Real-time POV for Cloud Storage
Without Caching Hash Values of Files

Hung-Fu Chen
Abstract

Real-time POV for Cloud Storage Without Caching Hash Values of Files

Hung-Fu Chen

Cloud storage services are becoming very popular due to their usefulness: many companies use such services to store important files. However, there are still many unsolved security issues. Users can not verify whether the files stored in the cloud storage are correct. The content of files may be changed, and the order of write and read operations may be inconsistent. Clients need a way to ensure the consistent behavior of their cloud storage. To address this problem, we propose a light and fast way to audit files, called Real-time POV. Client devices do not need to cache any hash values of files in order to verify that the service behaves correctly, but just need to fetch two values of each 1 kB from a synchronization server before each operation. The cloud storage only needs to maintain a small structure of authenticity information, called an FBHTree, to preserve the authenticity information for its files. The clients and the cloud storage exchange attestation data with each operation. This system protects not only the clients, but also protects the service provider of cloud storage from malicious accusations. It also outperforms previous methods by one to two orders of magnitude.
CONTENTS

LIST OF TABLES .............................................................................................................................III

LIST OF FIGURES ...........................................................................................................................IV

1. INTRODUCTION ..........................................................................................................................1
   1.1 CLOUD STORAGE .....................................................................................................................1
   1.2 PROVE OF VIOLATION (POV) ..............................................................................................2
   1.3 GOAL .....................................................................................................................................2

2. A NOVEL REAL-TIME POV SCHEME .........................................................................................3
   2.1 HASH TREE ............................................................................................................................3
   2.2 SYSTEM ARCHITECTURE .......................................................................................................4
   2.3 FBTTree AND INDEX FUNCTION ........................................................................................5
   2.4 SLICE OF FBTTree ................................................................................................................7
   2.5 UPDATE SLICE .......................................................................................................................8
   2.6 TRANSFER AND derive SLICE FOR ROOT HASH .................................................................9
   2.7 AUDIT IN WRITE MODE .........................................................................................................10
   2.8 AUDIT IN READ MODE .........................................................................................................11
   2.9 EFFICIENT OPERATION OF FBTTree ....................................................................................12

3. PROTOCOL ..................................................................................................................................15
   3.1 WRITE A FILE .......................................................................................................................15
   3.2 READ A FILE .........................................................................................................................17

4. EXPERIMENTS AND RESULTS .................................................................................................21

5. DISCUSSION AND FUTURE WORK .........................................................................................29
   5.1 DISCUSSION ..........................................................................................................................29
   5.2 FUTURE WORK ......................................................................................................................30

6. RELATED WORK ........................................................................................................................31

7. CONCLUSION .............................................................................................................................32

8. REFERENCE ..................................................................................................................................33
LIST of TABLES

Table 1. The collision behavior by index function .................................................. 21

Table 2. The time required on slice manipulation ..................................................... 22

Table 3. The memory usage on Slice and FBHTree .................................................. 23

Table 4. The size of Slice in different tree height and link length ............................. 23

Table 5. Comparison the time required between Non-POV Scheme and Real-time POV
        Scheme on reading a file in the same network segment (in ms) ....................... 25

Table 6. Comparison time required between Non-POV Scheme and Real-time POV
        Scheme on writing a file in the same network segment (in ms) ....................... 26

Table 7. Comparison the time required between Non-POV Scheme and Real-time POV
        Scheme on reading a file in different network segment (in ms) ....................... 27

Table 8. Comparison the time required between Non-POV Scheme and Real-time POV
        Scheme on writing a file in different network segment (in ms) ....................... 28
LIST of FIGURES

FIG. 1. SYSTEM ARCHITECTURE ............................................................................................................. 4

FIG. 2. A FULL BINARY HASH TREE WITH 4 LEVELS IN HEIGHT ...................................................... 5

FIG. 3. THE SLICE OF AN FBHTREE FROM LEAF NODE ID 97 .............................................................. 7

FIG. 4. THE UPDATED NODES IN A SLICE ........................................................................................... 8

FIG. 5. A LIST OF A SLICE FROM AN FBHTREE WITH 9 LEVELS IN HEIGHT ..................................... 9

FIG. 6. TRANSFORM A LIST INTO A SLICE FROM LEAF NODE ID 97 .................................................. 9

FIG. 7. THE RELATION BETWEEN AN FBHTREE AND ONE-DIMENSION ARRAY ............................ 12

FIG. 8. PROTOCOL TO WRITE A FILE .................................................................................................. 15

FIG. 9. PROTOCOL TO READ A FILE .................................................................................................. 17

FIG. 10. DIFFERENT FILE AND DIFFERENT SLICE ........................................................................ 19

FIG. 11. SAME SLICE BUT DIFFERENT LEAF NODE ........................................................................ 19

FIG. 12. SAME LEAF NODE BUT DIFFERENT FILE ........................................................................... 20

FIG. 13. CHARTS TO SHOW THE RELATION IN TABLE 5 ................................................................. 25

FIG. 14. CHARTS TO SHOW THE RELATION IN TABLE 6 ................................................................. 26

FIG. 15. CHARTS TO SHOW THE RELATION IN TABLE 7 ................................................................. 27

FIG. 16. CHARTS TO SHOW THE RELATION IN TABLE 8 ................................................................. 28

FIG. 17. THE RATE OF TIME CONSUMING ......................................................................................... 29
CHAPTER 1

Introduction

1.1 Cloud Storage

Cloud storage is a network application that provides people a convenient way to store files. It provides high accessibility and reliability to a remote and virtualized storage server. Individuals and companies use it to backup or share files with each other. Cloud storage service include Dropbox [1], OneDrive [2], iCloud [3] and Amazon S3 [4].

However, clients lose control of files when uploading to the cloud storage. Files may be unexpectedly exposed to a third party or modified without permission. The cloud may not follow user’s instruction to safely store files. Security issues like authentication, confidentiality, data integrity and non-repudiation are potential problems. Some of them can be solved by using cryptographic cloud storage [5]. Encryption avoids data leaking to others. Digital signatures guarantee the integrity of files and support access control. Nevertheless, the method can not provide a solution to Roll-back Attack [6]. Cloud storage may accidentally reverse files to previous version or even lose them.

Client devices usually can access files in cloud storage simultaneously. But they cannot sure that the cloud storage provides a safe concurrency control, i.e. serializability on writing and freshness on reading. Cloud storage may modify the order of transaction after client devices finish each one. Although some paper provide methods for client devices to detect the violations [7], the cloud still can repudiate what they have done.

Some cloud storages claim that they provide monitors to measure the quality of service, for example Amazon S3 and Microsoft Azure [10]. But the monitors are maintained by themselves. These cloud storages can modify the result in monitors at any time to assert that they can achieve the quality listed on Service Level Agreement (SLAs) [11]. This is the situation of a player acting as a referee and not a fair solution for client devices.

All of the methods mentioned above can not fairly protect both the clients and cloud storage. If there is controversy happening to files, there will be no fair evidence to show an impartial third party, such as court, to get reasonable compensation from the other.
1.2 Proof of Violation (POV)

Gwan-Hwan Hwang, Jenn-Zjone Peng, and Wei-Sian Huang proposed a scheme for Proof of Violation (POV) [12]. It is a concept for the clients and the cloud storage to have an agreement on storing authenticity information from files on both sides. The information can be used to verify the correctness of files and capable to clarify controversies. Gwan-Hwan Hwang implements the concept by defining a procedure that involves three parts: Criteria, Cryptographic proofs and Audit. Criteria means the claim listed on the contract of SLAs, such as Integrity of files and Consistency on the actions of reading and writing [13]. With Criteria, the scheme can process files into cryptographic proofs during transaction. Those proofs are processed and stored. Proofs can not be tampered by clients and cloud storage. With the proofs, clients can audit the files stored on the cloud to assure that they meet criteria.

However, Gwan-Hwan Hwang’s work can only support clients audit files at the end of a period of time, such as few days or weeks. We found it unsuitable for some important cases, ex. business document. If the client can not instantly check the correctness of files at the moment he/she retrieves, he/she may suffer from a huge loss of money for the incorrectness. Gwan-Hwan Hwang’s continuous work [13] can cache partial hash tree on client devices [14] and synchronizes by synchronization server to achieve Real-time POV, but the system still costs 80 to 100 times overhead on each transaction. The huge overhead motivates me to propose a new scheme for improvement.

1.3 Goal

In the new system, efficiency will be increased to one to two magnitude. Client devices do not need to cache any hash values of files, but still can achieve the function of Real-time POV. Furthermore, the hash value of files will be stored in a new data structure proposed in this paper. The scheme is more practical for using cloud storage.

This paper is organized as followed: Chapter 2 will introduce the scheme of Real-time POV. Chapter 3 will present the protocol of writing and reading files under the scheme of Real-time POV. Chapter 4 presents the the detail of implementation and experimental result. Chapter 5 and 6 present related work and conclusions.
Chapter 2
A Novel Real-time POV Scheme

2.1 Hash Tree

Before presenting our Real-time POV scheme, we want to show an intuitive solution for storing the information of files. Hash values are small-size datum for the purpose of detecting errors. They can be produced by cryptographic hash function. For checking the integrity of a group hash values, many mechanisms use a unique root value produced by a tree structure to easily store and share with others. The tree is known as Merkle tree or Hash tree [15]. For example, we can use SHA-256 to transform the context of files into hash values separately. For the structure of hash tree, we can follow the hierarchy of a file system. The top of hash tree is a hash value, called root hash, from merging and calculating hash values from leaf nodes. In the environment of cloud storage, clients can simply use root hash to assure the status of the files stored in cloud storage, because any modification of values in the Hash Tree leads to a different root hash.

However, we found out problems on using hash tree in previous papers [12], [13]. For clients to use hash tree, he/she has to either cache them on local devices or download them from cloud storage. The two solutions have different effects on efficiency. The first is that each client device need to synchronize hash tree with all the others while updating local hash tree. If there were any client device not online [16] to get the broadcasted information, client device will be in the danger of suffering rollback attacks by cloud storage [6]. The second is that each client device has to download hash tree from cloud storage while accessing files. Time required to transfer hash tree back and forth is an overhead.

The structure of hash tree following the hierarchy of file system is also a problem. It turns a hash tree into an unbalanced structure. If there are many files located in a directory, calculating a root hash will need to spend much more time than only one file in a directory. We also can not anticipate the time required on calculating a root hash for each file. The dynamic structure also forces the nodes in hash tree connected by pointers, which make the procedure of accessing hash value counts on sequentially travers instead of randomly access.
2.2 System architecture

In this chapter, there is an overview on system architecture. The system involves a cloud storage, a synchronization server and some client devices that user employs to access his/her account in the cloud storage. We can see the relation among them in Fig. 1.

![System Architecture Diagram]

**Fig. 1. System Architecture**

The cloud storage is responsible for storing the files from client devices and maintaining hash values of files in the FBHTree. The FBHTree, Full Binary Hash Tree, is a new structure proposed in this paper to increase the efficiency of storing and manipulating hash values. That is, this paper does use the structure of hash tree following hierarchy of file system that mentioned in Chapter 2.1. The detail of manipulating the FBHTree will be discussed in chapter 2.3. For determining the location of leaf nodes to store hash value of files, client devices and cloud storage agree on an index function $\Gamma$ when the system starts. The main idea of proposed scheme is to let client device no need to fetch the whole FBHTree from cloud storage for auditing a file, but only a slice of it.

The synchronization server is maintained by all client devices in local environment or implemented on another cloud provider. Synchronization server is responsible for helping client devices transfer requests to the cloud storage one by one and deliver cryptographic proofs among client devices. That is, it locks the next client until the present client returns a new proof. The proof is a three-tuple values ($\gamma$, SN, Sig) generated by the cloud storage. “$\gamma$” is the root hash in FBHTree. “SN” is the sequence number of request for read and write. It will be initialed to 1 in the first transaction, and add 1 in the next. “Sig” is digital signature on the composition of $\gamma$ and SN by cloud storage’s private key [15]. It is to assure the order of root hash. That is, client devices can assure the correctness of files fetched from cloud storage. The establishment of
synchronization server also provides the advantage that client devices do not need to be online all the time for the FBHTree.

### 2.3 FBHTree and Index function

![FBHTree Diagram]

**Fig. 2. A Full Binary Hash Tree with 4 levels in height**

**Pair value** = \( \text{hash}^1 (\text{path name}) \mid \text{hash} (\text{path name} \mid \text{hash} (\text{file})) \)

**Pair List** = Pair value \(1_{st} \rightarrow\) Pair value \(2_{nd} \rightarrow\) ... \(\rightarrow\) Pair value \(N_{th}\)

**Leaf node** = \(\text{hash} \text{ (Pair List)}\)

**Internal node, Root node** = \(\text{hash (left child node} \mid \text{Right child node)}\)

**SN** = Sequence Number

In this Chapter, we are going to introduce the structure of an FBHTree and how it works. The structure of an FBHTree is similar to original hash tree on storing one hash value in each node, but only two child nodes are connected to a parent node. That is, there will be totally \(2^N-1\) nodes and \(2^N-1\) leaf nodes if the tree is \(N\) levels in height. For example, Fig. 2 is an FBHTree with 4 levels in height and 15 nodes in totally. An FBHTree can be split into five different parts: Pair lists, Leaf nodes, Internal nodes, a Root node (\(γ\)) and a Sequence number (SN).

Each leaf node is in the same level and has a leaf node ID according to the order from left to right. Each leaf node ID also refers to a list of hash values that connected in a structure of link list, which is called pair list here. Those hash values are exactly from the information of clients’ files, i.e. pathname and content.

---

1 hash is referred to SHA-256 hash function
In original Hash tree, hash value of a file is directly stored in a certain leaf node according to hierarchy of a file system. In an FBHTree we use pathname of a file to determine the leaf node ID that a file belongs to. The procedure is through index function $\Gamma$ that looks like below.

$$\Gamma (\text{Pathname}) = \text{hash (Pathname)} \mod 2^{N-1}$$

Pathname is the only input in this function. That the hash value from hashing a pathname modulo the number of leaf nodes will come out the output from 0 to $2^{N-1}-1$, if the FBHTree is N levels in height. For example, $\Gamma(/d1/d3/d5/f2) = 3$. It means the information of a file f2 from directory d1, d3, d5 will be mapped into a pair list that referred by leaf node ID 3. Through the function, we can see different files could be mapped into the same leaf node ID, so we use pair list to connect them. In the second experiment of Chapter 4, we will study the influence on efficiency from different length of a pair list.

Leaf node here stores the hash value from hashing the pair list in the same leaf node ID. Internal node stores the hash value by hashing left and right child nodes. Repeat the operation to the top of an FBHTree, we can get a unique hash value in root node, called root hash or “γ”. Client devices can merely conserve the root hash to assure the status of files stored in cloud storage.

For the height of an FBHTree, the first client who registers an account in the cloud storage determines the height by the number of files going to store. The height remains the same since an account starts and till the end. From the first to forth experiments in chapter 4, we studied the best configuration on the height referring to the number of files. When the height is setup, the first client device and synchronization server will get the first $\gamma$, SN and cloud’s digital signature on them.
2.4 A slice of an FBHTree

![Diagram of FBHTree]

Fig. 3. The slice of an FBHTree from leaf node ID 97

A slice of FBHTree = Root node | Left Internal node | Right internal node | ... | Left Leaf node | Right Leaf node.

A slice is the most important unit structure in an FBHTree. It has three functions. First, cloud storage uses it as the route to update an FBHTree with the hash value of a file from client devices. Second, client device uses it to assure that cloud storage updates the hash value of a file correctly on an FBHTree. Last, client device uses it to audit the correctness of files from cloud storage.

Using a slice can bring many benefits. Each leaf node ID = i refers to a slice denoting to slice(i). From a leaf node ID, slice consists a list of nodes by tracing the route from a pair list to the root. Each parent and sibling nodes passing through are the elements of the list. That is, there will be only a pair list and 2N-1 nodes in a slice where N is the height of an FBHTree. It is a slim area and every slice is anticipated to have the same number of nodes. Fig. 3 is an example on slice(97) from an FBHTree with 9 levels in height. The usage of index function \( \Gamma \) also disperse the length of a pair list. In the first experiment of chapter 4, we can see the result.

Client devices can use any slice in an FBHTree to derive a root hash. A slice includes all the information on the route from a pair list, leaf nodes and internal nodes. Using the root hash from a slice and the one from synchronization server, client devices can determine the correctness of the slice, so as to determine the correctness of a file.
2.5 Update a slice

![Diagram of FBHTree with updated nodes and new root hash and SN]

**Fig. 4. The updated nodes in a slice**

Here we are going to show how a cloud storage follows a procedure to update an FBHTree by updating the slice that a file refers to. There are totally 4 steps.

**Step 1**: In the beginning, the cloud storage needs to have three elements from a client device. It includes a *Path Name*, a new *Hash Value* of a file, and an old *Sequence Number*. The *Sequence Number* is to notify cloud storage the version of FBHTree in this transaction.

**Step 2**: The cloud storage uses *Path Name* and index function \( \Gamma \) to find a leaf node ID and the pair list that the ID refers to. In the pair list, cloud storage uses *Path Name* again to sequentially search the pair value in pair list and update it with the new *Hash Value*.

**Step 3**: With the updated pair list, cloud storage can update the leaf node with the same leaf node ID, internal nodes and root node from bottom of slice to the top. Then a *new Root Hash* will come out. For example, the nodes in Fig. 4 with cross on them are the ones that are updated through the route in slice. This is the 1\(^{st}\) output produced in the operation.

**Step 4**: The cloud storage increases old *Sequence Number* by one to get a new *Sequence Number*. This is the 2\(^{nd}\) element produced in the operation.
2.6 Transfer slice and derive root hash

Every time a client tries to WRITE or READ a file, he/she always audit a slice from the cloud storage for complete the operation. In the WRITE mode, the client uses a slice to assure that the cloud storage updated the hash value of a file correctly on an FBHTree; In the READ mode, the client uses a slice to assure that the file from the cloud storage is correct. In this system, the cloud storage has to extract a slice from an FBHTree and send it to a client device. A slice will be transformed into a list structure before transferring, such as the example in Fig. 5. The list includes the index of a pair value the file refers to, pair list and the nodes in a slice.

<table>
<thead>
<tr>
<th>Index of PV</th>
<th>Length of PV</th>
<th>PV_1</th>
<th>PV_2</th>
<th>...</th>
<th>PV_n</th>
<th>L9</th>
<th>R9</th>
<th>L8</th>
<th>R8</th>
<th>...</th>
<th>L2</th>
<th>R2</th>
<th>Root</th>
</tr>
</thead>
</table>

PV = pair value  
L9 = left node in level 9  
R9 = right node in level 9

**Fig. 5. A list of a slice from an FBHTree with 9 levels in height**

When a client receives the list, he/she transforms it back into a slice according to the leaf node ID from index function $\Gamma$ and calculate root hash, such as the example Fig. 6.

**Fig. 6. Transform a list into a slice from leaf node ID 97**
2.7 Audit a slice in WRITE Mode

When the cloud storage finished updating a slice (mentioned in chapter 2.5), the client device has to audit that cloud storage did it correctly. Client device collects the information that includes a Root Hash “r” and a Sequence Number “SN” from synchronization server; an Old Slice, a new Root Hash and new Sequence Number from the cloud storage. There are totally 5 steps in the procedure.

**Step 1** : The client device derives the Root Hash from the Old Slice.

**Step 2** : Compare the Root Hash from step1 with the Root Hash from synchronization server. If they are the same, we could assure that the Old Slice comes from the last transaction.

**Step 3** : Find the pair value in pair list from the index of pair value in Old Slice. We can see the example of index in Fig. 5. Then update the pair value with Path Name and new Hash Value. Here we get an updated pair value.

**Step 4** : The client device derives the Old Slice again from leaf node to the root. He/she gets an updated Root Hash.

**Step 5** : Compare the updated Root Hash in step 4 with the new Root Hash from cloud storage. If they are the same, we can declare that cloud storage updated hash value of file correctly on an FBHTree.
2.8 Audit a slice in READ Mode

In READ mode, the client device always audits the file from the cloud storage. The action of auditing is to determine the correctness of a file. Before a client device starts to audit, he/she has to collect some information. It includes a Root Hash and a Sequence Number from the synchronization server; a Slice and a File from the cloud storage. There are totally 4 steps in the procedure.

**Step 1**: The client device derives the Root Hash from Slice.

**Step 2**: Compare the Root Hash and Sequence Number from step 1 with the Root Hash from the Synchronization Server. If the two Root Hash are the same and SN is plus by one, we can assure the Slice from the correct FBHTree.

**Step 3**: Calculate the hash value by hashing the Path Name and hash value of the File.

**Step 4**: Find the pair value in pair list by the index along with Slice. Compare the hash value in pair value with the hash value from step 3. If they are the same, we can declare that the File is correct.
2.9 Efficient operation of FBHTree

In this chapter, we are going to show how to implement an FBHTree and study the efficiency. Besides the dynamic length of a pair list, the number of nodes (leaf nodes, internal nodes and root node) in an FBHTree remains unchanged since an account is created, so we propose to use one-dimensional array as data structure to implement the nodes. For example, if there is an FBHTree with $N$ levels in height, we use a one-dimensional array with $2^N-1$ elements to store the nodes. Each node from root to the bottom of an FBHTree has a tree node ID that sequentially maps to the index in the array. Such as the example on Fig. 7, we use one-dimensional array with $2^4-1 = 15$ elements to implement the nodes from FBHTree with four levels in height.

From the slice mentioned in chapter 2.4, we know that each slice is referred by a unique leaf node ID. When we want to extract the nodes in a slice, we can use the leaf node ID to get a tree node ID for indexing the elements in one-dimensional array. For example, a leaf node ID, $I$, can derive a tree node ID, $X$, by the function $X = I + 2^{N-1}$ where $N$ is the levels of an FBHTree in height. A node with tree node ID, $X$, can derive its parent node with tree node ID, $P$, by the function $P = \lfloor X + 2 \rfloor$. A node with tree node ID, $X$, can derive its sibling node with tree node ID, $S$, by the function $S = X + 1$ or $S = X - 1$.

![Fig. 7. The relation between an FBHTree and one-dimension array](image)

Algorithm 1 shows the procedure to extract a slice from an FBHTree stored in a one-dimensional array. Because $\Psi$ is a one-dimensional array, fetching a hash value does not need to traverse through pointers in a binary tree but the direct access to an array to pick up elements from it.
Algorithm 1: Extract a slice from an FBHTree. Assume that the height of the FBHTree is N.

**Input:** I: a leaf node ID
Ψ: a one-dimensional array which stores an FBHTree

**Output:** Slice(I)

1. Slice(I) = φ
2. $X = I + 2^{N-1}$ // Transforming leaf node ID to tree node ID
3. **WHILE** (X ≠ 1) **DO** // Bottom up to the root node
   - **IF** X is an even number **THEN**
     - Slice(I) = Slice(I) ∪ Ψ[X] ∪ Ψ[X+1]
   - **ELSE**
     - Slice(I) = Slice(I) ∪ Ψ[X-1] ∪ Ψ[X]
   - **END IF**
   - $X = \lceil X / 2 \rceil$ //Unconditional rounding
   - **END WHILE**
4. Slice(I) = Slice(I) ∪ Ψ[1]

Here use the FBHTree and array shown in Fig 7 to illustrate Algorithm 1. We assume to extract slice(3) from an FBHTree when $N = 4$, $I = 3$, and $Ψ = \{ h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}, h_{13}, h_{14}, h_{15} \}$. In the step (3), we have slice(3) = $\{ h_{10}, h_{11}, h_4, h_5, h_2, h_3 \}$ at the time when values of X are 11, 5 and 2. In the step (4), we have slice(3) = $\{ h_{10}, h_{11}, h_4, h_5, h_2, h_3, h_1 \}$.

Algorithm 2 is to derive the root hash from a slice. WHILE loop of step (5) iterates the hashing and concatenating operation from bottom of the slice to the top. The complexity is $O(N)$ when $N$ is the height of an FBHTree.

Algorithm 2: Derive the root hash from a slice. Assume that the height of the FBHTree is N.

**Input:** S: A slice  I: A leaf node ID

**Output:** $γ$: Root hash of $S$

1. **IF** (Number of elements in S) ≠ $(2*N-1)$ **THEN**
   - RETURN ERROR **END IF**
2. $X = I + 2^{N-1}$ // Transforming leaf node ID to tree node ID
3. $pt = 0$
4. **IF** (X is an even number) **THEN**
   - $γ = S[pt]$
   **ELSE**
\[ \gamma = S[pt+1] \]

\text{END IF}

(5) \text{WHILE}([X+2] \neq 1) \text{ DO}

\hspace{1cm} \text{IF (X is an even number) THEN}

\hspace{2cm} \gamma = \text{hash}(\gamma \mid S[pt+1])

\hspace{1cm} \text{ELSE}

\hspace{2cm} \gamma = \text{hash}(S[pt] \mid \gamma)

\hspace{1cm} \text{END IF}

\hspace{1cm} pt = pt + 2

\hspace{1cm} X = [X+2]

\text{END WHILE}

(6) \text{IF (X is an even number) THEN}

\hspace{1cm} \gamma = \text{hash}(\gamma \mid S[pt+1])

\hspace{1cm} \text{ELSE}

\hspace{2cm} \gamma = \text{hash}(S[pt] \mid \gamma) \text{ END IF}

Here continues with our previous example when \( N = 4 \), \( I = 3 \) and slice(3) = \{h_{10}, h_{11}, h_4, h_5, h_2, h_3, h_1\}. According to algorithm 2, we have \( \gamma = \{h_{11}\} \) in step (4), because 11 is an old number. The step (5) iterates the hashing action with child nodes for parent nodes, so we have \( \gamma = hash(h_{10} | h_{11}) \) and \( \gamma = hash(h_4 | hash(h_{10} | h_{11})) \) for the loop. Finally, we have the root hash \( \gamma = hash(hash(h_4 | hash(h_{10} | h_{11})) | h_3) \) in step (6).
CHAPTER 3
Protocol

In a system, WRITE and READ are the two main operations for client devices to access files from a cloud storage. This chapter is going to have an overview on the two operations with detailed protocols among client devices, the synchronization server and the cloud storage. CREAT and DELETE are the similar operations to them, so we are not going to introduce in this paper.

3.1 WRITE a file

![Diagram of Protocol to write a file]

The protocol has total 9 steps for a client device to complete the operation on writing a file to a cloud storage. It includes updating the file’s hash value on an FBHTree in cloud storage first and use a slice to audit that cloud storage did the updating correctly. Not until the end, client device will transfer the content of a file to the cloud storage, because the design can provide a better throughput for client devices accessing files together. In the second experiment of chapter 4, we will show that slice manipulation takes less time than transferring a file.

**Step 1:** The client device gets a Root Hash and a Global Sequence Number from the synchronization server. At the same time, synchronization server postpones
the request from next client through lock mechanism until step 8.

**Step 2:** The client device sends a request of writing a file to the cloud storage. The request includes operation type (OP), a *File Name*, a *Hash Value* from content of a file and a *Global Sequence Number*. File name and hash value are from the file that client device going to write, so client device can produce the hash value before sending the request. The request looks like below.

\[(OP \mid \text{File Name} \mid \text{Hash Value} \mid \text{Global Sequence Number}) \text{ Sign\_Client Device}\]

**Step 3:** The cloud storage uses the file name and index function \(\Gamma\) to find the corresponding leaf node ID and extract the slice of an FBHTree. We call it *Old Slice* here. This is the 1\(^{st}\) element to send back to client device in step 5.

**Step 4:** The cloud storage follows the algorithm of updating a slice in chapter 2.5 and gets a new *Root Hash* and a new *Global Sequence Number* from an FBHTree. These are the 2\(^{nd}\) and 3\(^{rd}\) elements to send back to client device in step 5.

**Step 5:** The cloud storage sends the three elements from step 3 and step 4 back to the client device. The message looks like below.

\[(\text{Old Slice} \mid \text{New Root Hash} \mid \text{New Global Sequence Number}) \text{ Sign\_Cloud Storage}\]

**Step 6/7:** The client device follows the procedure of auditing a slice described in chapter 2.7. He/she has to assure the operation that the cloud storage updated the file’s hash value correctly on a slice. If the auditing fails, clients have to ask the cloud storage to update the slice on FBHTree again.

**Step 8:** The client device sends the new *Root Hash* and new *Global Sequence Number* from step 5 to the synchronization server. At the same time, the synchronization server replies the next client’s request through recovering the lock mechanism.

**Step 9:** The client device starts to upload the content of file to the cloud storage.
3.2 READ a File

The protocol has 8 total steps for a client device to complete the operation on reading a file from the cloud storage. It includes retrieving a slice from the cloud storage and auditing the correctness. Not until the end, the client device retrieves the content of file from the cloud storage, because this design can provide a better throughput for client devices accessing files together. In the second experiment of chapter 4, we will show that slice manipulation takes less time than transferring a file.

**Step 1**: The client device collects the Root Hash and Sequence Number (SN) from the synchronization server. At the same time, synchronization server postpones the request from next client through lock mechanism until step 6.

**Step 2**: The client device sends a request of reading a file to the cloud storage. The request includes operation type (OP), Path Name and SN. The file name is from the file that client device wants to read.

\[(OP \mid \text{Path Name} \mid \text{SN}) \text{ Sign\_Client\_Device}\]

**Step 3**: The cloud storage uses the file name and index function \(\Gamma\) to find the corresponding leaf node ID and extract the slice from an FBHTree. After that,
cloud storage compares the SN with the one from synchronization server. If they are the same, increase the value by one to get SN’. The slice and SN’ will be returned to client device in step 4. Cloud storage also uses the path name to find the File that client device wants to read, which will be returned to client device in step 7.

**Step 4**: The cloud storage returns a slice and SN’ from step 3 to the client device. The package of message looks like below.

(Slice | SN’) Sign_Cloud Storage

**Step 5**: The client device follows the procedure of auditing a slice described in step 1 and step 2 of chapter 2.8. If the auditing fails, it means that the slice is incorrect. Clients have to ask cloud storage to send a slice again.

**Step 6**: The client device sends the new Root Hash and SN’ from step 4 to the synchronization server. At the same time, the synchronization server replies the next client’s request through recovering the lock mechanism.

**Step 7**: The cloud storage sends back the File from step 3 to the client device. The message looks like below.

(File) Sign_Cloud Storage

**Step 8**: The client device follows the operation on auditing a slice described in step 3 and step 4 of chapter 2.8. If the result is correct, client device can declare that the File is correct. If not, clients have to ask the cloud storage to send the file again or ask for reasonable compensation.
In the end of the chapter, we will show that the clients can always detect the wrong slice or wrong file retrieved from the cloud storage by the procedure of auditing. We use three different scenarios to demonstrate.

**Fig. 10. Different file and different slice**

First example in Fig. 10, a client device sends a request to the cloud storage for reading a file $f$ and expects to retrieve a slice($\Gamma(f)$) for auditing. However, he/she receives a different slice($\Gamma(f')$) from $f'$. The client can detect the error through the procedure described in step 1 of chapter 2.8. When the client derives a root hash for the slice($\Gamma(f')$) by the leaf node ID $\Gamma(f)$ and compare the one from the synchronization server, he/she will find the difference on the two root hashes. Because the left slice can derive a hash value for the right node in third level from its child nodes, the right slice can only derive a hash value for the left node in third level from its child nodes.

**Fig. 11. Same slice but different leaf node**
Second example in Fig. 11, a client device sends a request to the cloud storage for reading a file $f$ and expects to retrieve a slice($\Gamma(f)$) for auditing. However, he/she receives a different slice($\Gamma(f')$) from $f'$ and we can find that the two slices are the same from the picture. The client can still detect the error through the procedure described in chapter 2.8. Because the client can get the correct leaf node ID $\Gamma(f)$, he/she will swap the order of nodes in forth level to calculate their parent node when deriving the root hash. The procedure will make the root hash being different from the one from synchronization server.

![Diagram showing correct and wrong nodes]

**Fig. 12. Same leaf node but different file**

Third example in Fig. 12, a client device sends a request to the cloud storage for reading a file $f$ and expects to retrieve a slice($\Gamma(f)$) auditing. However, he/she get the same slice but different file. The client can detect the error through the pair value in the leaf node. Pair value is calculated from the pathname and the hash value from the content of a file, i.e. $\text{hash(pathname | hash(content of a file))}$. Client device can not find the pair value of $\text{hash(pathname of } f \text{ | hash(content of } f'\text{))}$ in pair list.
Chapter 4

Experiments and Results

We conduct a series of experiments to evaluate the performance of proposed Real-time POV scheme. The implementation details and experimental results are presented as follows. The system is implemented with Java programming language. The Digest function was “java.security.MessageDigest” with the “SHA-256” algorithm. RSA algorithm for the public-key encryption function was “java.security.KeyPairGenerator”. The cloud storage and client device are computers with Intel Core i5 3.10 GHz CPU, 6 GB memory space and Intel Core i5 1.4 GHz, 8 GB memory space, respectively.

<table>
<thead>
<tr>
<th>Height of FBHTree</th>
<th>Leaf nodes</th>
<th>File numbers</th>
<th>File stock ratio</th>
<th>#AVG collision</th>
<th>#MAX collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>256</td>
<td>85</td>
<td>33%</td>
<td>1.19</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>170</td>
<td>66%</td>
<td>1.42</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>100%</td>
<td>1.64</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1024</td>
<td>341</td>
<td>33%</td>
<td>1.17</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>682</td>
<td>66%</td>
<td>1.35</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1024</td>
<td>100%</td>
<td>1.61</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>4096</td>
<td>1365</td>
<td>33%</td>
<td>1.18</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2730</td>
<td>66%</td>
<td>1.37</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4096</td>
<td>100%</td>
<td>1.58</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>16384</td>
<td>5461</td>
<td>33%</td>
<td>1.17</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10922</td>
<td>66%</td>
<td>1.37</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16384</td>
<td>100%</td>
<td>1.59</td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>65536</td>
<td>21845</td>
<td>33%</td>
<td>1.17</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43690</td>
<td>66%</td>
<td>1.37</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65536</td>
<td>100%</td>
<td>1.58</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1. The collision behavior by index function

#AVG collision = Average number of hash value from files in a non-empty leaf node
#MAX collision = Maximum number of hash value from files in a non-empty leaf node

The first experiment was a research on the collision behavior by index function Γ. The source of files is collected over 20-year period in a disk. Randomly picked files from the source without duplicate are put into an FBHTree through Γ to observe the behavior. The height of FBHTree is tested from 9 to 17, and leaf nodes are from 256 to 65536,
respectively. These are the number of files usually in a file system. In table 1, we can see the result in column of #AVG and #MAX. The average/maximum number of pair values under file stock ratio from 33% to 100% are below 2 and 8.

<table>
<thead>
<tr>
<th>Height of FBHTree</th>
<th>( \rho )</th>
<th>( \sigma )</th>
<th>( \omega )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>0.15</td>
<td>0.106</td>
<td>0.304</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.16</td>
<td>0.080</td>
<td>0.304</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.20</td>
<td>0.080</td>
<td>0.284</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.14</td>
<td>0.100</td>
<td>0.364</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.22</td>
<td>0.084</td>
<td>0.304</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.20</td>
<td>0.080</td>
<td>0.364</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.20</td>
<td>0.104</td>
<td>0.344</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.26</td>
<td>0.080</td>
<td>0.340</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.25</td>
<td>0.080</td>
<td>0.324</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.24</td>
<td>0.140</td>
<td>0.408</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.22</td>
<td>0.060</td>
<td>0.384</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.26</td>
<td>0.060</td>
<td>0.340</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0.26</td>
<td>0.100</td>
<td>0.406</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.26</td>
<td>0.100</td>
<td>0.384</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.34</td>
<td>0.120</td>
<td>0.406</td>
</tr>
</tbody>
</table>

Table 2. The time required of slice manipulation

\( \rho \): The number of pair values in a leaf node
\( \sigma \): The time consumption of extracting a slice from an FBHTree (in ms)
\( \omega \): The time consumption of deriving the root hash from a slice (in ms)
\( \delta \): The time consumption of updating a slice (in ms)

The second experiment was an extended research to the first one. We can see that the length of pair lists could be different under various conditions in Table 1, so we would like to study the relation between manipulating a slice and the time required of it. Through the protocols of writing and reading files in chapter 3, we distinguish them into three actions: Extract a slice, Derive a slice and Update a slice. In order to simulate the different situation for experiment, we use slices with 1, 5 and 10 pair values on an FBHTree and from 9 to 17 levels in height. From the result in Table 2, we can see all the actions in different situation are below 1 millisecond to finish.
### Table 3. The size of Slice from different tree height and pair list length

\( \rho \): Number of pair value from files stored in a leaf node

The third experiment was to study the size of a slice sent from cloud storage to a client device, because the time required on transferring a slice is also a concerned on efficiency. Table 4 shows the memory usage on a slice from different combination, the height of an FBHTree and the number of pair values. We can see the size of slices are less than 2 Kb and between 0.608 Kb to 1.696 Kb. Under the circumstance of network nowadays, transferring a slice can be very fast.

<table>
<thead>
<tr>
<th>Height of FBHTree</th>
<th>( \rho )</th>
<th>Memory usage of a slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>0.608 kB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.864 kB</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.118 kB</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.736 kB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.992 kB</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.132 kB</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.864 kB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.120 kB</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.440 kB</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.992 kB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.248 kB</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.568 kB</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1.120 kB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.376 kB</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.696 kB</td>
</tr>
</tbody>
</table>

### Table 4. The memory usage of a Slice and an FBHTree

<table>
<thead>
<tr>
<th>Height of FBHTree</th>
<th>AVG Memory usage on a slice</th>
<th>Memory usage on an FBHTree</th>
<th>Rate (Slice / FBHTree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>0.649 kB</td>
<td>( \approx 32 ) kB</td>
<td>0.0200</td>
</tr>
<tr>
<td>11</td>
<td>0.775 kB</td>
<td>( \approx 131 ) kB</td>
<td>0.0060</td>
</tr>
<tr>
<td>13</td>
<td>0.901 kB</td>
<td>( \approx 524 ) kB</td>
<td>0.0020</td>
</tr>
<tr>
<td>15</td>
<td>1.029 kB</td>
<td>( \approx 2.1 ) MB</td>
<td>0.0005</td>
</tr>
<tr>
<td>17</td>
<td>1.157 kB</td>
<td>( \approx 8.4 ) MB</td>
<td>0.0001</td>
</tr>
</tbody>
</table>
The forth experiment was to study the relation of memory usage between a slice and an FBHTree, because the memory usage is usually one of the most important to a system. We implement the FBHTree from 9 to 17 levels in height respectively and 100% stock ratio of files. For example, we put 256 files into the FBHTree with 9 levels in height and study memory usage on a slice and the whole FBHTree. From the structure of an FBHTree described in chapter 2.3, we know that a node takes 32 bytes to store a hash value, a pointer takes 8 bytes and a pair value takes 2 hash values to store the information from a file. Taking the first row in Table. 3 as an example, a slice takes average 0.649 kB and an FBHTree take 32kB in memory from the situation of 9 levels in height. By comparison, a slice is only 2% to the whole FBHTree. As the height increases, we can even see the rate decreasing down to 0.01%. Because there are only two more nodes in a slice but an FBHTree is double when the height increases one. The result also supports second experiment that a slice from different height of an FBHTree can have similar time required on manipulation.

The following four experiments are to measure turnaround time on file reading and writing operation, which is exactly the performance of Real-time POV scheme proposed in chapter 3. We implement the FBHTree with 15 and 17 levels in height, which refers to POV-L15 and POV-L17 in experiments. Because we believe the FBHTree with 17 levels usually can hold the most number of files, i.e. lower than 65536 files. To determine the feasibility, we use the transaction time from pure file transfer as baseline, which refers to Non-POV in experiments. The whole actions on pure file transfer includes a file transferring from one peer to the other and checking hash value of the file. We use rate to show the difference on the two schemes, i.e. POV-L15 / Non-POV and POV-L17 / Non-POV. For the size of files, we use 10kB, 100kB, 1MB and 10 MB to study the difference on performance.

We implement the system over different network environment: same network segment and different network segment. Same network segment means that there is no router between a client device and cloud storage. Different network segment means there are 6 routers between a client device and cloud storage. This design is to approach the environment in a cloud storage.
The fifth experiment was to study the time required on reading a file by Real-time POV scheme and Non-POV scheme, which are implemented in the same network segment.

<table>
<thead>
<tr>
<th>READ operation</th>
<th>10kB</th>
<th>100kB</th>
<th>1MB</th>
<th>10MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-POV</td>
<td>15.5</td>
<td>18.9</td>
<td>116.8</td>
<td>999.3</td>
</tr>
<tr>
<td>POV-L15</td>
<td>23.4</td>
<td>28.9</td>
<td>124.7</td>
<td>1009.2</td>
</tr>
<tr>
<td>POV-L17</td>
<td>24.8</td>
<td>31.2</td>
<td>126.8</td>
<td>1010.3</td>
</tr>
<tr>
<td>POV-L15 / Non-POV</td>
<td>1.50</td>
<td>1.53</td>
<td>1.06</td>
<td>1.01</td>
</tr>
<tr>
<td>POV-L17 / Non-POV</td>
<td>1.59</td>
<td>1.65</td>
<td>1.08</td>
<td>1.01</td>
</tr>
</tbody>
</table>

**Table. 3. Comparison the time required between Non-POV Scheme and Real-time POV Scheme on reading a file in the same network segment (in ms).**

![Bar charts](image)

**Fig. 13. Charts to show the relation in Table 5.**

In the Table 5 and Fig. 12, we can see the time required of reading a file from Real-time scheme and Non-POV scheme. For different height of an FBHTree, POV-L15 and POV-L17 have similar time required on a transaction. It shows the scalability on our scheme.
The sixth experiment tested the time required on writing a file with Real-time POV scheme and Non-POV scheme, which are implemented in the same network segment.

<table>
<thead>
<tr>
<th>WRITE operation</th>
<th>10kB</th>
<th>100kB</th>
<th>1MB</th>
<th>10MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-POV</td>
<td>13.2</td>
<td>18.0</td>
<td>108.9</td>
<td>977.9</td>
</tr>
<tr>
<td>POV-L15</td>
<td>24.6</td>
<td>28.2</td>
<td>127.3</td>
<td>1058.7</td>
</tr>
<tr>
<td>POV-L17</td>
<td>24.3</td>
<td>30.1</td>
<td>127.5</td>
<td>1061.7</td>
</tr>
<tr>
<td>POV-L15/Non-POV</td>
<td>1.87</td>
<td>1.56</td>
<td>1.17</td>
<td>1.08</td>
</tr>
<tr>
<td>POV-L17/Non-POV</td>
<td>1.84</td>
<td>1.67</td>
<td>1.17</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Table. 4. Comparison time required between Non-POV Scheme and Real-time POV Scheme on writing a file in the same network segment (in ms).

![Charts to show the relation in Table 6.](image)

In the table 6, we can see the time required of writing a file from Real-time scheme and Non-POV scheme. For different height of FBHTree, POV-L15 and POV-L17 have similar transaction time. It shows the scalability on our scheme. For comparing different schemes, Fig 13 shows the rate from 1.87 to 1.08. We can see that large files are able to lower difference.
The seventh experiment was to study the time required on reading a file with Real-time POV scheme and Non-POV scheme, which are implemented in different network segment with six routers among them.

<table>
<thead>
<tr>
<th>READ operation</th>
<th>10kB</th>
<th>100kB</th>
<th>1MB</th>
<th>10MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-POV</td>
<td>24.6</td>
<td>42.0</td>
<td>749.2</td>
<td>9747.0</td>
</tr>
<tr>
<td>POV-L15</td>
<td>30.0</td>
<td>49.2</td>
<td>827.0</td>
<td>9784.0</td>
</tr>
<tr>
<td>POV-L17</td>
<td>32.0</td>
<td>51.4</td>
<td>883.2</td>
<td>9882.0</td>
</tr>
<tr>
<td>POV-L15/Non-POV</td>
<td>1.22</td>
<td>1.17</td>
<td>1.10</td>
<td>1.01</td>
</tr>
<tr>
<td>POV-L17/Non-POV</td>
<td>1.30</td>
<td>1.22</td>
<td>1.18</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Table. 7. Comparison the time required between Non-POV Scheme and Real-time POV Scheme on reading a file in different network segment (in ms).

In the Table. 7 and Fig. 14, we can see the time required of reading a file from Real-time scheme and Non-POV scheme. For different height of FBHTree, POV-L15 and POV-L17 have similar time required on a transaction. It shows the scalability on our scheme.
The eighth experiment was to study the time required on writing a file with Real-time POV scheme and Non-POV scheme, which are implemented in different network segment with six routers among them.

<table>
<thead>
<tr>
<th>WRITE operation</th>
<th>10kB</th>
<th>100kB</th>
<th>1MB</th>
<th>10MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-POV</td>
<td>16.8</td>
<td>27.2</td>
<td>792.4</td>
<td>9476.0</td>
</tr>
<tr>
<td>POV-L15</td>
<td>32.0</td>
<td>40.2</td>
<td>814.7</td>
<td>9483.5</td>
</tr>
<tr>
<td>POV-L17</td>
<td>36.5</td>
<td>40.8</td>
<td>844.6</td>
<td>9497.6</td>
</tr>
<tr>
<td>POV-L15/Non-POV</td>
<td>1.90</td>
<td>1.48</td>
<td>1.03</td>
<td>1.001</td>
</tr>
<tr>
<td>POV-L17/Non-POV</td>
<td>2.17</td>
<td>1.50</td>
<td>1.07</td>
<td>1.001</td>
</tr>
</tbody>
</table>

Table. 8. Comparison the time required between Non-POV Scheme and Real-time POV Scheme on writing a file in different network segment (in ms).

Fig. 16. Charts to show the relation in Table 8.

In the Table. 8 and Fig. 15, we can see the time required of reading a file from Real-time scheme and Non-POV scheme. For different height of FBHTree, POV-L15 and POV-L17 have similar time required on a transaction. It shows the scalability on our scheme.
Fig. 17. The rate of time consuming on $\alpha(=\text{POV-L15/Non-POV})$ and $\beta(=\text{POV-L17/Non-POV})$.

From Table. 5 to Table. 8, we can have a view that the time consuming on POV scheme are approaching to Non-POV scheme when transferring a file with bigger size. In Fig. 16, the four line graphs give this trend by the rate from comparing POV-L15 and POV-L17 with Non-POV. The large files can have almost similar performance to Non-POV scheme.

Chapter 5
Discussion and Future work

5.1 Discussion
This research is focused to develop an efficient scheme on achieving Real-time POV between client devices and cloud storage. The client devices can use the proofs to audit the correctness of files from the cloud storage. The cloud storage can also use the proofs to protect himself from mischievous accusation by client devices. To accomplish the objective, we propose a new structure of tree, FBHTree, to store the hash value of files and also a new flow for storing and auditing an FBHTree between the client devices and the cloud storage.

From result of the first and second experiment, it shows the height of an FBHTree can be determined proportional to the amount of files going to store in a cloud storage. That
is, whatever the amount of files is stored, the average number of pair values in each leaf node can be controlled below two. That is, the slice has an upper bound in length. Each file can have similar time required for manipulation. From the experiment, we can see the time is lower than 1 millisecond. Even the worst case with collision have the same result. Compare to earlier work [14], the method of using a hash tree has to calculate lots of hash values in each level and need to traverse many levels while deriving a root hash. For the aspect of structure, earlier work needs to use pointers to connect each node, which is a complex structure for manipulation. In this paper, FBHTree uses one-dimensional array for supporting direct access on each node.

From the result on third and forth experiment, the size of a slice is only around 2 kB. Clients can store an FBHTree on the cloud storage, and retrieve a slice when needed. Syncing three-tuple($\gamma$, SN, Sig), less than 1 kB, from a synchronization server, clients are able to have an audition for a file. Contrary to earlier work [14], storing a partial hash tree on each client device need to take several MB at using memory and time synchronizing with client devices.

From fifth to eighth experiments, they show the scheme outperforms earlier work [14] on performance by one to two order of magnitude. This scheme only needs 2.17 to 1.03 times higher than pure file transaction for read and write files from 100kB to 1MB, respectively.

5.2 Future work

During the transactions of reading and writing a file through network, it is hard to avoid them from stopping in the middle. If there were a client device does not correctly complete the transaction, the next client device will not be able to transmit his/her request to the cloud storage and got locked eternally. We have to implement error detection for this potential problem.
Chapter 6
Related work

Cloud storages are popular now, such as Dropbox, OneDrive and iCloud etc. However, none of them support the concept of POV to protect the stored files.

SiRiUs [7] provides the owner of files to be able to manage access control on the sharing files to other people in an untrusted network environment. The new structure of metadata from files can guarantee the owner grant/revoke the authority of files to other people and uses a hash tree to maintain the metadata of whole files in a file system. They use hash tree to accelerate users to verify the metadata and always read the fresh one. However, the scheme can not support users to assure the content of files is fresh or not.

SUNDR [8] provides the concept of fork consistency. Anyone who modified a file can be detected by the next person who is also going to write the file. They can support everyone to have the same view of manipulation on files. Their solution is to store the snapshots of file version on a remote server and uses a hash tree to order them. Nevertheless, they can not support users to assure the content of a file is fresh or not.

Iris [9] is a system that can audit the integrity and freshness of files stored on a cloud storage. It is an enterprise oriented architecture. They propose a portal residing between enterprise clients and a cloud storage. The portal is maintained by the enterprise itself. All the operations from clients have to go through portal, because it is in charge of caching all the hash values of files stored on cloud storage. When client wants to read a file from cloud, the portal also audits the file for him. However, we believe that this may cause a huge duty on the portal. In our paper, we let each client audit the file themselves. They do not have the problem on caching hash value of files. Iris also can not prove the result to others if they found the files do not satisfy audition.

CloudProof [13] proposes a system that can support epoch-based POV. The system can be divided into three parties: users, the data owner and the cloud storage. Users get an attestation at every time he retrieves a file from the cloud storage. The attestations are signed with digital signature on each side, so none of them can repudiate the attestations. Data owner accumulates the attestation and launch auditing in every period of time. They can detect the violation on Integrity, read serializability and read freshness.
However, each user has to broadcast the latest attestation to all the other users to assure them can correctly finish the auditing.

**Chapter 7**

**Conclusion**

It is important to have a scheme that can audit consistency of files stored on a cloud storage. The clients and service providers should both agree on the result of audition and obey to the agreement on SLAs. However, there is no existing mechanism supported by the existing service providers. Clients may lose their protection on files. To solve the problem, we propose a light duty and efficient way to achieve. Clients can use it to check the violation on files and prove the violation to service providers at any time. We call it as Real-time Prove of Violation.

The scheme consists of three characters to interact with each other, a cloud storage, client devices and a synchronization server. Cloud storage stores the contents and the hash values of files in a remote server. The hash values are produced by a hash function based on the cryptography, such as SHA256, to keep the integrity of files. To raise the convenience of accessing hash values for client devices, we propose a new hash tree to build them up. Every time a client device uploads a file also simultaneously updates the tree. The tree will return two short values for representing the update of files in an account. The client device will cache the values in a self maintained synchronization server for the purpose of sharing and concurrency control with other client devices. Once a client device would like to access and audit a file, the device can use the values to prove the correctness of file and show the result to service providers while a violation happens.

Comparing the time consumption of new scheme with the method widely used in auditing a remote file, i.e. time of transferring a file and auditing the checksum. The new scheme only takes average 26% and 1% more time to hold the function to audit cloud storage's violation to a 10kB and 10Mb file respectfully. Furthermore, client devices will not need to cache any information of files but merely rely on the cloud storage and synchronization service to safely store all the files.
Chapter 8

Reference
