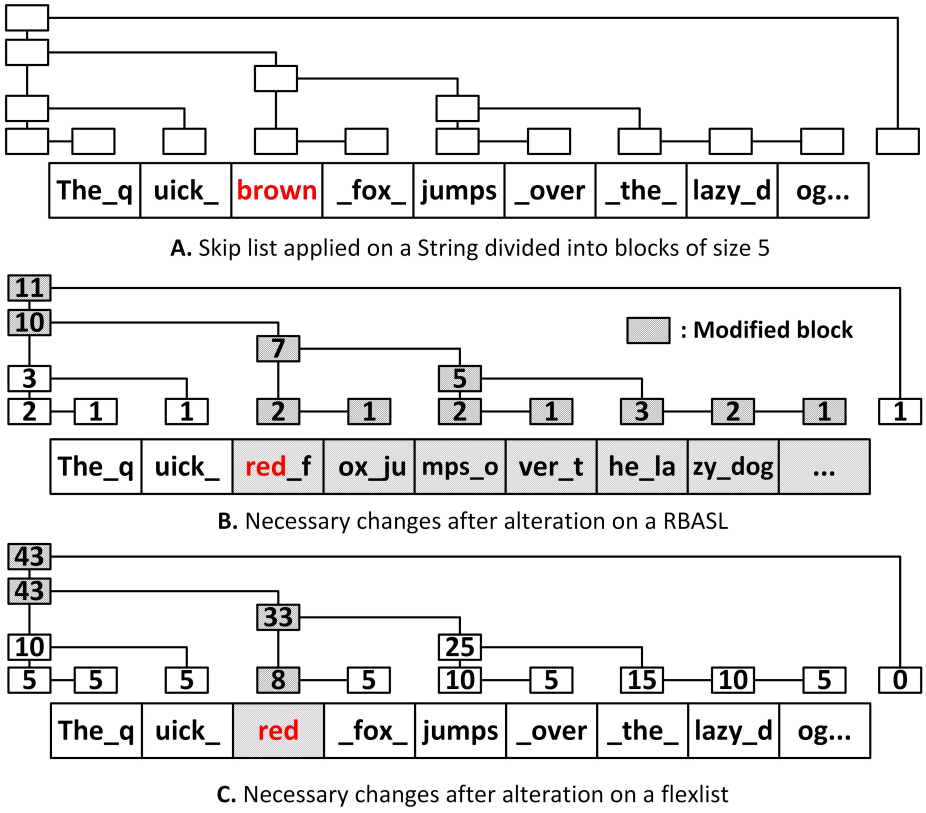

Figure 3.3: Skip list alterations depending on an update request.

(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

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. We discuss the insecurity of an implementation that does not include the length information in the hash function calculation in Section 4.6. 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

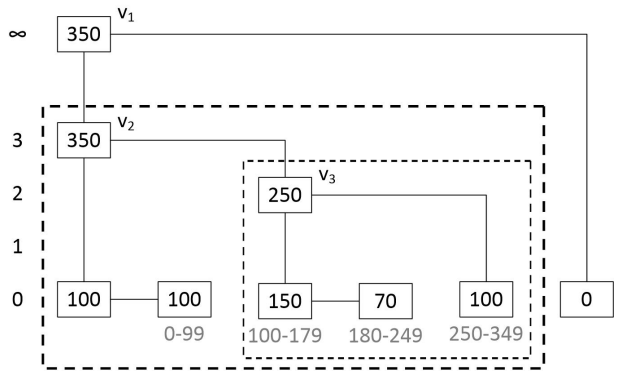

Figure 3.4: A FlexList example with 2 sub skip lists indicated.

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

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.

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.

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
```

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
```

Algorithm 3.2.4: deleteNode Algorithm

Input: pn, cn
Output: pn, cn

```
1  pn.after = cn.after
```

deleteNode (Algorithm 3.2.4), 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 Methods of FlexList

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.

A search path, which is the basic idea of a proof path, is visible in the stack in

the basic algorithms.

search (Algorithm 3.2.5) 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.

Algorithm 3.2.5: 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 3.2.5, 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. 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 3.2.6: 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.
```

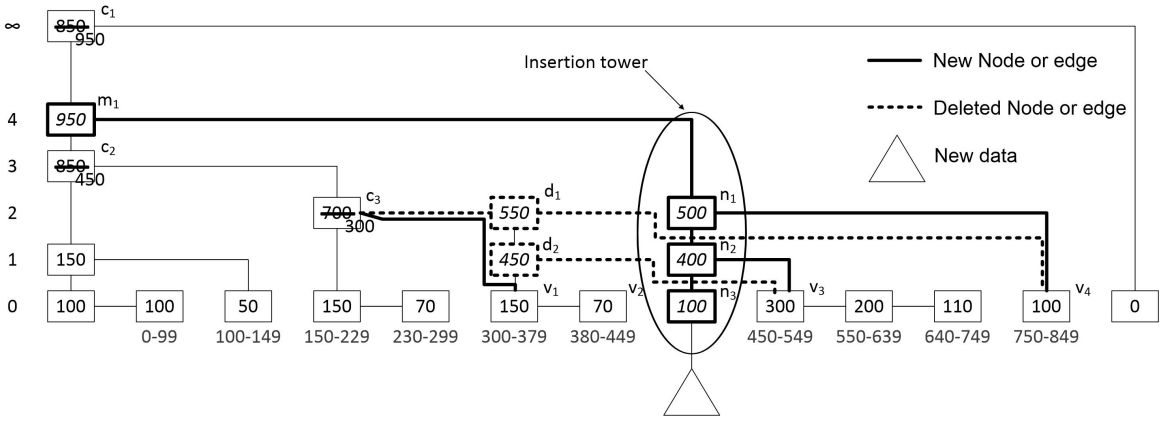

Figure 3.5: Insert at index 450, level 4 (FlexList).

insert (Algorithm 3.2.6) 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 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 3.2.7) 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.

[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).

Algorithm 3.2.7: 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.
```

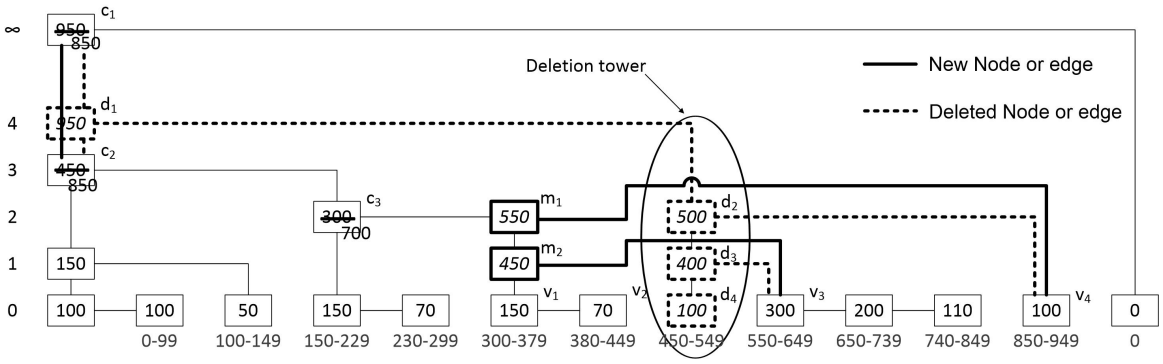

Figure 3.6: Remove block at index 450(FlexList).

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 Build from Scratch Algorithm

Algorithm 3.2.8: 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 takes 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 3.2.8) 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.

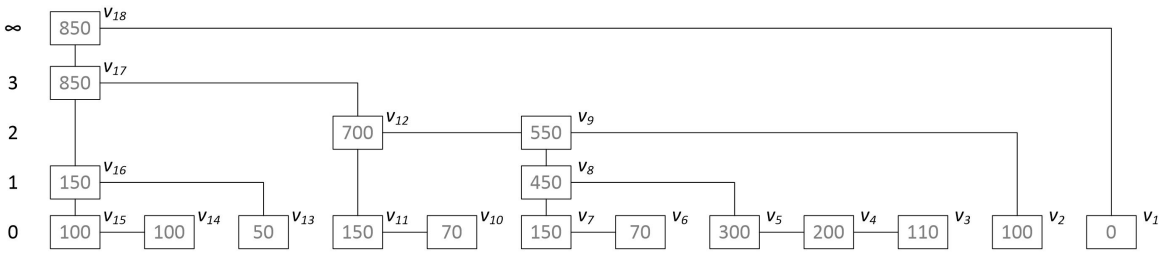

Figure 3.7: buildFlexList example.

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.

3.3 Performance Analysis

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 is stated otherwise. The size of a FlexList is suitable to keep a lot of FlexLists in RAM.

Core FlexList Algorithms Performance:

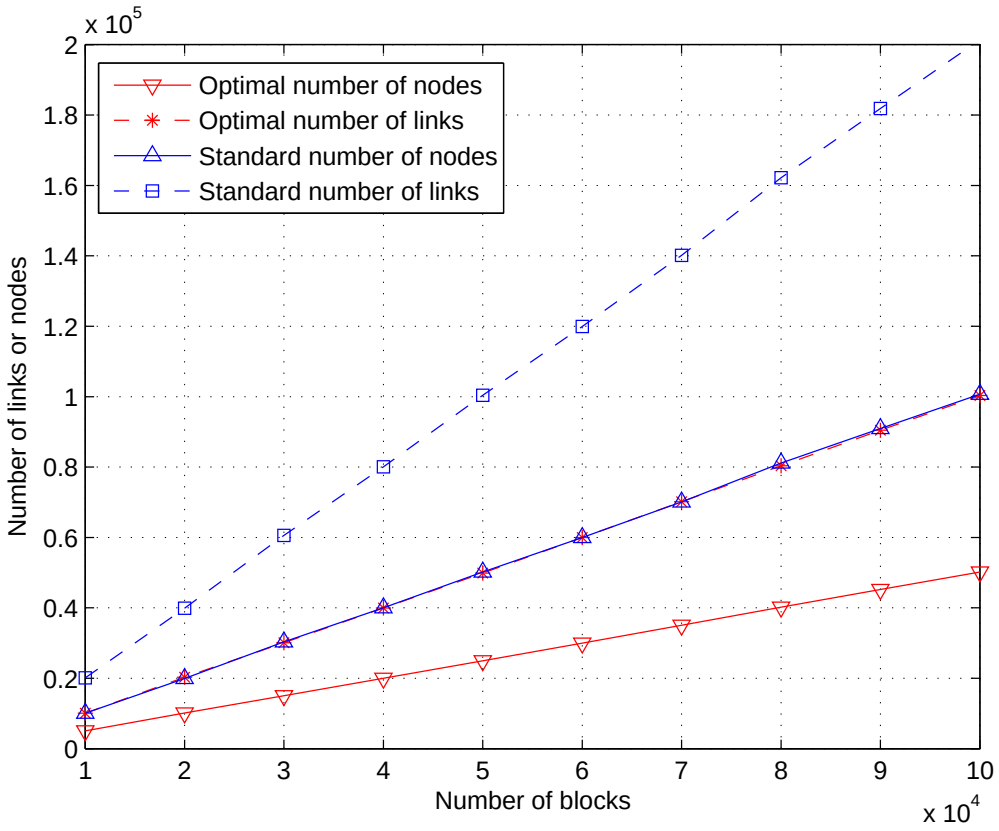

Figure 3.8: The number of nodes and links used on top of leaf level nodes, before and after optimization.

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.

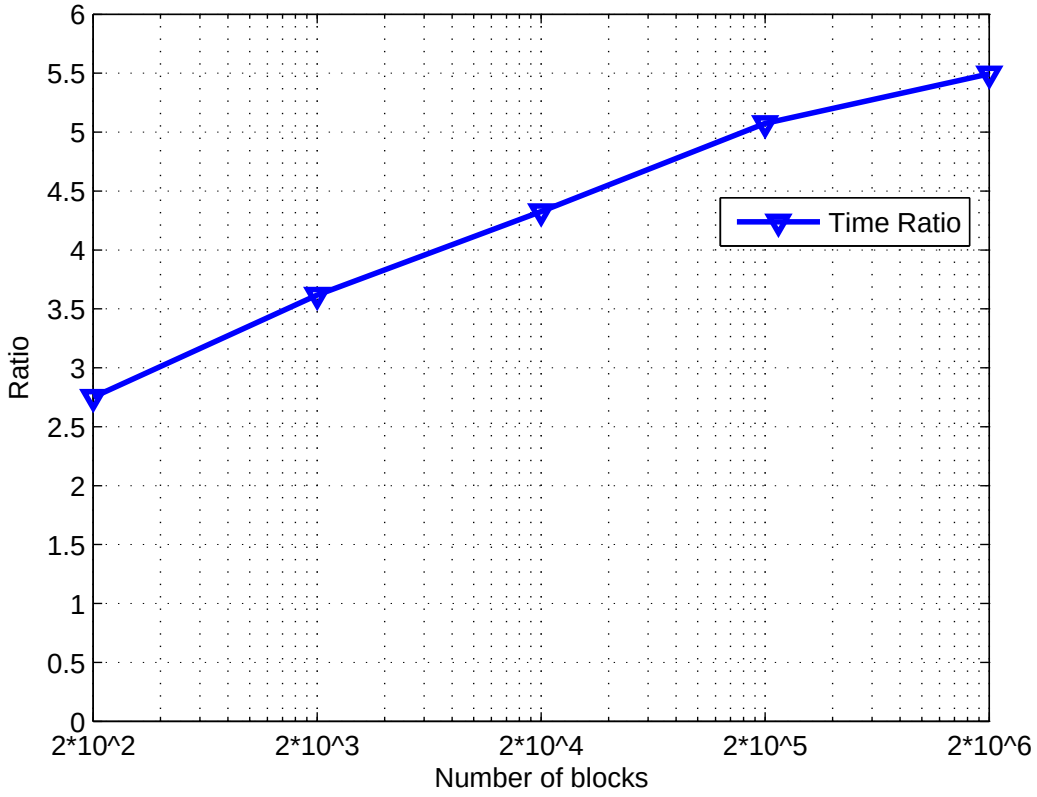

Figure 3.9: Time ratio on buildFlexList algorithm against insertions.

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 ratio 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 building FlexList using insertion method by the time needed by the *buildFlexList* algorithm. In our time ratio experiment, 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**.

Chapter 4

FLEXDPDP: FLEXIBLE DYNAMIC PROVABLE DATA POSSESSION

In this chapter, we describe the application of our FlexList to integrity checking in secure cloud storage systems according to the DPDP model [Erway et al., 2009]. 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.

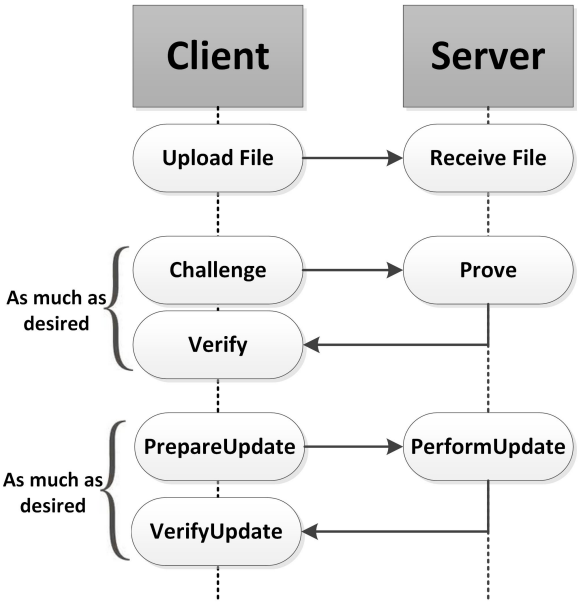

Figure 4.1: Client Server interactions in FlexDPDP.

We construct the above model with FlexList as the authenticated data structure. We provide new capabilities and efficiency gains as discussed in Section 3.2 and call the resulting scheme **FlexDPDP**. In this section, we describe our corresponding algorithms for each step in the DPDP model.

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.

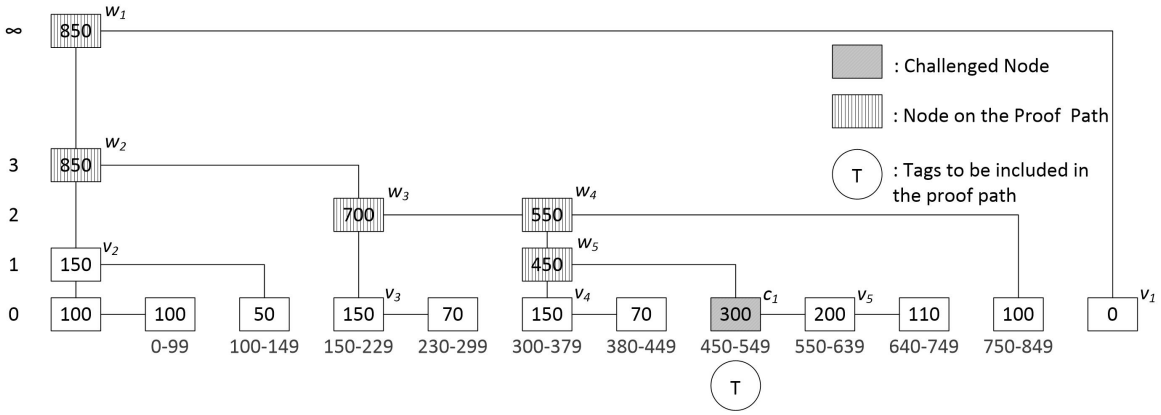

Figure 4.2: Proof path for challenged index 450 in a FlexList.

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 Handling 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.

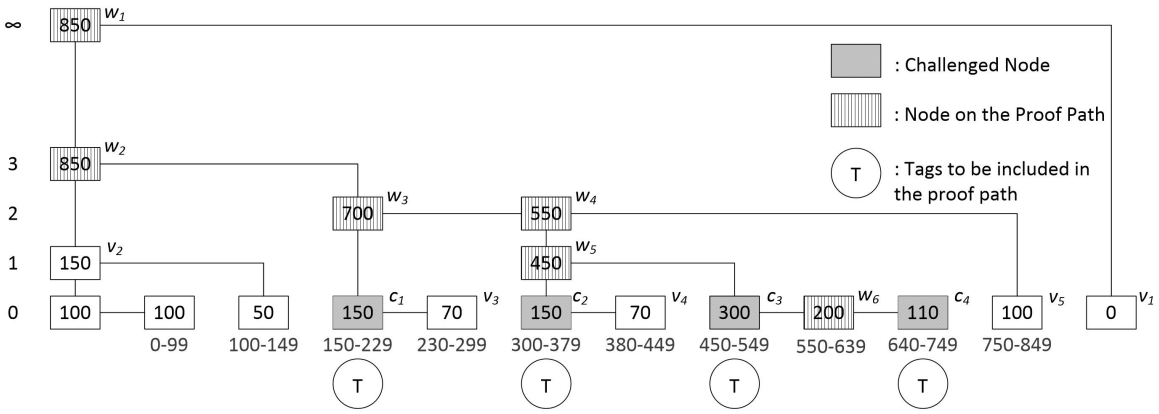

Figure 4.3: Multiple blocks are challenged in a FlexList.

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 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 5.3. 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

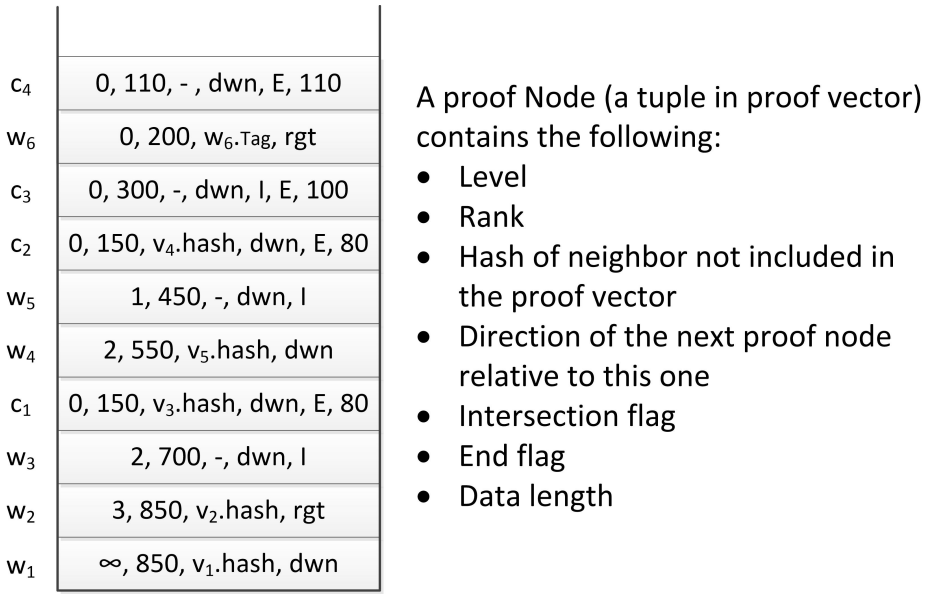

Figure 4.4: Proof vector for Figure 4.3 example.

4.2.2 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 5.3.

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 5.3) 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).

Algorithm 4.2.2: verifyMultiProof Algorithm

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

Let $P = (A_0, \ldots, A_k)$, where $A_j = (level_j,\ r_j,\ \text{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ 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 $= 0$; $\text{hash}_{prev} = 0$; $startTemp = 0$; $\sqcup_h$ 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 $= hash(level_j, r_j, tag_t, \text{hash}_{prev}, length_j)$; 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 $= hash(level_j, r_j, tag_t, \text{hash}_j, length_j)$; 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 $= hash(level_j, r_j, \text{hash}_{prev}, \sqcup_h.pop())$
16 **else if** $rgtOrDwn_j = rgt$ *or* $level_j = 0$ **then**
17 hash $= hash(level_j, r_j, \text{hash}_j, \text{hash}_{prev})$
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 $= hash(level_j, r_j, \text{hash}_{prev}, \text{hash}_j)$
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 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.

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 Performing an Update

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 Verifying an Update

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*, $MetaData_{bySever}$
Output: accept or reject, $MetaData'$

```
   Let C= (i_0) where i_0 is the index of the left neighbor; P
   = (A_0, ..., A_k), where A_j = ( level_j, r_j, hash_j, rgtOrDwn_j, isInter_j,
   isEnd_j, length_j) for j = 0, ..., 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≠ 0 then
7          hash = hash(0, A_i.length, tag_0, 0);
8      decrement(i)
9      while A_i.level ≠ 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_bySever = 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

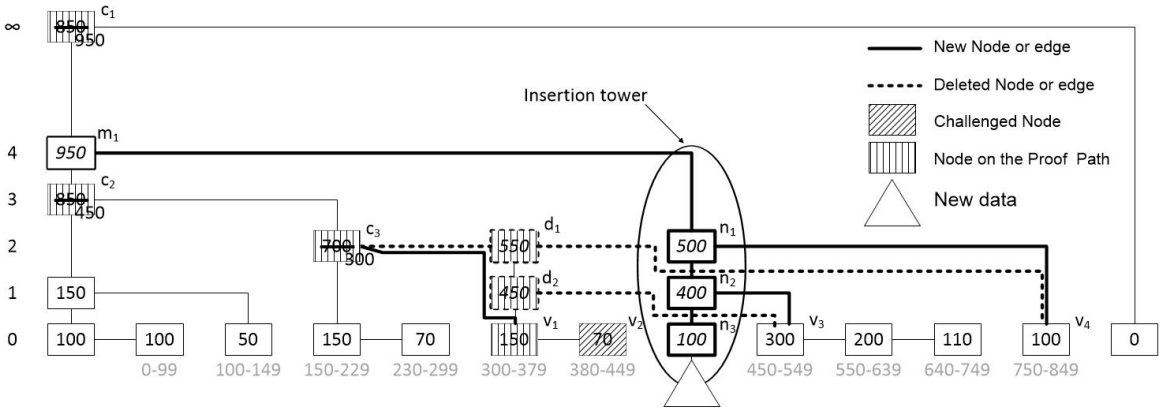

Figure 4.5: Verifiable insert example.

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.

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.

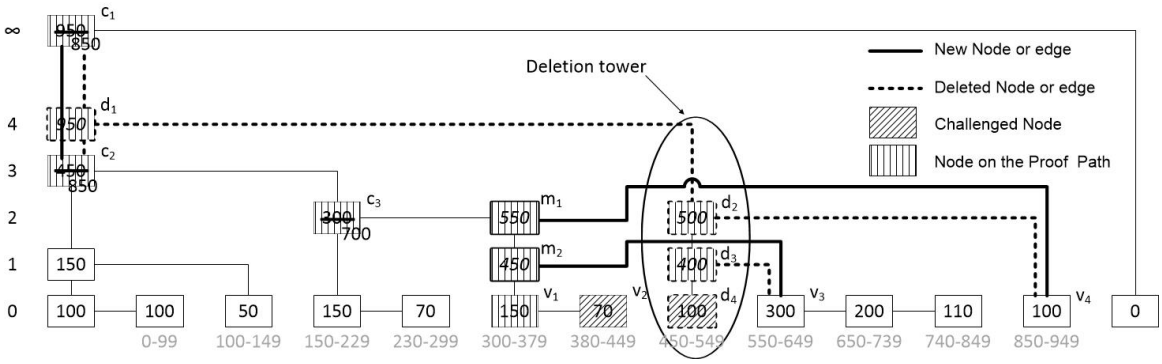

Figure 4.6: Verifiable remove example.

Algorithm 4.3.3: verifyRemove Algorithm

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

`Let C=` (i_0, i_1) `where` i_0, i_1 `are the index of the left neighbor and the removed index respectively;` $P = (A_0, \ldots, A_k)$, `where` $A_j = (level_j,\ r_j,$ `hash`$_j$, $rgtOrDwn_j$, $isInter_j$, $isEnd_j$, $length_j)$ `for` $j = 0, \ldots, k$; $T = (tag_0, tag_1)$ `where` tag_1 `is tag value for deleted node and` tag_0 `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** *!A_i.isEnd* **do**
6 decrement(i)
7 rank = A_{dn}.rank; hash = *hash*(0, rank, tag_0, A_{dn}.hash, A_{dn}.length)
8 decrement(dn)
9 **if** *!A_{dn}.isEnd or !A_i.isInter* **then**
10 decrement(i)
11 **while** *!A_{dn}.isEnd or !A_i.isInter* **do**
12 **if** *A_i.level < A_{dn}.level or A_{dn}.isEnd* **then**
13 rank += A_i.rank - A_{i+1}.rank
`// A`$_i$`.length is added to hash calculation if A`$_i$`.level =` 0
14 hash = *hash*(A_i.level, rank, A_i.hash, hash)
15 decrement(i)
16 **else**
17 rank += A_{dn}.rank - A_{dn+1}.rank
18 hash = *hash*(A_{dn}.level, rank, hash, A_{dn}.hash)
19 decrement(dn)
20 decrement(i)
21 $MetaData_{new}$ = calculateRemainingHashes(i, hash, A_{last}.length, P)
22 **if** $MetaData_{bySever} = MetaData_{new}$ **then**
23 MetaData = $MetaData_{new}$
24 return accept
25 return reject

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

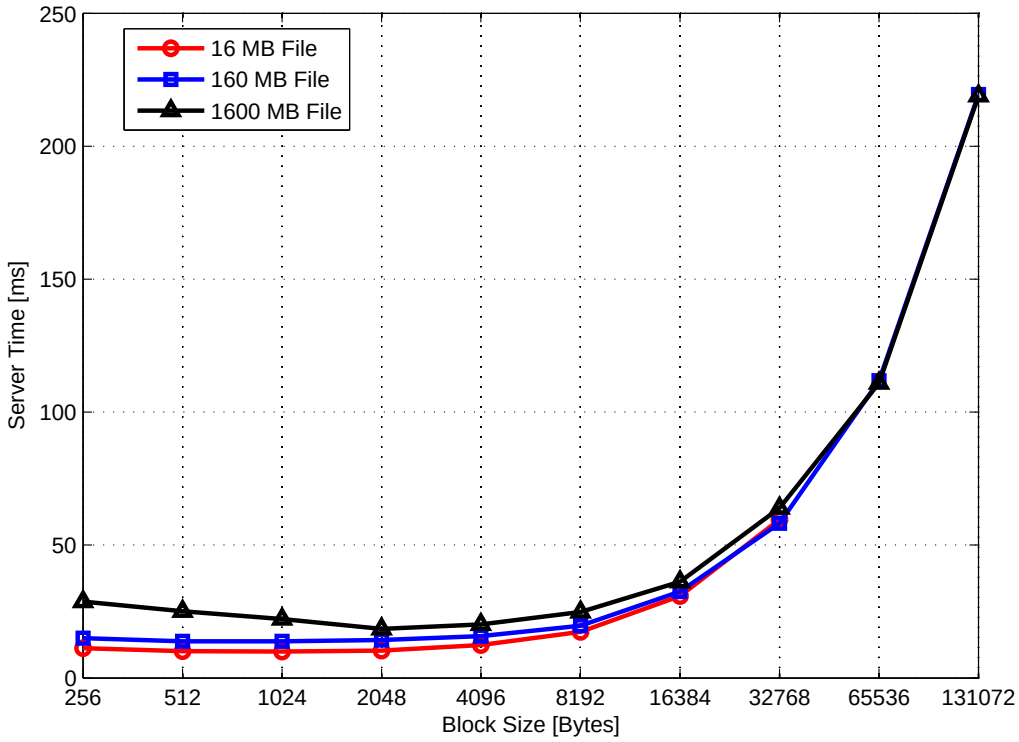

Figure 4.7: Server time for 460 random challenges as a function of block size for various file sizes.

sizes (since there will be fewer blocks), the creation of a FlexList happens only once. On the other hand, the proof generation algorithm runs periodically depending on the client, therefore we chose to optimize 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

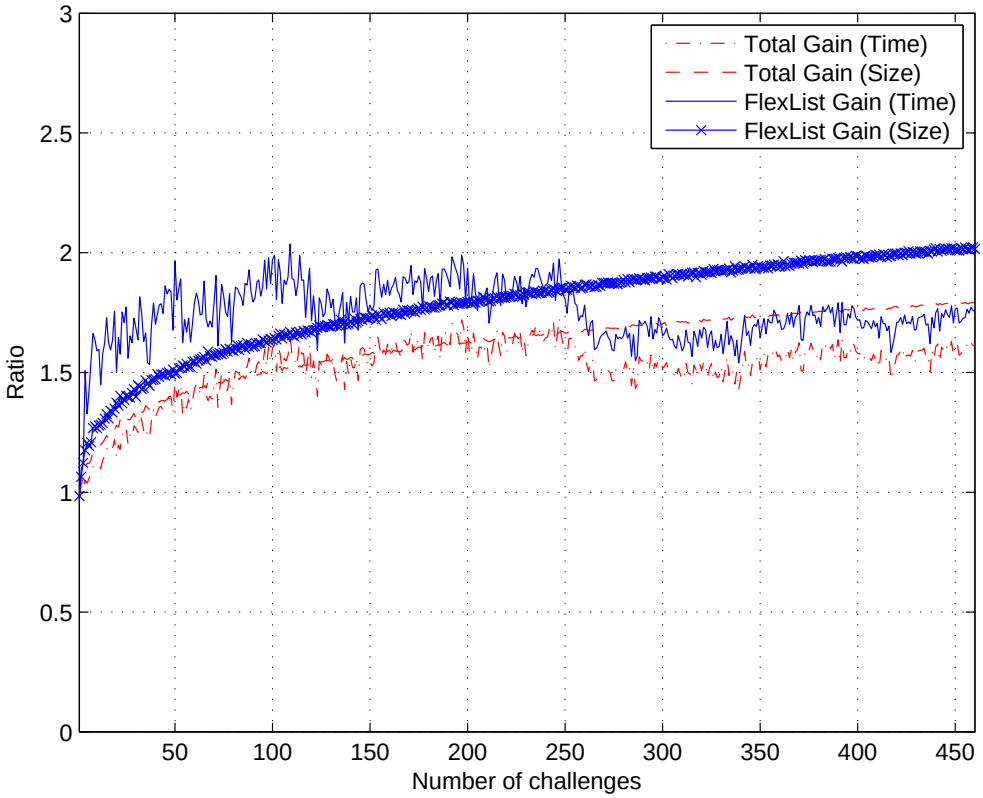

Figure 4.8: Performance gain graph ([460 single proof / 1 multi proof] for 460 challenges).

proofs in terms of the **FlexList proof** size and computation, as a function of the 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 communica-**

tion, 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.

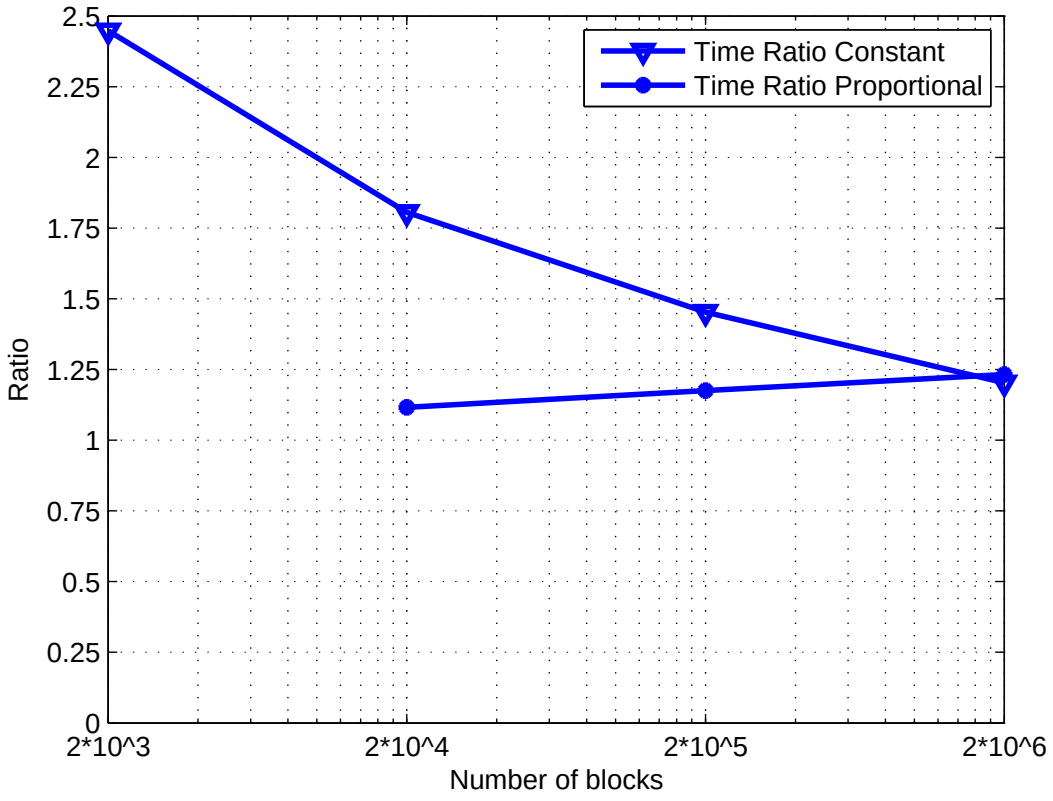

Figure 4.9: Time ratio on *genMultiProof* algorithm.

Furthermore, we tested the performance of *genMultiProof* algorithm. 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.

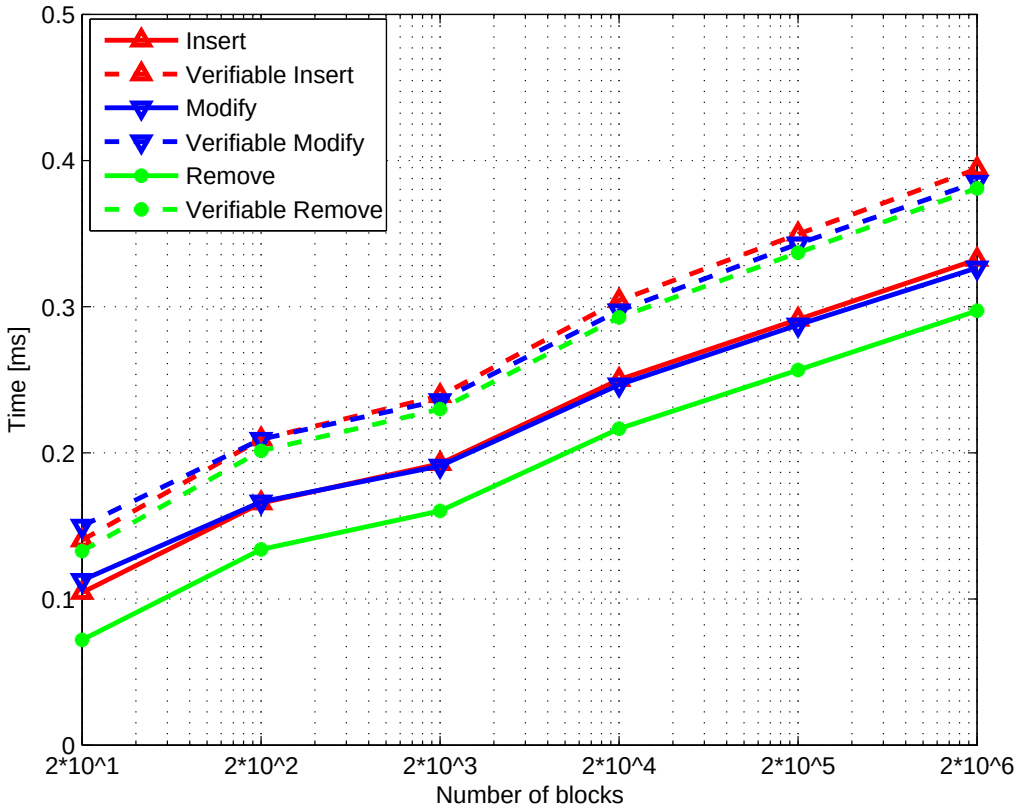

Figure 4.10: Performance evaluation of FlexList methods and their verifiable versions.

4.5 Comparison with Static Cloud Storage on the PlanetLab

We compare static PDP with FlexDPDP, which is a dynamic system. The server in PDP computes the sum of the challenged blocks and the multiplication and exponentiation of their tags. FlexDPDP server only computes the sum of the blocks and FlexList proof, but not the multiplication and exponentiation of their tags, which are expensive cryptographic computations. In such a scenario the FlexDPDP server outperforms such a naive PDP server, since the multiplication of tags in PDP takes much longer than the FlexList proof generation in FlexDPDP. This result is in con-

trast to the fact that PDP proofs take $O(1)$ time and space whereas FlexDPDP proofs require $O(\log n)$ time and space, due to a huge difference in the constants in the Big-Oh notation.

We note that it is possible for PDP to be implemented by the server sending the tags to the client and the client computing the multiplication and exponentiation of the tags. If this is done in a PDP implementation, even though the proof size grows, the PDP server can respond to challenges faster than FlexDPDP. Therefore, we realize that where to handle the multiplication and exponentiation of tags is an implementation decision for PDP.

	PDP	PDP*	FlexDPDP
Local Server Computation	413.19	12.97	38.60
Close Client Total	466.82	557.49	649.11
Mid-range Client Total	496.856	714.47	874.63
Distant Client Total	551.376	986.98	1023.25

Table 4.2: Time spent for a challenge of size 460, in milliseconds. PDP* is the modified PDP scheme, where we send all challenged tag values to the client instead of multiplying them.

We deployed FlexDPDP, together with original and modified PDP versions, on the world-wide network test bed, PlanetLab. On PlanetLab, a node has minimum requirements of having 6x Intel Xeon E5 cores @ 2.2GHz processor, 24 GB of RAM, and 2TB shared hard disk space. The nodes are also required to have minimum of 400kbps of bi-directional bandwidth to the Internet [PlanetLab, 2013] As a central point in Europe, we chose a node in Berlin, Germany (planetlab01.tkn.tu-berlin.de) as the server. We measured the whole time spent for one challenge at both the client and the server side (Table 4.2). We moved our client location and tested serving a close range client in Munich, Germany (1.lkn.ei.tum.de), a mid-range client in Koszalin, Poland (ple2.tu.koszalin.pl), and a distant client in Lisbon, Portugal (planetlab1.di.fct.unl.pt). We used a single core at each side. The protocols are run

on a 1GB file, which is divided into blocks of 2KB, having 500000 nodes.

Inducting from Table 4.2, we conclude that using 6 cores (usual core count in PlanetLab nodes), a **PDP server can answer 14.5 queries per second** whereas a server using **PDP* and FlexDPDP can serve 462 queries and 155.5 queries per second** respectively. We discern that, for the server, tag multiplication is the most time consuming task in each challenge. It is clear that to increase the server throughput, tag multiplication should be delegated to the client. This delegation increases the total time spent by the client a bit more than it saves from the server, since the tags should be sent over the network. However, the outcome is the dramatic increase in the server throughput. Note that, when one considers the total time a client spends for sending a challenge, obtaining the proof, and verifying it, the overhead of being dynamic (FlexDPDP vs. PDP*) is around 40 to 90 ms, which is a barely-visible difference for a real-life application (especially considering that the whole process takes on the order of a second).

4.6 Security Analysis

Note that within a proof vector, all nodes which are marked with the end flag "E" contain the length of their associated data. These values are used to check if the proof in the process of verification is indeed the proof of the block corresponding to the challenged index. A careless implementation may not consider the authentication of the length values. To show the consequence of not authenticating the length values, we will use Figure 4.3 and Figure 5.3 as an example.

The scenario starts with the client challenging the server on the indices {170, 400, 500, 690} that correspond to nodes c_1, v_4, c_3, and c_4 respectively. The server finds out that he does not possess v_4 anymore, and therefore, instead of that node, he will try to deceive the client by sending a proof for c_2. The proof vector will just be the same

as the proof vector illustrated in Figure 5.3 with a slight change done to deceive the client. The change is done to the fourth entry from the top (the one corresponding to c_2): Instead of the original length 80, the server puts 105 as the length of c_2. The verification algorithm (without authenticated length values) at the client side will accept this fake proof as follows:

- The block sum value and the tags get verified since both are prepared using genuine tags and blocks of the actual nodes. The client cannot realize that the data of c_2 counted in the block sum is not 105 bytes, but 80 bytes instead. This is because the largest challenged data (the data of c_4 of length 110 in our example) hides the length of the data of c_2.
- Since the proof vector contains genuine nodes (though not necessarily all the challenged ones), when the client uses *verifyMultiProof* algorithm on the proof vector from Figure 5.3, the check on line 22 (Algorithm 4.2.2), "$hash \neq$ Meta-Data" will be passed.
- The client also checks that the proven nodes are the challenged ones by comparing the challenge indices with the reconstructed indices by "$\forall$a, $0 \leq a \leq n$, $0 \leq C_a - V_a < endnodes_{n-a}.length$" (Algorithm 4.2.2 on line 22). This check will also be passed because:
  - c_1 is claimed to start at index 150 and contain 80 bytes, and hence includes the challenged index 170 (verified as $0 < 170 - 150 < 80$).
  - c_2 is claimed to start at index 300 and contain **105** bytes, and hence includes the challenged index 400 (verified as $0 < 400 - 300 < 105$).
  - c_3 is claimed to start at index 450 and contain 100 bytes, and hence includes the challenged index 500 (verified as $0 < 500 - 450 < 100$).
  - c_4 is claimed to start at index 640 and contain 110 bytes, and hence includes the challenged index 690 (verified as $0 < 690 - 640 < 110$).

There are two possible solutions. We may include either the authenticated rank values of the right neighbors of the end nodes to the proofs, or use the length of the associated data in the hash calculation of the leaf nodes. We choose the second solution, which is authenticating the length values, since adding the neighbor node to the proof vector also adds a tag and a hash value, for each challenged node, to the communication cost.

Lemma 1. *If there exists a collision resistant hash function family, FlexList is an authenticated dictionary.*

Proof. The only difference between FlexList and RBASL is the calculation of the rank values at the leaf levels. All rank values, which are used in the calculation of the start indices of the challenged nodes, are used in hash calculations as well. Therefore, both *length* and *rank* values contribute to the calculation of the hash value of the root. To deceive the client, the adversary should fake the rank or length value of at least one of the proof nodes. By Theorem 1 of [Papamanthou and Tamassia, 2007], if the adversary sends a verifying proof vector for any node other than the challenged ones, we can break the collision resistance of the hash function, using a simple reduction. Therefore, we conclude that our FlexList protects the integrity of the tags and data lengths associated with the leaf-level nodes. □

Remember that the tags verification protects the integrity of the data itself, based on the factoring assumption, as shown by DPDP [Erway et al., 2009]. Combining this with Lemma 1 concludes security of FlexDPDP.

Theorem 1. *If the factoring problem is hard and a collision resistant hash function family exists, then FlexDPDP is secure.*

Proof. Consider the proof by Erway et al. for Theorem 2 of [Erway et al., 2009]. Replacing Lemma 2 of [Erway et al., 2009] in that proof with our Lemma 1 yields an identical challenger, and the exact proof shows the validity of our theorem. □

Chapter 5

OPTIMIZED FLEXDPDP: A PRACTICAL SOLUTION FOR DYNAMIC PROVABLE DATA POSSESSION

In this chapter, we provide further optimizations on the FlexDPDP. We use parallelization techniques to make the buildFlexList function faster, thus the preprocess operation can be performed faster. We provide optimized algorithms to perform update operations faster, thus we reduce the server load. We provide optimized algorithms to perform verification faster, thus making the client faster.

First, we start with some examples and definitions to be used throughout this chapter.

Figure 5.1 is the base FlexList we use in this chapter.

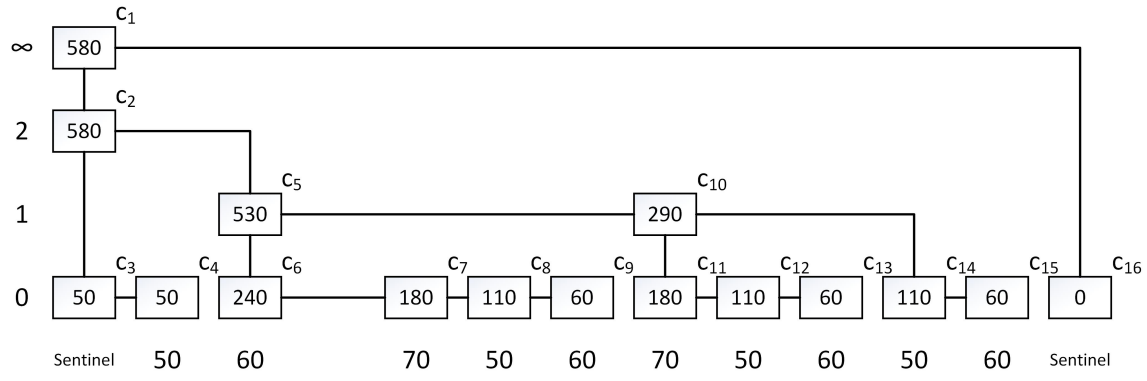

Figure 5.1: A FlexList example.

Remember that Insert/Remove operations perform add/remove of a leaf node by keeping the necessary non-leaf nodes and removing the unnecessary ones, thus

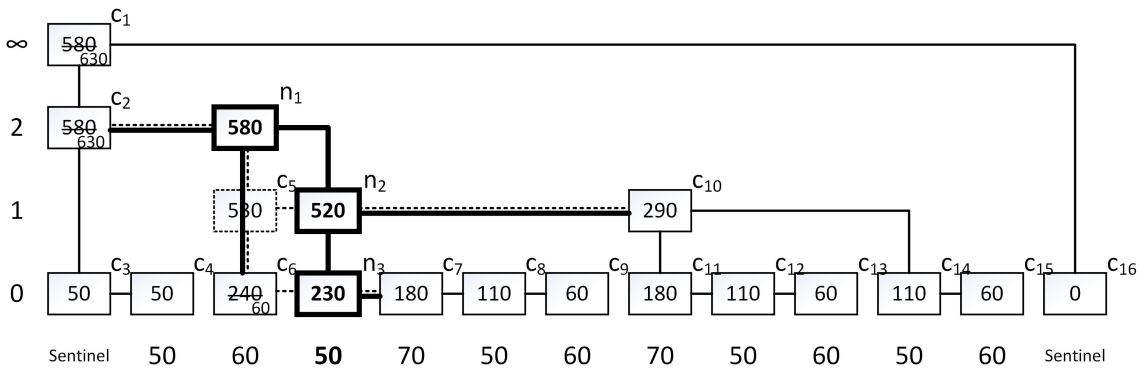

Figure 5.2: Insert and remove examples on FlexList.

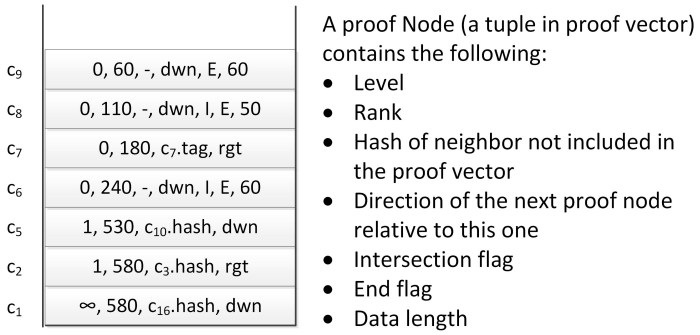

Figure 5.3: An output of a multiProof algorithm.

preserving the optimality of the structure. Figure 5.2 illustrates an example of both **insert and remove** operations. This example is given for better understanding of the verify algorithm provided later in this section.

First, we insert a data of length 50 to index 110 at level 2. Dashed lines show the nodes and links which are removed, and strong lines show the newly added ones. The old rank values are marked and new values written below them. For the removal of the node at index 110, read the figure in the reverse order, where dashed nodes and lines are newly added ones and strong nodes and lines are to be removed, and the initial rank values are valid again.

For proof of possession, **genMultiProof** collects all necessary values through search paths of the challenged nodes without any repetition. A multi proof is a response to a challenge of multiple nodes. For instance, on Figure 5.1 a challenge to indices 50, 180, 230 is replied by a proof vector as in Figure 5.3 and a vector of tags of the challenged nodes and the block sum of the corresponding blocks. This proof vector is used to verify the integrity of these specific blocks. We use, in Section 5.1.3, this proof vector to verify the multiple updates on the server.

Update information consists of the index of the update, the new data and the corresponding tag.

5.1 Optimizations

In this section, we describe our optimizations on FlexDPDP and FlexList for achieving an efficient and secure cloud storage system. We then demonstrate the efficiency of our optimizations in the Analysis section.

First, we observe that a major time consuming operation in the FlexDPDP scheme is the preprocess operation, where a *build FlexList* function is employed. Previous $O(n)$ time algorithm is an asymptotic improvement, but in terms of actual running times, it is still noticeably slow to build a large FlexList (e.g. half a minute for a file of size 1GB with 500000 blocks). A parallel algorithm can run as fast as its longest chain of dependent calculations, and in the FlexList structure each node depends on its children for the hash value, yet we show that building a FlexList is surprisingly well parallelizable.

Second, we observe that performing or verifying FlexDPDP updates in batches yield great performance improvements, and also match with real world usage of such a system. The hash calculations of a FlexList take the most of the time spent for an update, and performing them in batches may save many unnecessary calculations.

Therefore, in this section, we provide a **parallel algorithm for building FlexList**, a **multiple update algorithm** for the server to perform updates faster, and a **multiple verification algorithm** for the client to verify the update proofs sent by the server. The notations used in our algorithms is presented in Table 5.1 as a reminder. They are the same with the notations used so far.

Symbol	**Description**
cn / nn	current node / new node
$after$ / $below$	node reached by following the after link / by following the below link
i / $first$ / $last$	index / C's current index / C's end index
rs	rank state, is the state to indicate the bytes that can not be reached from that point
$state$	state contains a node, rank state, and last index and these values are used to set the cn to the point where the algorithm will continue
C	contains the indices that are challenged (ascending order)
P / T / M	proof vector / tag vector / block sum
$\sqcup_s$	intersection stack, stores states at intersections
$\sqcup_l$	stores nodes for which a hash calculation is to be done
Method	**Description**
canGoBelow	returns true if the searched index can be reached by following the below link
isIntersection	returns true if the current node is the lowest common ancestor(lca) of indices i and $last$ of vector C. This algorithm also decrements $last$ until it finds a C_{last} for which the current node is lca or until confirms that the current node is not a lca of two indices, where one is i
generateIndices	depending on the update information adds the indices of the nodes which are required to an array and return the array. If the update index is $idx \Rightarrow$ for insert and modify we add idx, for a remove we add idx and idx - *1*

Table 5.1: Symbols and helper methods used in our algorithms.

5.1.1 Parallel Build FlexList

We propose a parallel algorithm to generate FlexList over the file blocks, resulting in the same FlexList as a sequentially generated one. The algorithm has three steps. First, it divides the task into the sub-tasks. It generates small skip lists using the buildFlexList function. Second, it connects one root after another to discard right

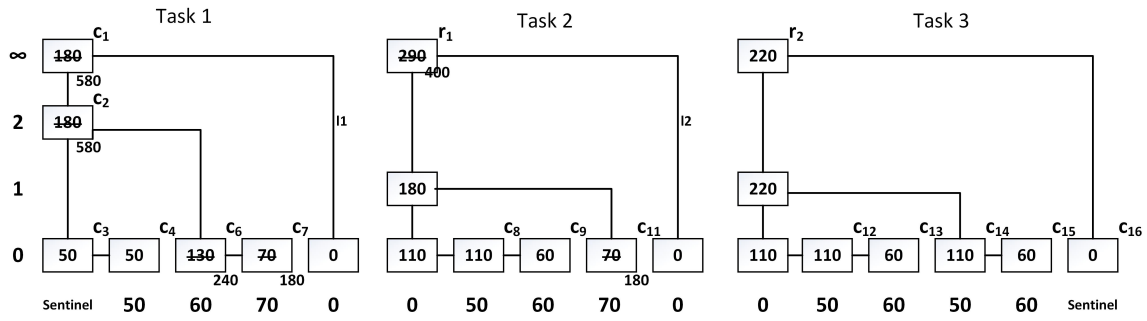

Figure 5.4: A build skip list distributed to 3 cores.

sentinel nodes of the FlexLists, but the rightmost one. Then, only left sentinel nodes remain extra in the list. Third, it removes the left sentinel nodes using basic remove function of the FlexList. As a result, all the nodes of the small FlexLists are connected to their level on the FlexList. Observe the node c_{10} while the remove operation in Figure 5.2 to understand the unifying property of remove operation.

Figure 5.4 shows the parallel construction of the FlexList (in Figure 5.1) on three cores. We first determine three tasks distributed to three threads and generate FlexLists. To unify the parts of the FlexList, we first connect all roots together with links (c_1 to r_1 and r_1 to r_2 in our example, thus eliminate l_1 and l_2) and calculate new rank values of the roots (r_1 and c_1). Then, we use basic remove function to remove left sentinels, which remain in between each part. We use the remove function (to indices 360 and 180: 360 = c_1.rank - r_2.rank and 180 = c_1.rank - r_1.rank). Remove operation generates c_5 and c_{10} of Figure 5.1 and connects the remaining nodes to them, and rank values of c_2, c_6, c_7, c_{11} get recalculated after the removal of sentinel nodes. After the unify operation, we obtain the same FlexList of Figure 5.1 generated efficiently in a parallel manner.

5.1.2 Handling Multiple Updates at Once

We investigated the verifiable updates and inferred that the majority of the time spent is for the hash calculations in each update. We discuss this in detail in Analysis Chapter. When a client alters her data and commits it to the server, she generates a vector of updates (U) out of a diff algorithm. An update information u, in U, includes an index i, and (if insert or modify) a block and a tag value. Furthermore, the updates on a FlexList consist of a series of *modify* operations followed by either *insert* or *remove* operations, all to adjacent nodes. This nature of the update operations makes single updates inefficient since they keep calculating the hash values of the same nodes over and over again. To overcome this problem, we propose dividing the task into two: doing a series of **updates without the hash calculations**, and then **calculate** all **affected nodes' hash values** at once, where affected means that at least a value of the hash calculation of that node has changed. **multiUpdate** (Algorithm 5.1.1) gets a FlexList and vector of updates U, and produces proof vector P, tag vector T, block sum M, and new hash value of the root after the updates newRootHash.

Algorithm 5.1.1: multiUpdate Algorithm

Input: FlexList, U
Output: P, T, M, newRootHash

```
    Let U= (u_0, ..., u_k) where u_j is the j^th update information
1   C = generateIndices(U) //According to the nature of the update for each
    u ∈ U , we add an index to the vector (u_j.i for insert and modify,
    u_j.i and u_j.i-1 for remove as it is for a single update proof)
2   P, T, M= genMultiProof(C) //Generates the multiProof using the
    FlexList
3   for i = 0 to k do
4       apply u_i to FlexList without any hash calculations
5   update C to all affected nodes using U
6   calculateMultiHash(C) // Calculates hash values of the changed nodes
7   rootHash = FlexList.root.hash
```

hashMulti (Algorithm 5.1.2), employed in *calculateMultiHash* algorithm, collects nodes on a search path of a searched node. In the meantime, it is collecting the intersection points (which is the lowest common ancestor (lca) of the node the col-

lecting is done for and the next node of which the hash calculation is needed). The repetitive calls from *calculateMultiHash* algorithm for each searched node collects all nodes which may need a hash recalculation. Note that each time, a new call starts from the last intersecting (lca) node.

calculateMultiHash (Algorithm 5.1.3) first goes through all changed nodes and collects their pointers, then calculates all their hash values from the largest index value to the smallest, until the root. This order of hash calculation respects all hash dependencies.

We illustrate handling multiple updates with an example. Consider a *multiUpdate* called on the FlexList of Figure 5.1 and a consecutive modify and insert happen to indices 50 and 110 respectively (insert level is 2). When the updates are done without hash calculations, the resulting FlexList looks like in Figure 5.2. Since the tag value of c_6 has changed and a new node is added between c_6 and c_7, all the nodes getting affected should have a hash recalculation. If we first perform the insert, we need to calculate hashes of n_3, n_2, c_6, n_1, c_2 and c_1. Later, when we do the modification to c_6 we need to recalculate hashes of nodes c_6, n_1, c_2 and c_1. There are 6 nodes to recalculate hashes but we do 10 hash calculations. Instead, we propose performing the insert and modify operations and call *calculateMultiHash* to indices 50 and 110. The first call of *hashMulti* goes through c_1, c_2, n_1, and c_6. On its way, it pushes n_2 to a stack since the next iteration of *hashMulti* starts from n_2. Then, with the second iteration of *calculateMultiHash*, n_2 and n_3 are added to the stack. At the end, we call the nodes from the stack one by one and calculate their hash values. Note that the order preserves the hash dependencies.

Algorithm 5.1.2: hashMulti Algorithm

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

```
   // Index of the challenged block (key) is calculated according to the
   current sub skip list root
1  i = C_first − rs
2  while Until challenged node is included do
3      cn is added to ⊔_l
       //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
4      if isIntersection(cn, C, i, last_k, rs) then
           //note that last_k becomes last_{k+1} in isIntersection method
5          state(cn.after, last_{k+1}, rs+cn.below.r) is added to ⊔_s
6      if (CanGoBelow(cn, i)) then
7          cn = cn.below //unless at the leaf level
8      else
           // Set index and rank state values according to how many bytes
           at leaf nodes are passed while following the after link
9          i -= cn.below.r; rs += cn.below.r; cn = cn.after
```

Algorithm 5.1.3: calculateMultiHash Algorithm

Input: C
Output:

```
   Let C= (i_0, ..., i_k) where i_j is the (j+1)^th altered index;
   state_m = (node_m, lastIndex_m, rs_m)
1  cn = root; rs = 0; ⊔_s, ⊔_l are empty; state= (root, k, rs)
   // Call hashMulti method for each index to fill the changed nodes
   stack ⊔_l
2  for x = 0 to k do
3      hashMulti(state.node,C,x,state.end,state.rs,⊔_l,⊔_s)
4      if ⊔_s not empty then
5          state = ⊔_s.pop(); cn = state.node ; state.rs += cn.below.r
6  for k =⊔_l.size to 0 do
7      calculate hash of k^th node in ⊔_l
```

5.1.3 Verifying Multiple Updates at Once

When *multiUpdate* algorithm is used at the server side of FlexDPDP protocol, it produces a proof vector in which all affected nodes are included and a hash value which corresponds to the root of the FlexList after all of the update operations are performed.

The solution we present to verify such an update is constructed in four parts. First, we **verify the multi proof** both by FlexList verification and tag verification. Second, we **construct a temporary FlexList** which is constituted of the parts

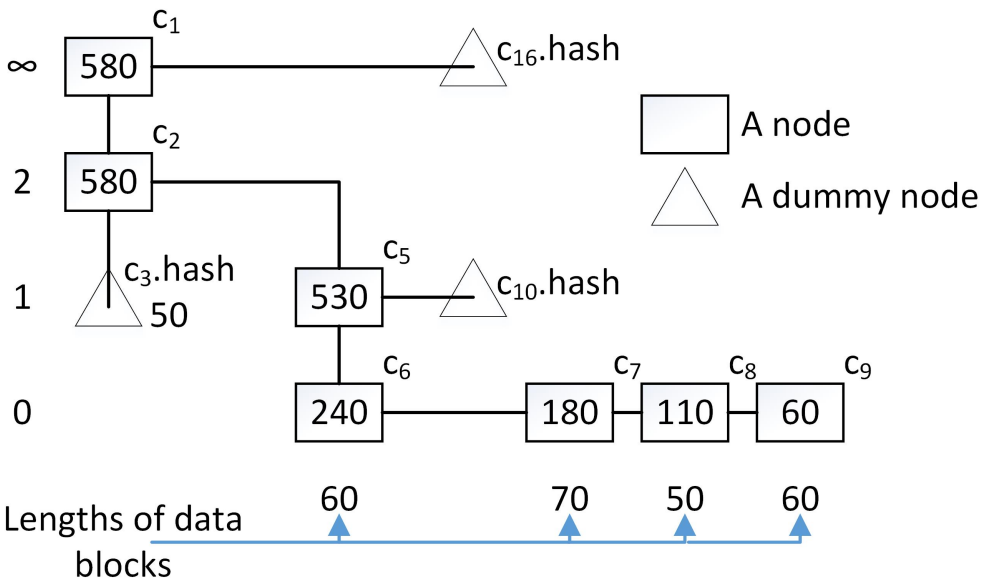

Figure 5.5: The temporary FlexList generated out of the proof vector in Figure 5.3. Note that node names are the same with Figure 5.1.

necessary for the updates. Third, we **do the updates** as they are, at the client side. The resulting temporary FlexList has the root of the original FlexList at the server side after performing all updates correctly. Fourth and last, we **check** if the **new root** we calculated is the same with the one sent by the server. If they are the same return accept and update the meta data that is kept by the client.

Constructing a temporary FlexList out of a multi proof:

Building a temporary FlexList is giving the client the opportunity to use regular FlexList methods to do the necessary changes to calculate the new root. **Dummy nodes** that we use below **are** the nodes which have some values set and **never subject to recalculation**.

We explain the algorithm 5.1.4 using the proof vector presented in Figure 5.3 and the output of the algorithm given the proof vector is the temporary FlexList in Figure

5.5. First the algorithm takes the proof node for c_1, generates the root using its values and adds the dummy after, with the hash value (of c_{16}) stored in it. Later in the loop, nodes are connected to each other depending on their attributes. The proof node for c_2 is used to add node c_2 to the below of c_1 and the c_2's dummy node is connected to its below with rank value of 50, calculated as rank of c_2 minus rank of c_5. Note that the rank values of below nodes are used in regular algorithms so we calculate and set them. These values can also be calculated on the fly, but we choose to put them while constructing the dummy FlexList). The next iteration sets c_5 as c_2's after and its dummy node added to its after. The next step is to add c_6 to the below of c_5. c_6 is both an end node and an intersection node, therefore we set its tag (from the tag vector) and its length values. Then we attach c_7 and calculate its length value since it is not in the proof vector generated by default *genMultiProof* (but we have the necessary information, rank of c_7 and rank of c_8). Next, we add the node for c_8, set its length value from the proof node and its tag value from the tag vector. And last, we do the same to c_9 as c_8. The algorithm outputs the root of the new temporary FlexList.

Verification:

Recall that U is the proof of updates generated by the client. An update information u, in U, includes an index i and if insert or modify, a block and a tag value. The client calls *verifyMultiUpdate* (Algorithm 5.1.5) with its meta data and the outputs of *multiUpdate* from the server. Using *verifyMultiProof* with P and T and block sum, if it returns accept we call *buildDummyFlexList* with the proof vector P. The resulting temporary FlexList is ready to handle updates. Again we either choose to do regular FlexList updates or do the updates and call *calculateMultiHash* algorithm after applying updates without the hash calculations. If we have the *verifyMultiProof* algorithm verify the challenged indices for updates, that means after the update, we

do have to calculate all nodes present in the temporary FlexList; therefore, we do not need to track changes to call a *calculateMultiHash* at the end but instead calculate all nodes present in the list. Last, we check if the resulting hash of the root of our temporary FlexList is equal to the one sent by the server. If they are the same we accept and update the client's meta data.

Algorithm 5.1.4: constructTemporaryFlexList Algorithm

Input: P, T
Output: root (temporary FlexList)

```
    Let P = (A_0,...,A_k), where A_j = ( level_j, r_j, hash_j, rgtOrDwn_j, isInter_j,
    isEnd_j, length_j) for j = 0,...,k; T = (tag_0,...,tag_t), where tag_t is tag
    for challenged block_t and dummy nodes are nodes including only hash
    and rank values set on them and they are final once they are created;
    //
1   root = new Node(r_0, length_0) // This node is the root and we keep this
    as a pointer to return at the end//
2   ⊔_s = new empty stack
3   cn = root
4   dumN = new dummy node is created with hash_j
5   cn.after = dumN
6   for i = 0 to k do
7       nn = new node is created with Level_{i+1} and r_{i+1}
8       if isEnd_i and isInter_i then
9           cn.tag = next tag in T; cn.length = length_i ; cn.after = nn; cn = cn.after
10      else if isEnd_i then
11          cn.tag = next tag in T; cn.length = length_i ; if r_i != length_i then
12              dumN = new dummy node is created with hash_i as hash and r_i -
                length_i as rank
13              cn.after = dumN
14          if ⊔_s is not empty then
15              cn = ⊔_s.pop() ; cn.after = nn; cn = cn.after
16      else if level_i = 0 then
17          cn.tag = hash_i ; cn.length = r_i - r_{i+1} ; cn.after = nn ; cn = cn.after
18      else if isInter_i then
19          cn is added to ⊔_s ; cn.below = nn; cn = cn.below
20      else if rgtOrDwn_i = rgt then
21          cn.after = nn
22          dumN = new dummy node is created with hash_i as hash and r_i - r_{i+1} as
            rank
23          cn.below = dumN ; cn = cn.after
24      else
25          cn.below = nn
26          dumN = new dummy node is created with hash_i as hash and r_i - r_{i+1} as
            rank
27          cn.after = dumN ; cn = cn.below
28  return root
```

Algorithm 5.1.5: verifyMultiUpdate Algorithm

Input: P, T,$MetaData$, U, $MetaData_{byServer}$
Output: accept or reject

```
    Let U= (u_0, ..., u_k) where u_j is the j^th update information
1   if !verifyMultiProof(P, T, MetaData) then
2       return reject
3   FlexList = buildTemporaryFlexList(P)
4   for i = 0 to k do
5       apply u_i to FlexList without any hash calculations
6   calculate hash values of all nodes in the temporary FlexList. //A recursive call
    from the root
7   if root.hash != MetaData_byServer then
8       return reject
9   return accept
```

Chapter 6

EXPERIMENTAL EVALUATION

In this chapter, we analyze the optimizations on FlexDPDP: preprocess, multi update and multi verification operations respectively. We show the reasons of the optimization algorithms and the gains on realistic scenarios. Then, we show the results of our implementation on the network test bed PlanetLab using real version control system workload traces.

From now on, in this Chapter, we changed our local testing computer and our local experiments are run on a 64-bit computer possessing 4 Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz CPU, 16GB of memory and 16MB of L2 level cache, running Ubuntu 12.04 LTS. Keep in mind that we keep each block of a file separately on the hard disk drive and include I/O times in our experimental analysis.

6.1 Analysis of the Preprocess Operation

Recall that at the beginning client and server should build a FlexList over the data. We reduced the amount of time required for the build operation by parallelizing the algorithm.

Parallel build FlexList performance and speedup: Figure 6.1 shows the build FlexList function's time as a function of the number of cores used in parallel. The case of one core corresponds to the sequential build FlexList function. From 2 cores to 24 cores, we measure the time spent by our parallel build FlexList function. Notice the speed up where parallel build reduces the time to build a FlexList of size

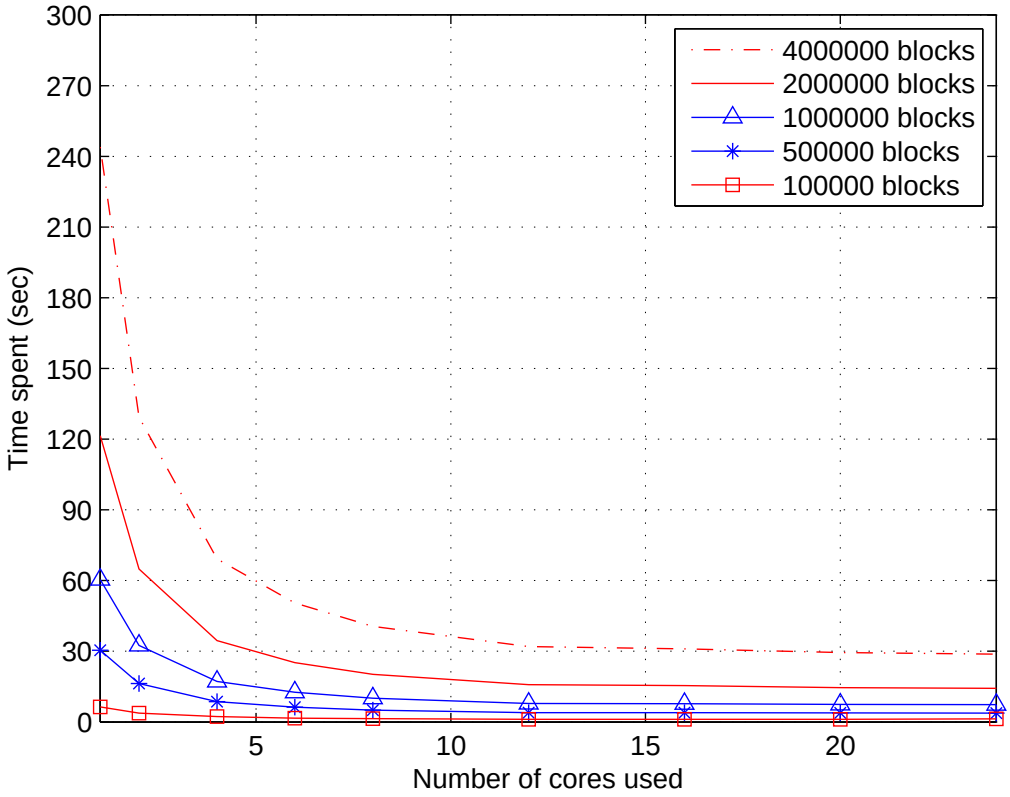

Figure 6.1: Time spent while building a FlexList from scratch.

4 million blocks from 240 seconds to 30 seconds on 12 cores, and to 40 seconds on 8 cores. The speedup values are reported in Figure 6.2 where T stands for time for a single core used and T_p stands for time with p number of cores used. The more sub-tasks created, the more time is required to divide the big task into parts and the more time is required to combine them. We see that a FlexList of size 100000 blocks doesn't get its share of the parallel build as the bigger ones do, since the sub tasks are getting smaller and the overhead of thread generation starts to surpass the gain of parallel operations. Starting from 12 cores, we observe this side effect for all sizes. For 500000 blocks (i.e. 1GB file) in size and larger FlexLists, **speed up of 6 and 7.7** are observed on 8 and 12 cores respectively.

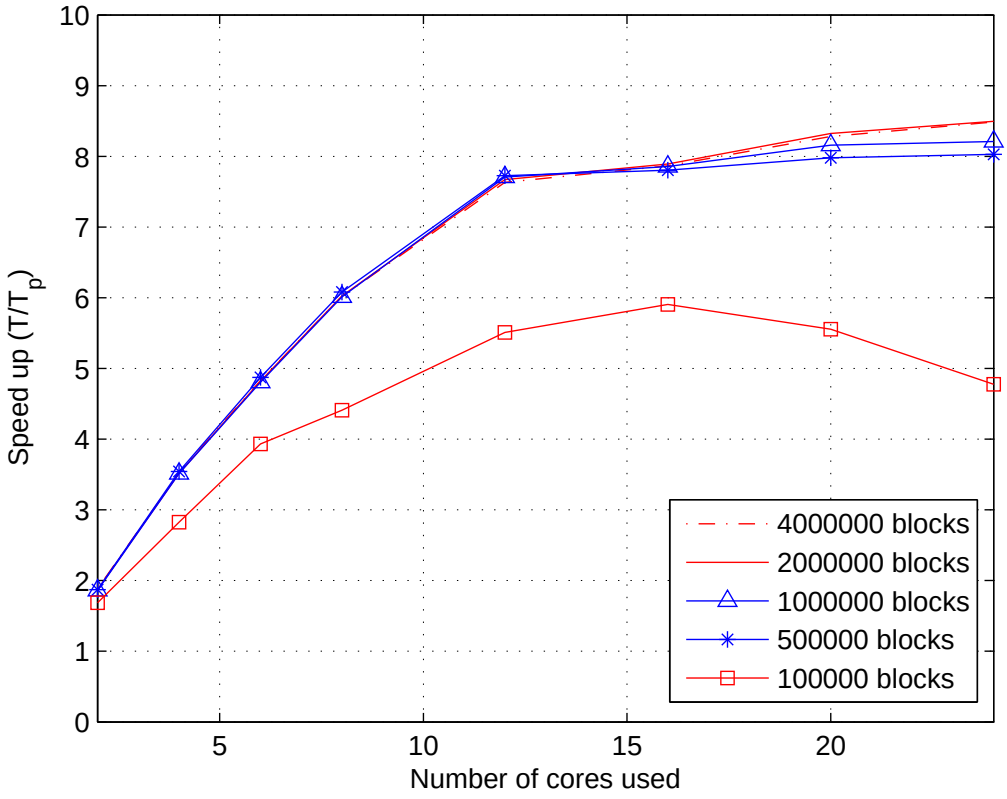

Figure 6.2: Speedup values of buildFlexList function with multiple cores. T: Time for a single core used, T_p: Time with p number of cores used.

6.2 Analysis of Multi Update Operations

Results for the core FlexList methods (insert, remove, modify) with and without the hash calculations for various sizes of FlexListare shown in Figure 6.3. To be realistic, these measurements take file I/O into consideration as well. Even with the I/O time, the operations with the hash calculations take 10 times more time than the simple operations in a file with size 4GB (i.e., 2000000 nodes). The hash calculations in an update take 90% of the time spent for an update operation. Therefore, this finding indicates the benefit of doing hash calculations at once for multiple updates in the *performMultiUpdate* algorithm.

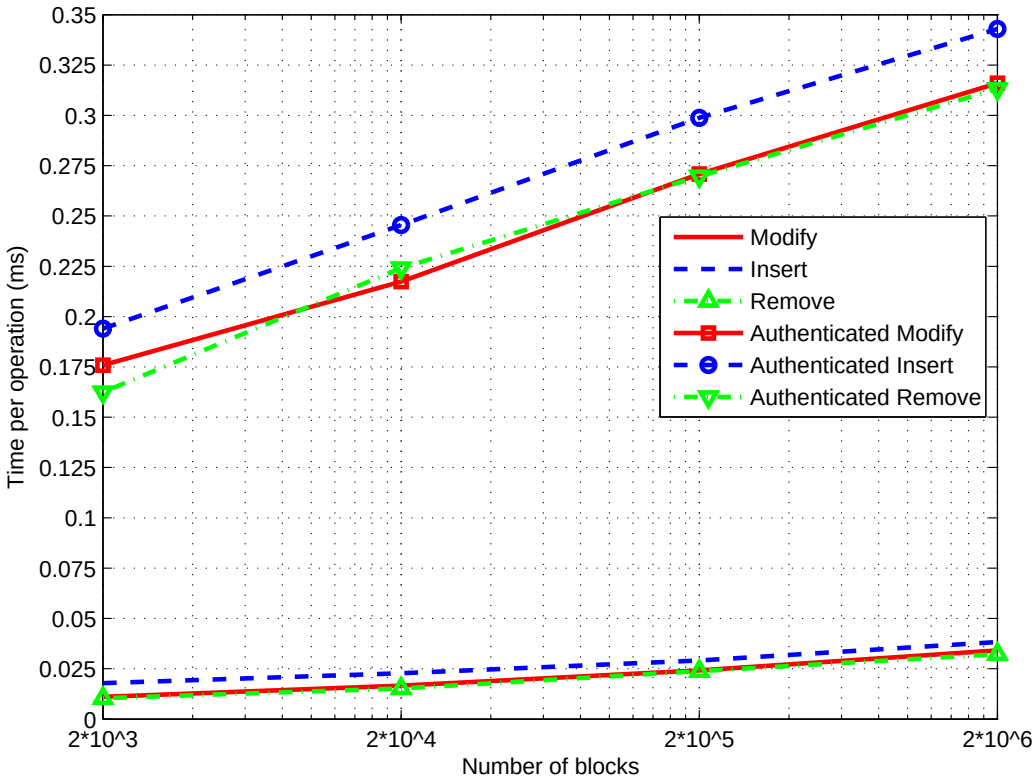

Figure 6.3: Time spent for an update operation in FlexList with and without hash calculations.

performMultiUpdate allows using multi proofs as discussed in Section 5. This provides ∼25% time and space efficiency on the verifiable update operations when the update is ∼20KB, and this gain increases up to ∼35% with 200KB of updates. Moreover, with our algorithm the update speed at the server side is increased as well since we use the *calculateMultiHash* algorithm, which calculates all possibly affected nodes' hash values at once, after a series of update operations. The time spent for an update at the server side for various size of updates is shown in Figure 6.4 with each data point reflecting the average of 10 experiments. Each update is an even mix of modify, insert, and remove operations. If the update locality is high, meaning

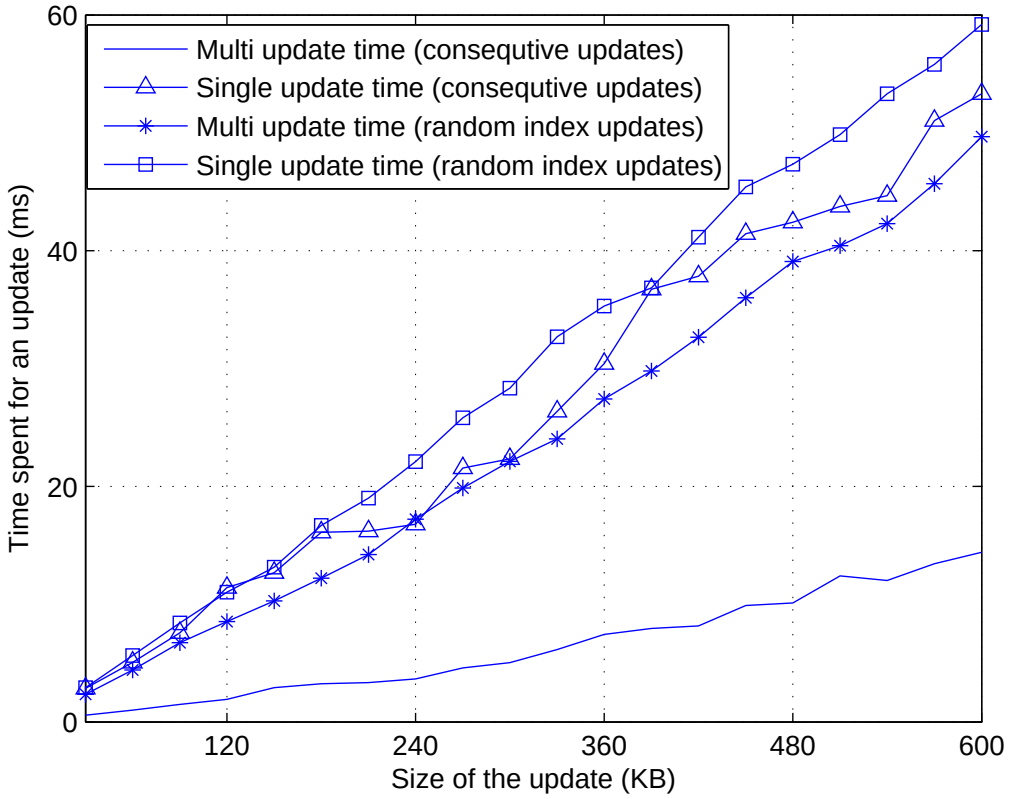

Figure 6.4: Time spent on performing multi updates against series of single updates.

the updates are on consecutive blocks (a diff operation generates several modifies to consecutive blocks followed by a series of remove if the added data is shorter than the deleted data, or a series of inserts otherwise, using our *calculateMultiHash* algorithm after the updates without hash calculation on a FlexList for a file of size 1GB, the server time for **300 consecutive update operations** (update of size 600KB) decreased **from 53ms to 13ms**. Overall, our results prove that a calculation of the hash values all together at the end is more efficient than the hash calculation at the end of each update operation.

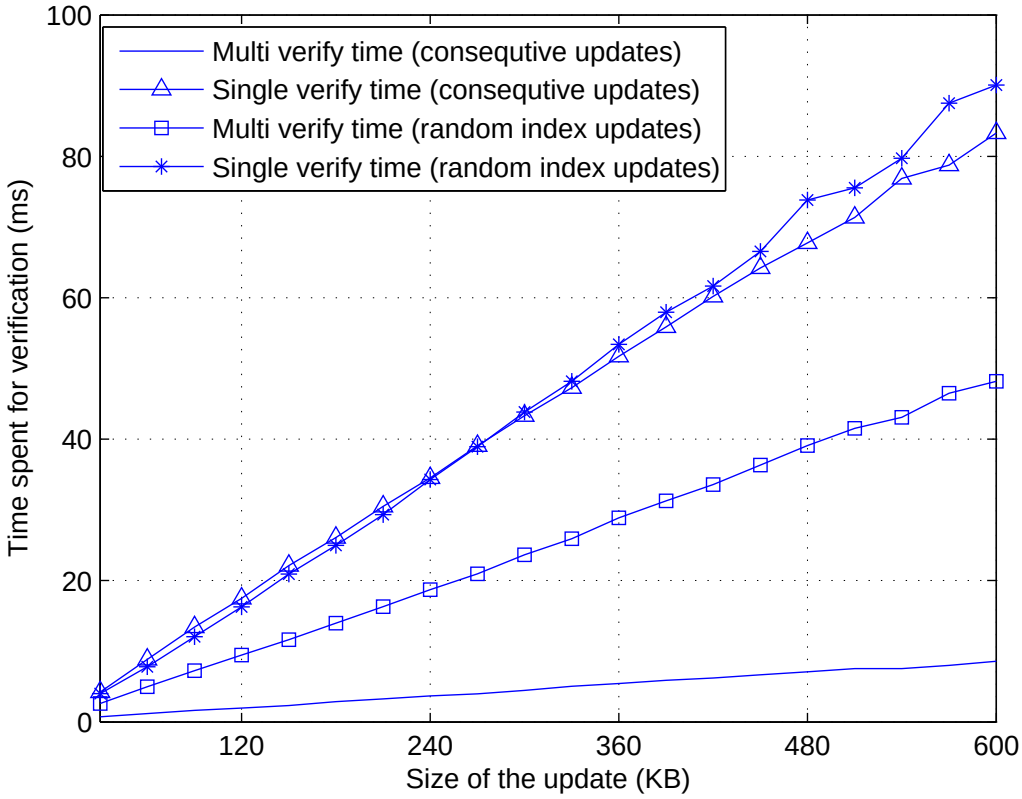

Figure 6.5: MultiVerify of an update against standard verify operations.

6.3 Analysis of the Verification Algorithms

Performance gain of verifying multiple updates at once: For the server to be able to use *multiUpdate* algorithm, the client could be able to verify multiple updates at once. Otherwise, as each single verify update requires a root hash value after that specific update, all hash values on the search path of the update should be calculated each time. Also, each update proof should include a FlexList proof alongside them. Verifying multiple updates at once not only diminishes the proof size but also provides time improvements at the server side. Figure 6.5 shows that a multi verify operation is also faster at the client side when compared to verifying all the proofs one by one. We tested the verify updates for two scenarios. One is for the updates randomly

distributed along the FlexList; the other is for the updates with high locality. The client verification is highly improved. For instance, with a file of size 1GB and an update of 300KB, a verification at client side was reduced from 45ms to less than a 5 ms. With random updates, the multi verification is still 2 times faster.

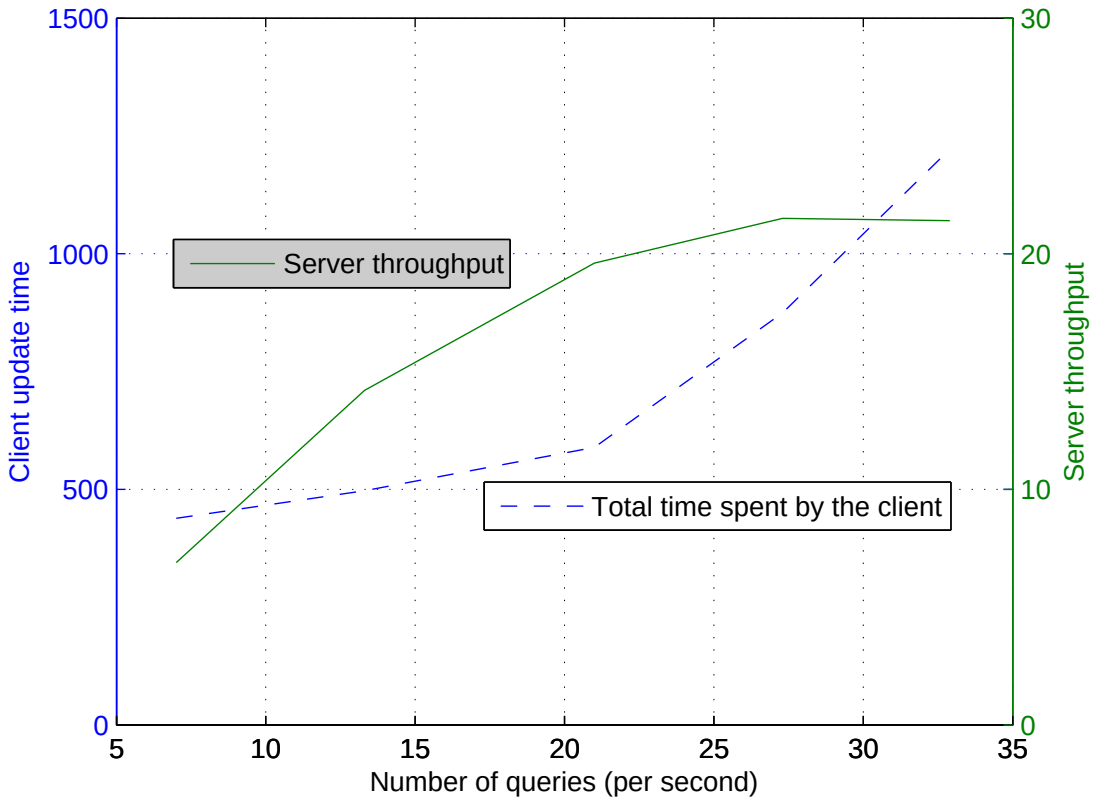

Figure 6.6: Clients challenging their data. Two lines present: first, server throughput in count per second and second, whole time for a challenge query of FlexDPDP, at client side, in ms.

6.4 PlanetLab Analysis: The Real Network Performance

This time, we chose a node, with a relatively faster connection, in Wuerzburg, Germany[1] on PlanetLab as the server which has two Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz (IC2) and 48 MBit upload and 80 MBit download speed. Our protocol runs on a 1GB file (for each client), which is divided into blocks of 2KB, having 500000 nodes. The throughput is defined as the maximum amount of queries the server can reply in a second. Our results are the average of 50 runs on the PlanetLab with randomly chosen 50 clients from all over the Europe.

6.4.1 Challenge queries

We measured two metrics, the whole time spent for a challenge proof interaction at the client side and the throughput of the server (both illustrated in Figure 6.6). As shown in the Figure, the throughput of the server is around 21. When the server limit is reached, we observe a slowdown on the client side where the response time increases from around 500 ms to 1250 ms. Given that preparing a proof of size 460 using the IC2 processor takes 40ms using *genMultiProof* on a single core, we conclude that the bottleneck is not the processing power. The challenge queries are solely a seed, thus the download speed is not the bottleneck neither. A proof of a multi challenge has an average size of 280KB (∼215KB FlexList proof, ∼58KB tags, ∼2KB blocksum), therefore to serve 21 clients in a second a server needs 47 MBit upload speed which seems to be the bottleneck in this experiment. The more we increase the upload speed, the more clients we can serve with a low end server.

[1] planetlab1.informatik.uni-wuerzburg.de

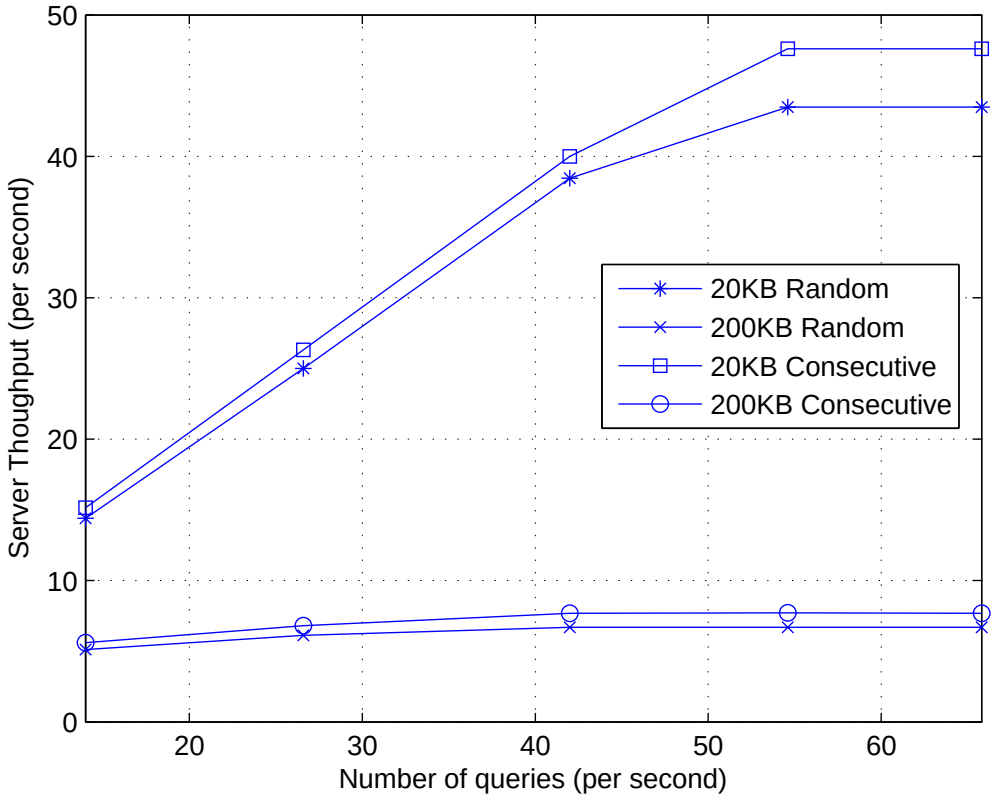

Figure 6.7: Server replies per second while the clients interacting with their data sending a query per 1.3 second.

6.4.2 Update queries

Real life usage analysis: We have conducted analysis on the SVN server where we have 350MB of data that we have been using for the last 2 years. We examined 627 consecutive commit calls and provide results for an average usage of a commit function by means of the **update locality**, the **update size** being sent through the network and the updated **block count**.

We consider the directory hierarchy proposed in Section 7.2 of [Erway et al., 2009]. The idea presented is to set root of each file's FlexList (of the single file scheme presented) in the leaf nodes of a dictionary used to organize files. The update locality

of the commits is very high. More than 99% updates in a single commit occur in the same folder, thus does not affect most parts of the dictionary but a small portion of it. Moreover, 27% of the updates are consecutive block updates on a single field of a single file.

With each commit an average of size 77KB is sent, where we have 2.7% commits of size greater than 200KB and 85% commits has size less than 20KB. These sizes are the amounts sent through the network. Note that only remove operations do not count in the size, since sending the index and number of bytes to be removed is sufficient. Erway et al. show analysis on 3 public SVN repositories. They indicate that the average update size is 28KB [Erway et al., 2009]. Therefore in our experiments on PlanetLab we choose 20KB (to show general usage) and 200KB (to show big commits) as the size sent for a commit call. The average number of blocks affected per commit provided by Erway et al. is 13 [Erway et al., 2009] and in our SVN repository, it is 57.7 which both basically show the necessity of multiple update operations.

We observe the size variation of the commits and see that the greatest common divisor of all commits is 1, as expected, thus we conclude that fixed block sized authenticated skip lists is not applicable to the cloud storage scenario. We have already explained the reason, showing an example where updates are not of fixed size 3.2.

Update size and type	**Server proof generation time**	**Corresponding proof size**
200KB (100 blocks) randomly dist.	30ms	70KB
20KB (10 blocks) randomly dist.	10ms	11KB
200KB (100 blocks) consecutive	7ms	17KB
20KB (10 blocks) consecutive	6ms	4KB

Table 6.1: Proof time and size table for various type of updates.

Update queries on the PlanetLab: We perform analysis using the same met-

rics as a challenge query. The first one is the whole time spent at the client side (Figure 6.8) and the second one is the throughput of the server (Figure 6.7), for updates of size ∼20KB and ∼200KB. We test the behavior of the system by varying the query frequency, the update size, and the update type (updates to consecutive blocks or randomly selected blocks). Table 6.1 shows the requirements for each kind of update when they are processed alone.

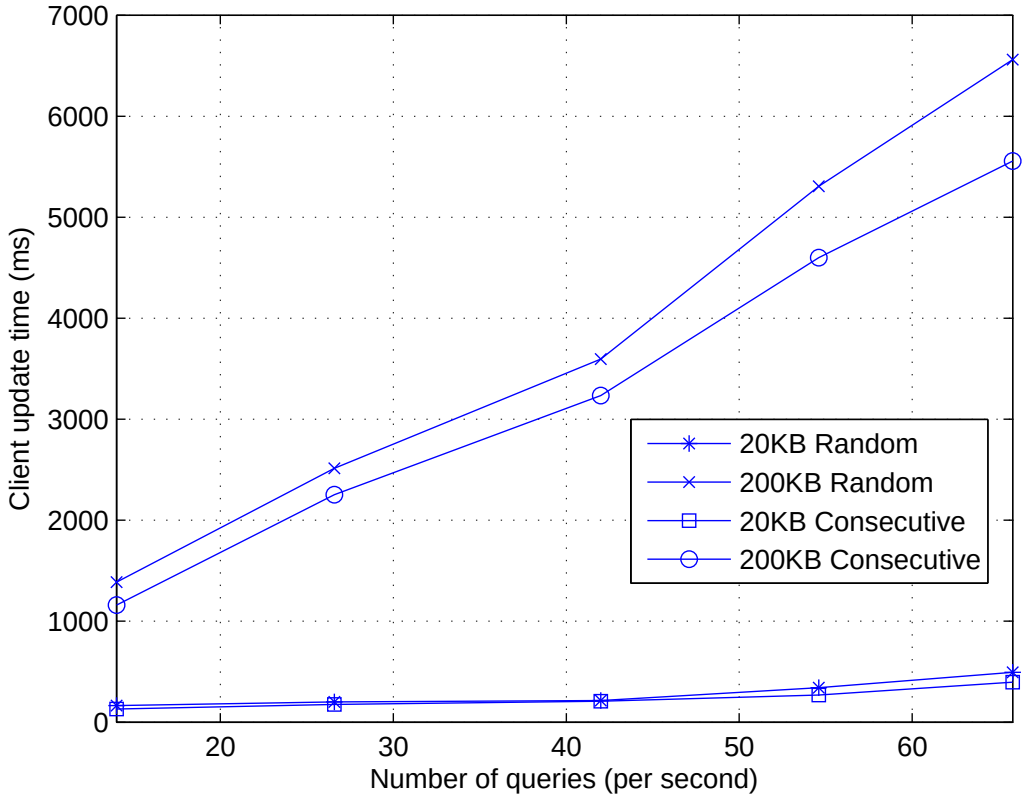

Figure 6.8: A client's total time spent for an update query (sending the update, receiving a proof and verifying the proof).

For a **randomly** distributed update of size **200KB**, a proof of size 70KB (∼55KB FlexList proof, ∼13KB tags, ∼2KB blocksum, 10 byte new hash of the root), for a **random** update of size **20KB**, a proof of size 11KB (∼8KB FlexList proof, ∼1.3KB

tags, ∼2KB blocksum, 10 byte new hash of the root), for a **consecutive** update of size **200KB**, a proof of size 17KB (∼1.5KB FlexList proof, ∼13KB tags, ∼2KB blocksum, 10 byte new hash of the root), and for a **consecutive** update of size **20KB**, a proof of size 4KB (∼1KB FlexList proof, ∼1.3KB tags, ∼2KB blocksum, 10 byte new hash of the root) is sent to the client by the server.

Figure 6.7 shows that a server can reply to ∼45 updates of size 20KB and ∼8 many updates of size 200KB per second. Figure 6.8 also approves, that the server is loaded, by the increase in time of a client getting served. Comparing update proofs with the proof size of only challenges (shown in Figure 6.6), we conclude that the bottleneck for replying update queries is not the upload speed of the server, since a randomly distributed update of size 200KB needs 70KB proof and 8 proof per second is using just 4.5 Mbit of the upload bandwidth or a randomly distributed updates of size 20KB needs a proof of size 11KB and 45 proof per second uses only 4MBit of upload bandwidth. Table 6.1 shows the proof generation times at the server side. 30ms per 200KB random operation is required so a server may answer up to 110-120 queries per second with IC2 processor and 10ms per 20KB random operation is required, thus a server can reply up to 300 queries per second. Therefore, the bottleneck is not the processing power either. Eventually the amount of queries of a size a server can accept per second is limited, even though the download bandwidth doesn't seem to be load up. It is worth noting that the download speed is checked with a single source and a continuous connection. When a server keeps accepting new connections, the end result is different. This was not a limiting issue in answering challenge queries since a challenge is barely a seed to show the server which blocks are challenged. In our setting, there is one thread at the server side which accepts a query and creates a thread to reply it. We conclude that the bottleneck is the server query acceptance rate. These results indicate that with a distributed and replicated server system a prover using FlexDPDP scheme may reply more queries.

Chapter 7

CONCLUSION

The security and privacy issues are significant obstacles toward the cloud storage adoption [Zhou et al., 2010]. With the emergence of cloud storage services, data integrity has become one of the most important challenges. Early works have shown that the static solutions with $O(1)$ 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. We also provide a novel algorithm to build a FlexList from scratch in $O(n)$ time. This optimization greatly affects the preprocess operation of our FlexDPDP scheme.

We have extended the FlexDPDP scheme with optimized and efficient algorithms, and tested their performance on real workloads in realistic settings. We obtained a speed up of 6, using 8 cores, on the preprocessing step by parallelization, 60% gain on server-side updates (commits by the client), 90% gain while verifying them at the client side. We deployed the scheme on PlanetLab and provided detailed analysis using real version control system workload traces.

This work can be extended by implementing FlexDPDP to distributed and replicated servers. It can also be employed in a peer to peer secure storage system. Other than that, we provide a dynamic provable data possession system but, in this work, we show neither what to do when a corruption is caught nor how to recover it back. As a future work, it would be nice to investigate the issue by using erasure code techniques.

BIBLIOGRAPHY

[Abraham et al., 2006] Abraham, I., Chockler, G., Keidar, I., and Malkhi (2006). Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*.

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

[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*.

[Bessani et al., 2011] Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2011). Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11. ACM.

[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.

[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.

[Cachin et al., 2009] Cachin, C., Keidar, I., and Shraer, A. (2009). Trusting the cloud. *SIGACT News*.

[Cachin and Tessaro, 2006] Cachin, C. and Tessaro, S. (2006). Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06.

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

[Chockler et al., 2009] Chockler, G., Guerraoui, R., Keidar, I., and Vukolic, M. (2009). Reliable distributed storage. *Computer*.

[Chockler and Malkhi, 2002] Chockler, G. and Malkhi, D. (2002). Active disk paxos with infinitely many processes. In *In Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC02*, pages 78–87. ACM Press.

[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*.

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

[Erway et al., 2009] Erway, C., Küpçü, A., Papamanthou, C., and Tamassia, R.

(2009). Dynamic provable data possession. In *ACM CCS*.

[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*.

[Furht and Escalante, 2010] Furht, B. and Escalante, A. (2010). *Handbook of Cloud Computing*. Computer science. Springer US.

[Gafni and Lamport, 2003] Gafni, E. and Lamport, L. (2003). Disk paxos. *Distrib. Comput.*

[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 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*.

[Goodson et al., 2004] Goodson, G., Wylie, J., Ganger, G., and Reiter, M. (2004). Efcient byzantine-tolerant erasure-coded storage. In *Proc. of the Int. Conference on Dependable Systems and Networks*, DSN '04, page 135144.

[Hendricks et al., 2007] Hendricks, J., Ganger, G. R., and Reiter, M. K. (2007). Low-overhead byzantine fault-tolerant storage. In *Proceedings of twenty-first ACM*

SIGOPS symposium on Operating systems principles, SOSP '07. ACM.

[Jayanti et al., 1998] Jayanti, P., Chandra, T. D., and Toueg, S. (1998). Fault-tolerant wait-free shared objects. *J. ACM*.

[Jensen et al., 2009] Jensen, M., Schwenk, J., Gruschka, N., and Iacono, L. L. (2009). On technical security issues in cloud computing. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 109–116. IEEE.

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

[Liskov and Rodrigues, 2006] Liskov, B. and Rodrigues, R. (2006). Tolerating byzantine faulty clients in a quorum system. *2012 IEEE 32nd International Conference on Distributed Computing Systems*.

[Malkhi and Reiter, 1998] Malkhi, D. and Reiter, M. (1998). Byzantine quorum systems. *Distrib. Comput.*

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

[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*.

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

[PlanetLab, 2013] PlanetLab (2013). Planetlab node requirements. https://planet-lab.org/node/225 [Online; accessed 20-March-2013].

[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*.

[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.*

[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*.

[Wooley, 2011] Wooley, P. S. (2011). Identifying cloud computing security risks. Technical report, 7 University of Oregon Eugene.

[Yuan and Yu, 2013] Yuan, J. and Yu, S. (2013). Proofs of retrievability with public verifiability and constant communication cost in cloud. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13. ACM.

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

[Zhang and Blanton, 2013b] Zhang, Y. and Blanton, M. (2013b). Efficient dynamic provable possession of remote data via balanced update trees. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications*

security, ASIA CCS '13.

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

[Zhou et al., 2010] Zhou, M., Zhang, R., Xie, W., Qian, W., and Zhou, A. (2010). Security and privacy in cloud computing: A survey. In *IEEE SKG*.

VITA

ERTEM ESİNER was born in İstanbul, Turkey on August 18. He received his French baccalaureate from Galatasaray Lycee, İstanbul in 2007 and his B.S degree in Computer Engineering from Koç University, İstanbul in 2011. He joined M.Sc. program in Computer Science and Engineering at Koç University in 2011 as a research and teaching assistant. During his study he worked on Secure Cloud Storage Systems as part of the Cryptography, Security, and Privacy Research Group at Koç University. He has co-authored a conference paper in ISCIS'11, a journal paper (under submission) and two conference papers (under submission).