Reproduction of Cloud Object Storage Synchronization

Abstract

Abstract. Cloud storage synchronization is a critical and useful data manage application for cloud storage. It can make data keeps the same copy both in local and cloud. However, existing schemes and systems are in closed form. It makes users hard to transfer data from one cloud storage service to another. To build this gap, an open-sourced cloud storage synchronization system is proposed. It can work on most mainstream cloud storage service providers, however, it causes relatively large extra storage cost. To solve this problem, this work optimizes the system, reduce the extra storage cost.

Keywords: cloud object storage, Synchronization, push-pull paradigm, extra storage cost.

1 Introduction

Nowadays, Cloud computing has become more and more popular and has been widely used in many fields. Individual users use cloud storage to storage their data for more convenience and lower cost. Enterprises also choose using cloud to manage their data and services. Users always want to maintain their data with the same copy both in local and the cloud, and cloud storage synchronization is a useful and critical solution for this problem. Considering following two application scenarios. The first one is about enterprise users. An enterprise stores data both in local and on cloud. The data in the cloud provides normal customer data access while the data in local serves as a backup. When the cloud breaks down, the local data can keep the services usable. In this scenario, the data in local and cloud should keep the same copy. The second scenario is about individual users. Yamashita outsource his data to the cloud, meanwhile, he has multiple devices (e.g., office computer, PC, mobile phone). He want to keep the same copy both in local devices and the cloud. In this case, cloud storage synchronization can make it. Researchers and cloud service providers has provided several cloud storage synchronization schemes or systems. However, existing schemes and systems are closed-sourced or can't work efficiently. To build this gap, paper [1] proposes a usable, open-sourced, efficient cloud storage synchronization system. The proposed system can work on most mainstream cloud object storage platform (e.g., Microsoft OneDrive, Ali Cloud, Tencent Cloud). The proposed system can also work on multiple local devices. Users only need to install the program in local devices and run, the data can be automatically synchronized. The proposed system uses tree structure to show the file system in local and cloud. Both local and cloud involves two trees: current tree and history tree. The current tree stores the current meta information of files and directories while the history tree stores the state after the latest round of synchronization. The system synchronizes the data in local and cloud by comparing the current tree and history tree.

Because the history tree has to be stored locally, it makes extra storage cost. This work reproduces the cloud storage synchronization system proposed in [1] and optimizes the proposed system. We revise the tree structure to reduce the total size of single tree and reduce the total extra storage cost. Our contributions are as follow:

- Optimize the tree structure, reduce the size of single tree.
- Reduce the total extra storage cost of the system.

2 Related works

Storage synchronization is a classical topic in computer science. It has been extensively studied by researchers. Existing research on this topic can be divided into two parts: delta synchronization and full synchronization. The main idea of delta synchronization is only to synchronize the changed parts of a file without synchronizing the whole file. It saves network resources by needing more computation resources. The most typical and influential solution in delta synchronization is the rsync application [2].

While the delta synchronization is only to synchronize the changed parts of a file, full synchronization is to synchronize the whole file. It requires more network resources and less computation resource than delta synchronization. The typical application of full synchronization is PandaSync [3]. Researchers found that full synchronization has better performance comparing to the delta synchronization.

3 Method

In this section, we first show the system model of proposed system. Then, wo give the high level idea of the system. Finally, we show the details of system design and show our optimization of the system.

3.1 System Modeling

Fig. 1 shows the architecture of the proposed system. The cloud storage synchronization system contains two entities: cloud and user. The cloud can be the cloud object storage provided by most mainstream cloud service providers. The user may have multiple local devices. The user's data is organized as a directory contains multiple files and sub-directories.

The system runs as follows: First, user set up the cloud storage synchronization system in his/her local devices. Second, Once set up finishes, the user runs the system, the system will automatically synchronize data between cloud and local. The system keeps detecting the data changes both in local and cloud, and synchronization the change files or directories at rounds.

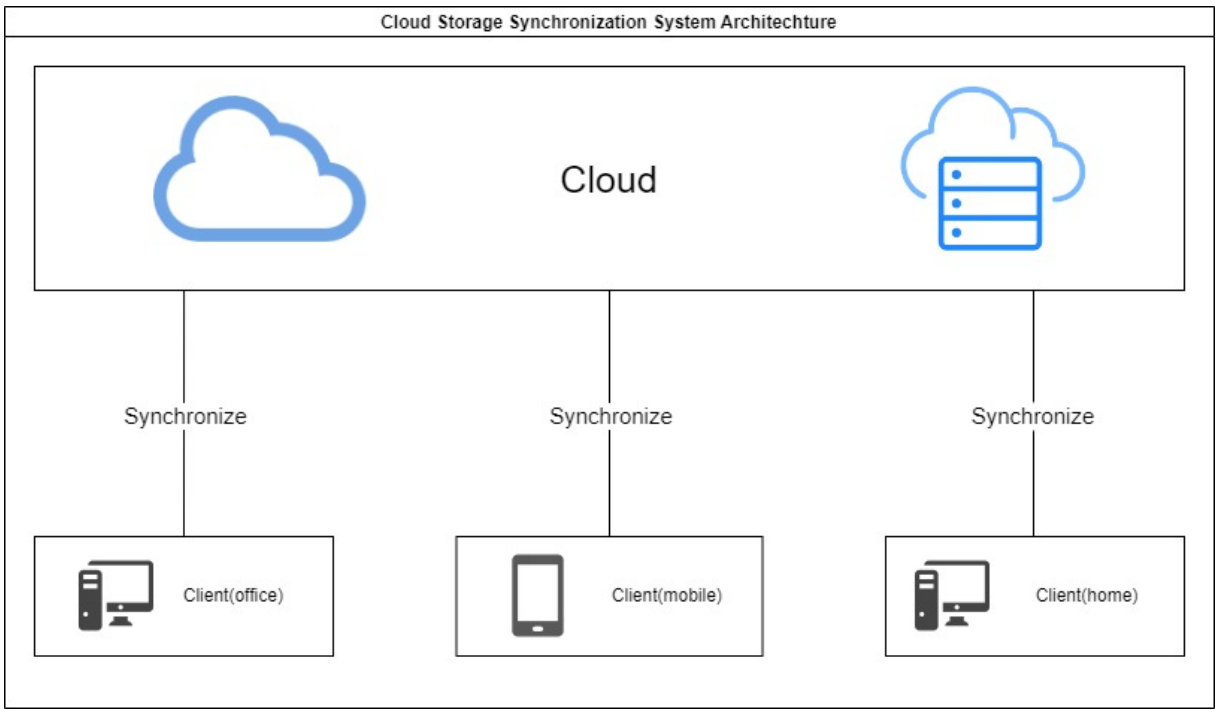

Fig. 1. Cloud storage synchronization system architecture

3.2 High Level Idea

The proposed system uses tree structure to store the meta information of files and directories. The meta information should include name and modification time. The system uses four trees: two to store the current state and history state of local and other two to store the current state and history state of cloud.

If the cloud synchronization system first runs, the history states are set as empty. The system builds the current states for both local and cloud. Then, the system synchronizes the local data into the cloud, which is called PUSH. After PUSH process, the system synchronizes the cloud data into the local, which is called PULL. This finished one round of synchronization. After that, the system updates the history states and stores them in local. After the first round, the system will synchronize the data regularly and automatically.

3.3 System Design

```
struct FileStatus {
    string name
    int    mtime
    string id
}
```

```
struct DirectoryStatus {
    string name
    int    mtime
    string id
    list   children[]
}
```

Fig. 2. Tree data structure

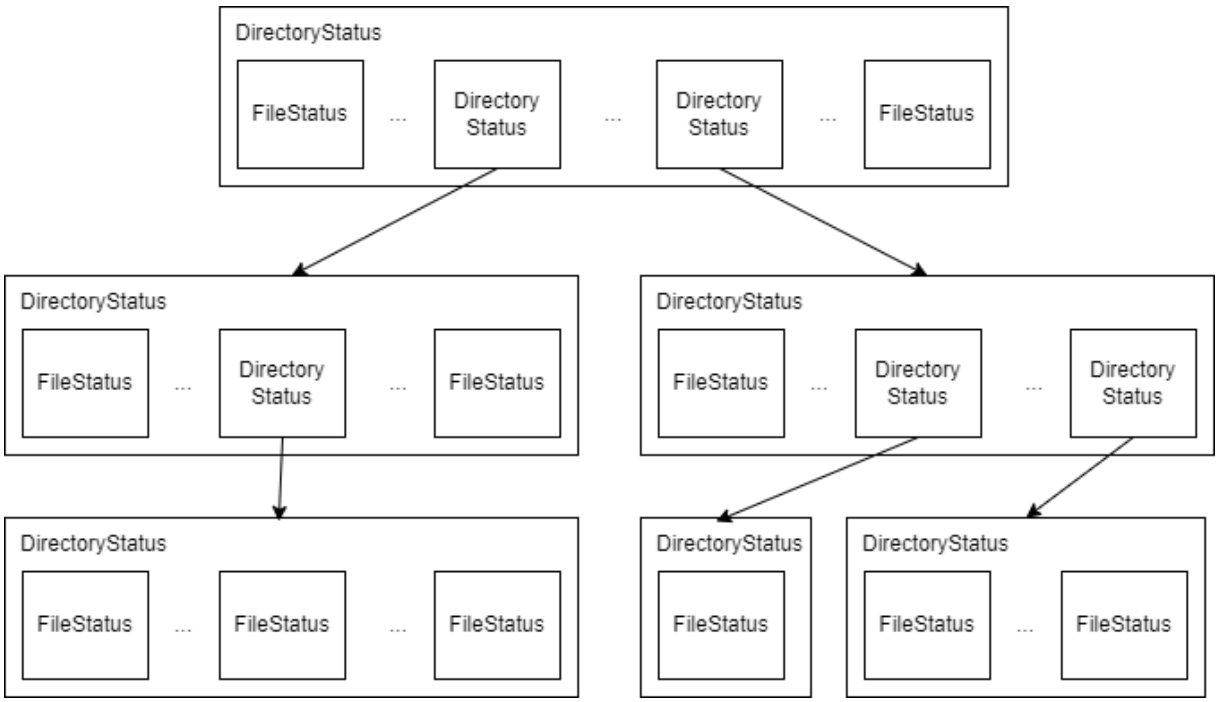

Fig. 3. Tree structure illustration

The proposed system uses tree structure to store the state both in local and cloud. The tree node structures are shown as Fig. 2. The node for files contains name, mtime and id. The name is the name of files, the mtime stores the modification time of the files, and the id is the hash value of the files. The node for directories is similar to the node for files. The name in it is the name of directories and the id is set as empty. The node for directories has list to store the state of sub-files and sub-directories. Fig. 3 shows the tree.

The system first start initialization. During initialization, the system get the history trees from local. Then the system start synchronization. In one round of synchronization, the system first run Build Tree algorithm to build cloud current tree and local current tree. Then the system run PULL algorithm to synchronize the cloud data into the local. Finally, the system run PUSH algorithm to synchronize the local data into the cloud. This finishes one round of synchronization. After each round of synchronization, the system update the history trees and save them locally. Fig. 4 shows the process of the system.

The BuildTree algorithm runs as follows: First, the system initializes a DirectoryStatus structure using the path of the root directory. Then, traverse the files and sub-directories within the directory and insert the states into list. For files, a FileStatus will be initialized and insert into the list. For sub-directories, the system will recursively run the BuildTree algorithm to get the DirectoryStatus of each sub-directories and insert into the list.

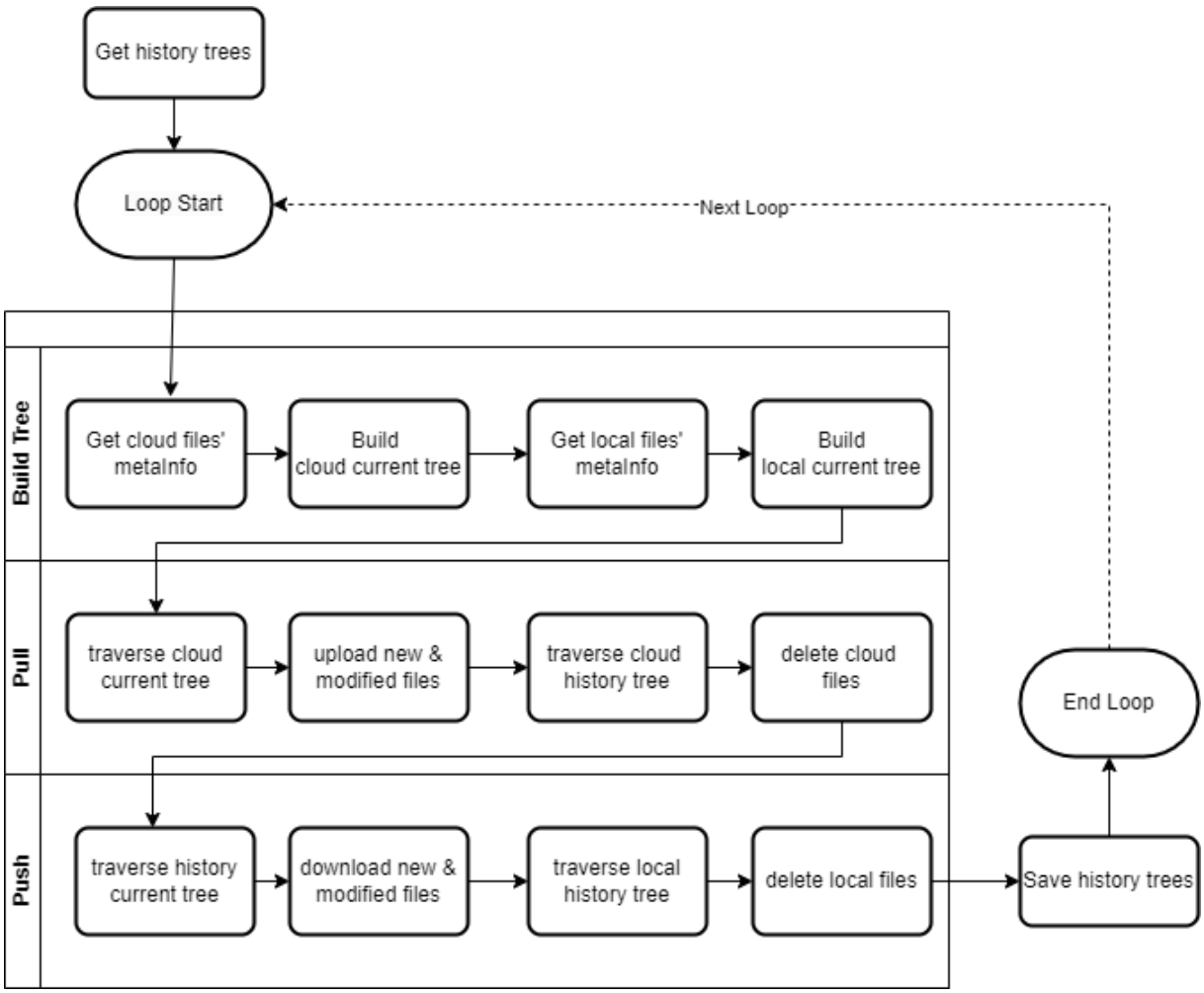

Fig. 4. System illustration

The PULL algorithm runs as follows: First, traverse the cloud current tree, and compare with the local history tree. If there is a directory or file has been changed, then it will be downloaded to local. Second, traverse the cloud history tree and compare with the local current tree. If there is a directory or file exists in cloud history tree but not in local current tree, then delete it.

The PUSH algorithm runs as follows: First, traverse the local current tree, and compare with the cloud history tree. If there is a directory or file has been changed, then it will be uploaded to cloud. Second, traverse the local history tree and compare with the cloud current tree. If there is a directory or file exists in local history tree but not in cloud current tree, then delete it.

About the upload/download and rename operations of files. The id can be used for fast operation. When upload/download operation occurs, the system can find if there is an existed same file by check the value of id. If found, the system will copy it to destination path. The rename operation is similar. The system can check if the file is renamed by checking the value of id. If the id are the same, the system can use open API provided by cloud to rename the file in the cloud or use system call to rename the file locally.

3.4 Optimization

After each round of synchronization, the system will update the history trees and save them locally. This causes extra storage cost. What's more, the system uses absolute path as the name of files and directories. When the file system is complex, the meta tree will also be complex and cost significant size of storage space. Therefore, we optimize the value of name. We only use the file name and directory name as the value of the name in FileStatus and DirectoryStatus respectively. In this case, the size of meta tree can be reduced. What's more, the local history tree and cloud history tree are the same, therefore, the number of history tree can be one. Thus, the extra storage cost can be reduced quite a lot.

4 Implementation details

4.1 Comparing with the released source codes

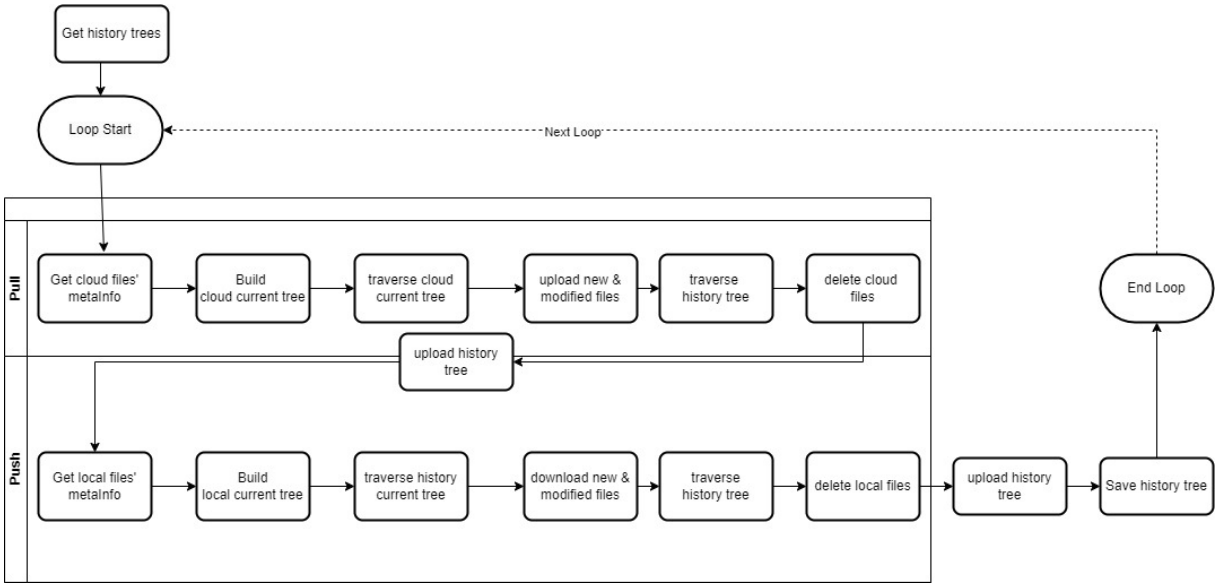

Fig. 5. Optimized System

As shown in section 3.4, we only use the original name of file and directory as the name in FileStatus and DirectoryStatus structure relatively, and we only use one history tree. Thus, the extra storage cost is reduced quite large. Because the name in FileStatus and DirectoryStatus is changed and the number of history tree is one, the process of the system should be changed. Fig. 5 shows the process of system after optimization. We divide BuildTree algorithm and build cloud current tree at the beginning of PULL algorithm and build local current tree at the beginning of PUSH algorithm relatively. In each round of synchronization, after PULL algorithm is finished, the history should be updated, make it equal to cloud current tree. After each round of synchronization, the history tree also needs to be updated, make it equal to cloud local tree and save it to local.

We use GO language for implementation, while the released code is implemented by PYTHON. We use the open GO SDK provided by the cloud service provider.

4.2 Experimental environment setup

We implement the optimized system by using GO 1.21.4. We use the cloud object storage provided by Tencent Cloud. The program runs on the PC with Intel i5 9500 CPU, 16GB memory, and 1TB hard disk. The PC is located in Shenzhen University with 10Mbps outbound network connection. In the implementation, we use SHA-256 algorithm as hash function to get hash value. For Tencent cloud object storage service, we set the storage geographical region as 'Guangzhou'. The COS service is the original course, without any acceleration service as CDN.

To evaluate the system performance, we mainly consider the storage cost, communication cost and computation cost. For storage cost, we measure the storage cost of meta tree. For communication cost, we measure the number of HTTP request caused by each operation. For computation cost, we measure the time cost.

For experiment we randomly generate 100, 1000, 10000 txt files relatively, each file is 1KB and name by its index in each set. For storage cost and time cost of build tree, we measure the value by using the sets relatively. For communication cost and time cost of each operation, we use the 1000 txt files and operate 100 files to measures the number of HTTP request of each operation.

4.3 Main contributions

Our main contributions are as follows:

- Change the value type of the name in tree node structure, reduce the number of history tree and the size of each tree. Reduce extra storage cost.
- Use GO language for implementation.

5 Results and analysis

5.1 Storage Cost

TABLE 1. Storage Cost

Number of files	100	1,000	10,000
local meta tree size (bytes)	9,252	89,456	90,0457
cloud meta tree size(bytes)	9,252	89,457	90,0457

TABLE 2. Original Storage Cost

Number of files	100	1,000	1,0000
local meta tree size (bytes)	15,640	160,540	1,609,558
cloud meta tree size(bytes)	15,639	160,539	1,609,557

TABLE 1 is the storage cost of optimized system, Fig. 6 is the storage cost of the original system. The size of history tree is the same as current tree, therefore, we measure the size of local meta tree and cloud meta tree. The result shows that the optimized system has much better performance in storage cost. The size of each tree under different test sets of optimized system is almost the half of the original system. Thus, the optimized system reduced quantity of extra storage cost.

5.2 Computation Cost and Communication Cost

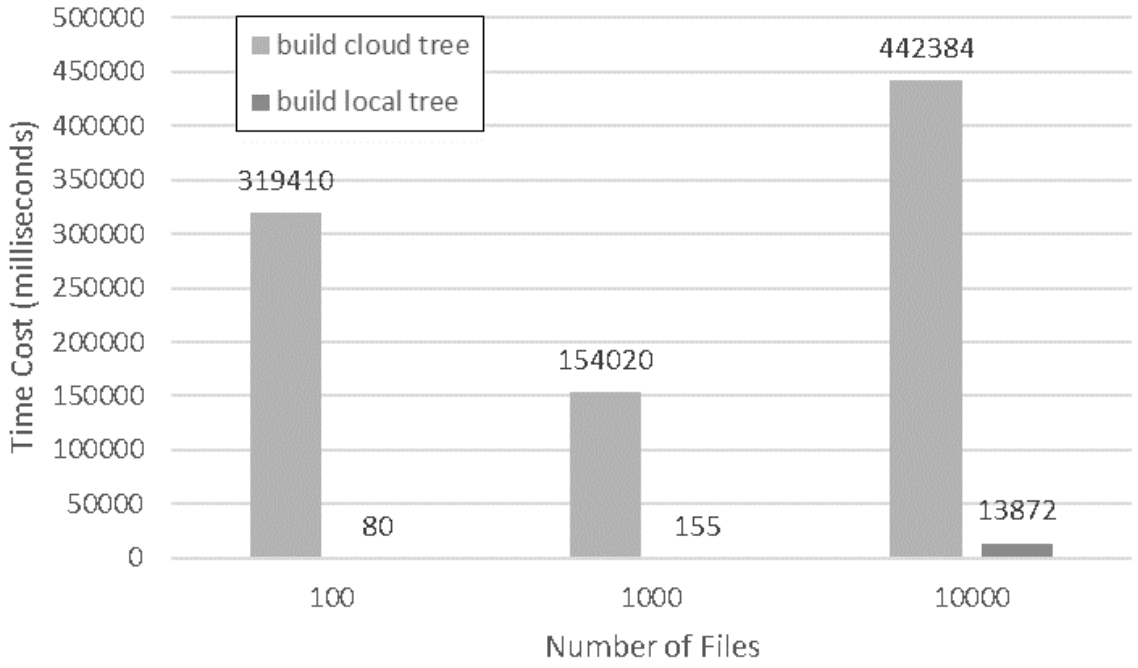

Fig. 6. BuildTree computation cost

Fig. 7 shows the computation cost of BuildTree. According to the results, we can find build local tree is negligible, even there is 10000 files in local, it only cost 13872ms to build the local tree. However, the time cost of building cloud tree is quite large. The reason is that building the cloud tree need to call the cloud API, and it significantly influenced by network conditions. Due to the bad network condition, the time cost of building cloud tree is quite large and unstable.

Fig. 8 shows the computation of operations in PUSH and PULL. Time cost of PULL in create, modify and delete operation is more than PUSH. This is because the network conditions is not stable and the download API is more complex than upload API in the GO SDK provided by Tencent Cloud. For the rename operation, it needs to call API in PUSH process, so it causes more time cost.

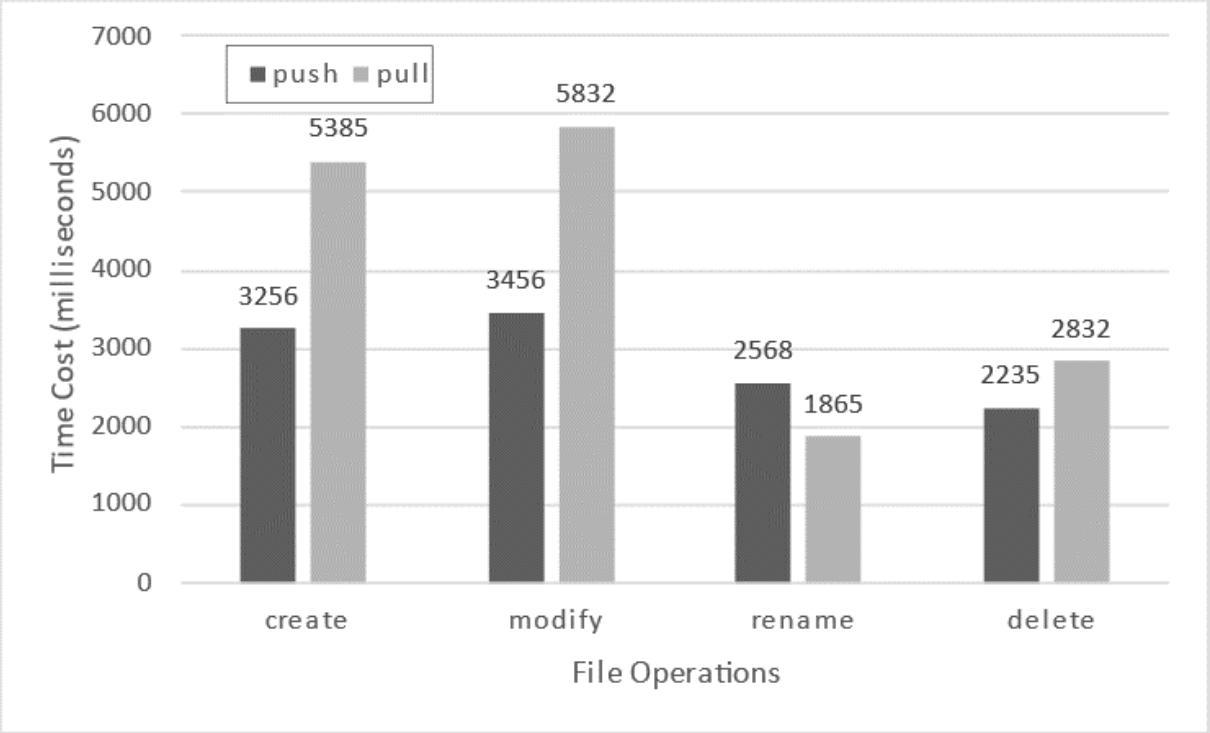

Fig. 7. PUSH/PULL computation cost

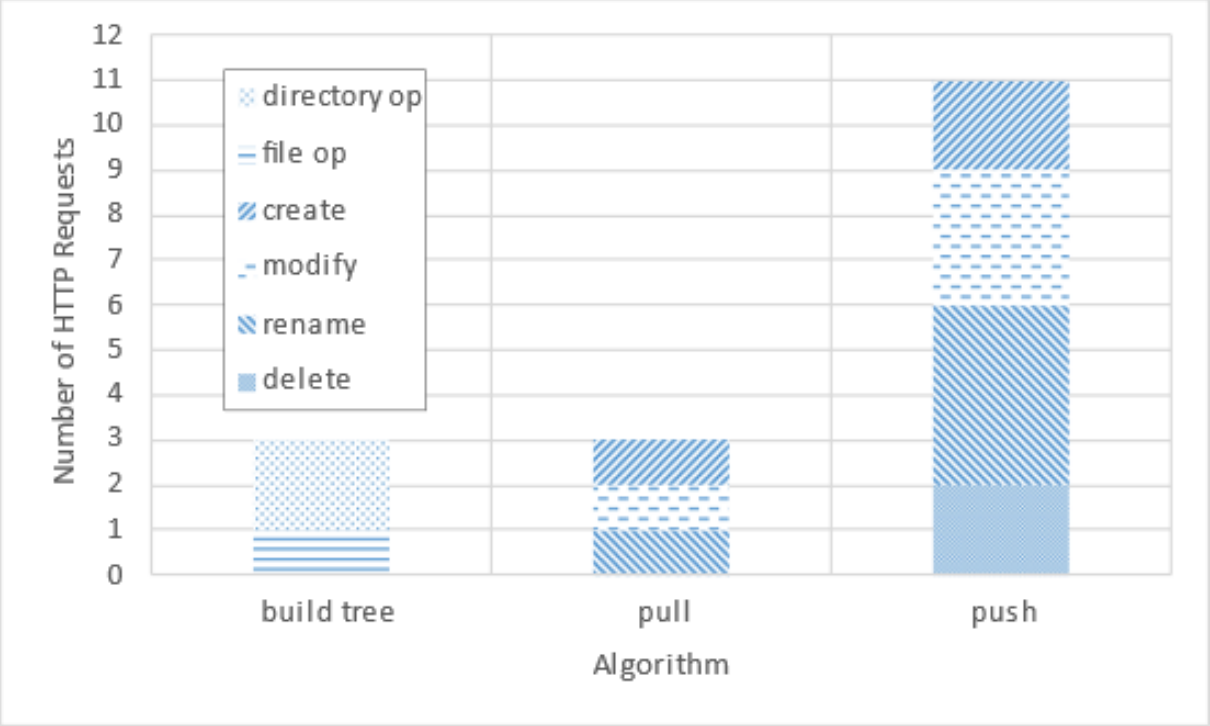

Fig. 8. Communication Cost

Fig. 9 shows the communication cost of the optimized system. In BuildTree process, file operation causes 1 HTTP request, directory operation causes 2 HTTP request because the system need to identify it is a directory. In PULL process, the rename, create and modify operations only need to call API once, so each operation only causes 1 HTTP request. In PUSH process, create and delete operation cause

2 HTTP requests, rename operation causes 4 HTTP requests and modify operation causes 3 HTTP requests. That's because rename need to check hash value, copy and delete, and modify also need to check hash value and need to create.

6 Conclusion and future work

This work optimizes the system proposed in [1]. We reduce the size of each meta tree and the number of history tree, achieving the goal reducing extra storage cost. For the future work, we consider if the delta synchronization can use in the system, because local computation resources cost less than communication cost. It is interesting to further investigate such whether the delta synchronization can improve the performance of the system.

References

[1] Fei Chen, Zhipeng Li, Changkun Jiang, Tao Xiang, and Yuanyuan Yang. Cloud Object Storage Synchronization: Design, Analysis, and Implementation. IEEE Transactions on Parallel and Distributed Systems, 33(12):4295–4310, December 2022.

[2] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[3] Suzhen Wu, Longquan Liu, Hong Jiang, Hao Che, and Bo Mao. PandaSync: Network and Workload Aware Hybrid Cloud Sync Optimization. In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pages 282–292, Dallas, TX, USA, July 2019. IEEE.