SyncCS: A Cloud Storage Based File Synchronization Approach

Chao Liang and Luokai Hu
Lenovo Mobile Communication Technology Ltd., Xiamen, P. R. China
Email: liangchao@lenovo.com, luokaihu@gmail.com

Zhou Lei and Jushu Wang
School of Computer Engineering and Science, Shanghai University, Shanghai, P. R. China
Email: {leiz, wangjushu}@shu.edu.cn

Abstract—**More and more people begin to own multiple computing devices. File synchronization technologies are needed to effectively manage data which spans multiple devices. However there is not a general standard for file synchronization system. In this paper we propose a file synchronization model based on cloud storage - SyncCS. We describe the SyncCS architecture and present a novel two-stage file synchronization protocol along with a conflict resolution mechanism. We evaluate the performance of our proposed file synchronization protocol. Our experimental results indicate that the number of operations to be synchronized when using our protocol is relatively smaller than that using a widely used method.**

Index Terms—**file synchronization model, synchronization protocol, conflict resolution**

I. Introduction

In the 1990s, the rapid development of network technologies, especially wireless networking technology, promoted the emergence of a new computing paradigm - mobile computing. Mobile computing has revolutionized the way we use computers [1]. More and more individuals own multiple computing devices. David Dearman and Jeffrey S. Pierce [2] conducted a survey on the use of people on multiple computing devices. They interviewed 27 people and found that people use multiple computing devices (including fixed equipment and mobile devices, such as desktop computers, notebook computers, personal digital assistants (PDAs), smart phones, etc.) for several reasons. People's daily activities may span multiple devices, or they may want to assign different roles to different devices depending on different environments, or they may want to separate their work activities from private life. The use of multiple devices creates a very important issue, that is, cross-device data management.

Manuscript received July 1, 2013; revised Nov 20, 2013; accepted Dec 11, 2013.
Corresponding author: Luokai Hu luokaihu@gmail.com.

Consider the following scenario. A person has a desktop computer in the office, a desktop computer at home, a personal laptop and a PDA. How to ensure that he can have consistent access to some important information on each computing device, such as address book, agenda, or meeting presentations? In the past, a commonly used method is using removable storage devices (such as U-disk and removable hard disk) or e-mail attachments to send smaller files. Some very important documents may be unsafe in removable storage devices, because the device may get lost or damaged. A fatal drawback of using e-mail attachments is that there is often a size limit on an attachment. The emergence of cloud computing [3] in recent years provides us with a better solution. That is, we can use cloud storage services provided by third party service providers as an intermediary to synchronize data or files between different devices of the same user. There have been some third-party services on the market. For instance, an online address book service *Plaxo*, [4], Firefox synchronous extension *Google Browser Sync* [5], and file synchronization services such as *Dropbox* [6], *Everbox* [7], *Kuaipan* [8] and so on. However, these companies often provide their products without implementation details. In this paper, we propose a file synchronization model based on cloud storage - SyncCS.

Data synchronization has been used in failure recovery of storage systems to improve system availability for a very long time. In such scenarios, we typically use incremental synchronization methods at data item or block level. However, when it comes to cross-device data management, the logical structure of a file cannot be ignored. In this paper, we focus on the issues pertinent to file-level data synchronization. In our file synchronization model, we use the concept of shared directory, like iFolder [9]. Each user sets up a shared directory When the user logs in at any client, the shared directory on the client will synchronize with the corresponding directory on the remote server. After synchronization, the shared directory on the client will be consistent with the corresponding directory on the server from structure to the contents of its files.

Since wireless coverage for mobile communications is often limited, wireless Internet is often accessible only in some fixed hot spot areas. Thus users cannot guarantee a connection with the network all the time. As a result, off-line operations on files are inevitable. In view of this situation, this paper presents a two-stage file synchronization protocol. The first stage is a non-real time two-way synchronization stage between the client and server, in which all update operations on the client and the server will be synchronized to each other. The second stage is a real time one-way synchronization stage between the client and server, in which the online update operations on the client will be synchronized to the server instantaneously. Since it supports offline operations, in the first stage it not only should synchronize the updates on the server to the client, but also the offline updates on the client to the server. Then we need a conflict mechanism to deal with file conflicts we may face in the two-way synchronization process. Inspired by the version control method in [10], our approach uses file-versions to detect and handle conflicts.

In this paper we address three major aspects of the file synchronization system, including system architecture, synchronization protocol, and conflict mechanism. The main contributions of this paper are: 1) we design a file synchronization model based on cloud storage. 2) We propose a two-stage file synchronization protocol. 3) Based on the proposed synchronization protocol, we design a conflict detection and resolution mechanism.

The rest of the paper is organized as follows. In Section 2 we give an overview of the whole system architecture. Section 3, we present the details of our proposed two-stage file synchronization protocol. Section 4 discusses our conflict mechanism alone. In Section 5, we experimentally evaluate the performance of SyncCS. Section 6 discusses the related work. Section 7 provides concludes this paper.

II. SyncCS Overview

In this section, we will briefly introduce the synchronization procedure and the system architecture of our file synchronization model - SyncCS

A. Synchronization Procedure

In the previous section, we assumed the following scenario: A user has multiple computing devices, including a desktop in the office, a desktop at home, a personal laptop and a PDA. He may use different devices in different occasions. However, there are some commonly used files that he wishes to access at any time and any place, such as the address book, agenda, photos, work documents and so on. For such a scenario, we propose SyncCS, a common file-level synchronization system model, which can synchronize the specified files on different devices of a user. We use the concept of shared directory, indicating that a user can specify a directory as a sync folder, and put the files that need to be synchronized into the sync folder, and then all update operations (including delete, modify and rename) on these files will be automatically synchronized to the server and to all devices of the user. We will introduce the synchronization process of SyncCS from the user's perspective as follows:

a) A user logs in at a SyncCS client (refer to this as the work client), and specify or create a local sync folder (below we will call it SyncCS-folder), then all the contents of this folder will be uploaded to the server. Thus, the work client and server come to a consistent state for the first time. Then the online updates of the work client will be uploaded instantaneously to the server until it is disconnected.

b) The user logs in at another SyncCS client (assume the laptop client). After logging in, he should create a local SyncCS-folder. Then the laptop client and server will do the first stage synchronization, in which the laptop client will download all contents of the SyncCS-folder on the server to the local SyncCS-folder. At this point, the laptop client and the server achieve consistency for the first time. Then the laptop client and server do the second stage synchronization, during which update operations on local SyncCS-folder on the laptop client will be monitored and uploaded to the server. Other clients will synchronize with the server in the same way when they first log in.

The user can update the contents of the local SyncCS-folder on any client when they are disconnected from the server. When he logs in again at a client (suppose the work client), there exists update operations to be synchronized both on the work client and the server. So the work client and server do the two-way synchronization in the first stage. The work client will synchronize to the server all offline update operations which have occurred since the last disconnection, and the server will synchronize to the work client all the update operations that have occurred by its synchronization with other clients. After the work client and the server come to a consistent state, they will proceed to the second synchronization stage, i.e., the real time one-way synchronization.

B. The SyncCS Architecture

Now we will describe the SyncCS architecture from system's perspective.

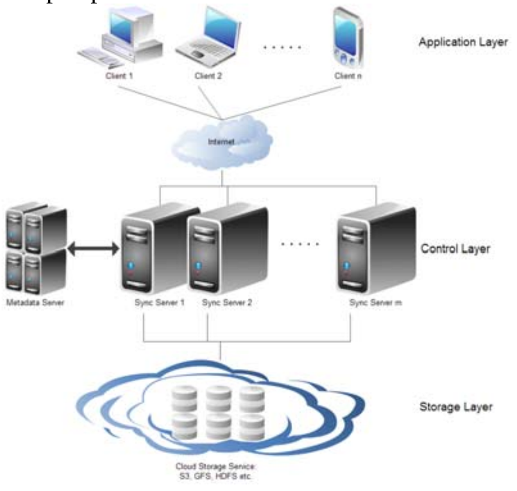

Figure 1. SyncCS Architecture

As shown in Figure 1, we divide the architecture of the SyncCS model into three layers: application layer, control layer and storage layer. The application layer includes different client software applications on different devices. These applications store the metadata of the files in the local SyncCS-folder, monitor users' update operations and record operation log, and synchronize with the SyncCS-folder by interacting with the server. The control layer includes a metadata server and some synchronization servers. The metadata server is used to store metadata information of the files in every SyncCS-folder on the server. While the synchronization server is responsible for handling client's synchronization requests, coordinating file conflicts occurred in the synchronization process, and interacting with the application layer and storage layer to synchronize files. The storage layer is a large-scale storage system, such as Amazon's S3 [11], GFS [12], HDFS [13] or other cloud storage services, and its responsibility is to store user files.

According to our synchronization protocol, as explained later in the next section, we will describe the information exchange process between the internal components of SyncCS as follows.

➢ After a user logs in at a client, the client will send a synchronization request to the Sync Server. After it receives the request, the Sync Server will respond to the client to initialize the synchronization.

➢ Then the client and the server will do the first stage synchronization. At this stage, we first upload, and then download.

➢ During the upload synchronization, the client will analyze and merge the operation log which has been recorded since the last synchronization, and send the processed log to the Sync Server. At the same time, the Sync Server will query the metadata server to get the metadata of the server-side SyncCS-folder. Then the Sync Server will conduct synchronization according to the operation entries recorded in the processed log. The modified file contents and metadata will be stored in the storage layer and the metadata server.

➢ Conflicts that are detected during the synchronization process will be stored in a temporary conflict table. After the upload synchronization completes, the Sync Server will interact with the client to coordinate the resolution of conflicts.

➢ During the download synchronization, the client will generate a view of the local SyncCS-folder and send it to the Sync Server. At the same time, the Sync Server will query the metadata server to generate a view of server-side SyncCS-folder. Then the Sync Server will compare these two views, generate an operation sequence based on the difference between them and send it to the client. After this, the client conducts download synchronization according to this operation sequence.

Finally, the client and the server will do the second stage synchronization which is real-time and one-way. At this stage, the client will monitor the user's update operations on SyncCS-folder and upload them to the Sync Server in real time. And the Sync Server will store the modified files into the storage layer, and at the same time update the metadata in the metadata server.

III. Two- stage Synchronization Protocol

We firstly introduce the organization of metadata. In our file-level synchronization protocol, each user file is associated with a metadata entry, which is defined as a seven-tuple like *(FID, User, Type, Path, Size, Time, Version)*. In this metadata tuple, *FID* indicates the unique identifier of a file which is specified by the server when it is synchronized to the server for the first time. It means that a local file does not have a *FID* or a corresponding metadata entry until it is synchronized to the server. *User* represents the user to whom a file belongs. *Type* indicates the type of a file. We consider directory as a special type of file and use *Type* to distinguish it from a normal file. *Path* indicates the full path of a file (SyncCS-folder as the root directory). *Size* is the size of a file. *Time* is the timestamp when the file is synchronized to the server. *Version* is the version number of a file, which is set to 1 when the file is first synchronized to the server. After this, the value of *Version* increments whenever the content of the file is modified on the server-side. Each client of each user maintains a copy of the current local metadata, while the metadata of all users' server-side SyncCS-folder is stored in the metadata server.

Suppose at some point (say T1), a client (say the work client) completes synchronization with the server and disconnects. Then the work client starts recording operation logs from T1. At point T2, the work client connects with the server again. Note that the server might have synchronized with other clients before T2. Below we describe the synchronization process between the work client and the server in detail in this case.

A. *Non-real-time Two-way Synchronization*

At this stage, we synchronize in the following order: upload synchronization, conflict resolution, and download synchronization. And we divide the update operations to be synchronized into four types: create, modify, rename, and delete. Each update operation corresponds to an operation entry, of which the format is like *(FID, Cmd, Para)*, where *FID* is the unique identifier of the updated file, *Cmd* is the operation code, and *Para* represents other parameters involved in the operation. Each operation and its corresponding entry are shown in Table 1.

TABLE I.
Operation Entrys

Operation Type	**Operation Entry**
Create	(FID, Create, Fpath)
Modify	(FID, Modify, Fversion)
Rename	(FID, Rename, OldPath, NewPath)
Delete	(FID, Delete, Fversion)

Once a user performed an update operation on the local SyncCS-folder, the client software will be able to detect it, and generate a corresponding entry. Operation log is a sequence of operation entries that is recorded by the client. To facilitate the subsequent log processing, the client will also record into the log the local time when an update operation occurs. So each entry in the operation log is recorded in a format like: *[LocalTime, (FID, Cmd, Para)]*, in which the *LocalTime* indicates the local time when an update operation occurs.

Upload Synchronization (client -> Server)

First, the work client analyzes and merges the operation log that is recorded from T1 to T2. There may be many redundant operations that are performed offline by a user on the local SyncCS-folder. For example, a file is revised many times. A file is created after T1 and deleted before T2 and so on. To synchronize these duplicate update operations will consume a large amount of time and network flow, and may even lead to some unnecessary conflicts. So it is better for the client to preprocess the operation log before synchronization, i.e., to merge the operation sequences in the log. We summarize several kinds of operation sequences that can be merged as shown in Figure 2. Note that the client can only merge the sequences that consist of operations on the same file.

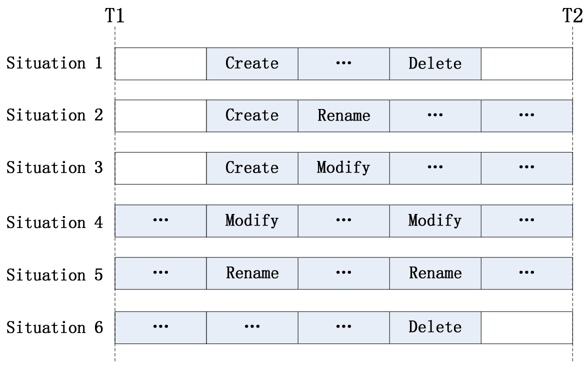

Figure 2. Operation Sequences to be merged

Figure 2 shows six kinds of situations. Each labeled square represents an update operation. The long grid composed by a number of small boxes represents a sequence of update operations on the same file. The blank small box represents a blank period, during which there is no update operation. The first long grid represents a case that a file has been through a process from creation to deletion during a period from T1 to T2 ('...' in a small box represents any number and any kind of update operations). In this case, we do not need to synchronize the file, because the file has completed its entire life cycle during the period from T1 to T2. The second and third ones describe cases that a file is created during the period from T1 to T2, and renamed or modified after the creation. In these cases, we just need to synchronize the create operation. The fourth and fifth ones describe cases that the same file is renamed or modified repeatedly during the period from T1 to T2. In these cases, we only need to synchronize once. The last one describes a case that a file is finally deleted after being through a variety of update operations in the period from T1 to T2. In this case, we only need to delete the file directly from the server.

The methods for processing the corresponding sequence in the log are different in different case. We call the log before processing L and the one after L'.

For Cases 1, 2 and 3, files are created after T1. They have never been synchronized to the server. Thus they have no metadata entries locally. To synchronize these files, the process of log merging would be too complicated. The easiest way is that the client software scans local SyncCS-folder and local metadata base, generates a create operation entry [T2, (NULL, Create, Fpath)] for each file that has no corresponding metadata entry, and inserts the operation entry into L'.

For cases 4, 5, 6, we need to merge the log sequence for the corresponding file. The merging process is described as follows:

a. Classify the log entries in L by *FID*. Then the entries corresponding to the same file will be gathered together.

b. Classify the log entries of the same file by *Cmd* and sort them in the order of *Delete*$\rightarrow$*Rename*$\rightarrow$*Modify*.

c. Order the update operation entries of the same kind in an increasing order of *LocalTime*.

d. For the log sequence of each *FID* (i.e., each file), perform the following steps:

e. If the first entry of the sequence is a *Delete* operation, then insert a *Delete* operation entry [LocalTime, (FID, Delete, Fversion)] into L’, go to step h.

f. If the first entries of the sequence are one or more *Rename* operations, such as: [LocalTime$_1$, (FID, Rename, OldPath$_1$, NewPath$_1$)] [LocalTime$_2$, (FID, Rename, OldPath$_2$, NewPath$_2$)] ……[LocalTime$_n$, (FID, Rename, OldPath$_n$, NewPath$_n$)], then insert a *Rename* operation entry [LocalTime$_1$, (FID, Rename, OldPath$_1$,NewPath$_n$)] into L’, and remove all of the rename operation entries of this FID from L.

g. If the first entries of the sequence are one or more *Modify* operations, such as: [LocalTime$_1$, (FID, Modify, Fversion)] [LocalTime$_2$, (FID, Modify, Fversion)] ……[LocalTime$_n$, (FID, Modify, Fversion)], then insert a *Modify* operation entry [LocalTime$_1$, (FID, Modify, Fversion)] into L'.

h. Repeat steps e to g for the log sequence of the rest FIDs.

i. Order the operation entries in L' in the increasing order of LocalTime.

After log processing is completed, the work client will send the processed log L’ to the server. Then the server can do upload synchronization according to the operation entries in the log.

➢ To synchronize the create operation, the server gets the content of the file from the work client, assigns a unique identifier FID for this file and creates a metadata entry for the file both on the server side and the work client. Since the file is synchronized to the server for the first time, the version number of its corresponding metadata entry is set to 1.

- Similar to the synchronization of create operation, when synchronizing a modify operation, the server gets the content of the file from the work client to update the content of the file on the server, and updates the metadata of the file both on the server and the work client. Because the synchronization of a modify operation changes the content of the file on the server, the version number of the modified file should be incremented.
- When synchronizing a rename operation, the server modifies the file path according to the parameters in the operation entry and correspondingly changes the metadata of the file on the server side and the work client. A rename operation does not cause any change in the content of the file, so the version number of the renamed file remains unchanged.
- To synchronize a delete operation, the server deletes both of the content and metadata of the file on the server.

Of course, we may encounter various conflicts in the synchronization process. We record the detected conflicts in a temporary conflict table, and wait until the upload synchronization is completed for centralized processing. Specific conflict detection and resolution strategies will be discussed in Section 4.

So far, we have completed the upload synchronization from the client to the server. In this process, all of the update operations on the work client will be synchronized to the server.

Download Synchronization (Server -> Client)

Next we will do download synchronization from the server to the client, which downloads the updates on the server to the work client. Since we have resolved the conflicts before, there will no longer be any file conflict in the download synchronization process. So we only need to find out the different parts of the SynCS-folder on the server and the work client, and replace the old version on the server with the new version on the client.

First, the work client and the server depth scan their SyncCS-folder respectively to generate a view of SyncCS-folder (the sequences of metadata entries of all the files in the folder). Then the work client transmits the local views to the server, and the server generates a set of operation sequence according to the different parts of two views as follows:

c) Look for a match in the view on the client based on the keyword FID for each metadata entry in the view on the server. In case of no match, generate a Create operation entry. Otherwise, continue to compare the file paths (Path) of the two metadata entries. If they are inconsistent, generate a Rename operation entry. Continue to compare the file version numbers (Version) of the two metadata entries. If they are inconsistent, generate a Modify operation entry. If both the Path and Version are the same, then we consider the two files to be consistent.

d) Look for a match in the view on the server based on the keyword FID for each metadata entry in the view on the client. If there is no match, generate a Delete operation entry.

At last the server will send the generated operation sequence to the work client. According to this sequence the work client will perform download synchronization. It is worth mentioning that the file contents on the server will not change during the download synchronization, so the version numbers of files on the server will not change either.

B. Real-time One Way Synchronization

When the work client completes the two-way synchronization with the server, the user will be able to continue to update the files of the SyncCS-folder on the work client on-line. Once a user performs an update operation on any file, the client software can detect it and generate a corresponding operation entry and send it to the server. The server will synchronize the corresponding update according to this entry.

IV. Conflict Mechanism

When a client uploads the updates of a local file to the server, if the same file on the server-side has also been updated then a file conflict may be caused. In this section, we analyze all the possible situations that may cause a conflict, and correspondingly define six kinds of file conflicts, according to which a conflict resolution scheme is then proposed.

We define the notion of a conflict as follows: During the uploading synchronization, suppose we need to update a specific file with operation A. However the same file on the server-side has already been updated with operation B. If operation A and operation B would lead to different operating results, then a conflict arises, which is called A-B conflict.

A. Conflict Detection

In the preceding sections we mentioned that, during the uploading synchronization, the client sends the processed log to the server, and the server updates the relevant files according to the operation entries in the log afterwards. The operation entries recorded in the log can be divided into four categories: create, modify, delete, and rename. We analyze the file conflicts that may be caused by the four update operations, and propose a corresponding conflict detection mechanism.

When we synchronize a create operation: First we search the server to see if there is a file with an identical path (i.e. file with the same name). If there is such a file then a ***file name conflict*** arises. Otherwise the server accesses the file contents from the client, generates a unique identifier – FID for this file, and sets the value of Version to 1. Then the metadata entry of this file is inserted into the metadata tables, and is returned to the client at the same time.

When we synchronize a modify operation: First we search the server to see if there is a file with the same FID. If there is no such file then a ***modify-delete conflict*** arises.

Otherwise we compare the version numbers of the files on the client and on the server-side. If the two version numbers are equal to each other, then we update the file content on the server-side. If they are not consistent then a ***modify-modify conflict*** arises. Whenever the content of a file on the server-side is modified successfully, the version number on the server-side is increased by 1, and the metadata on the client are also updated. Modify-rename conflicts are not considered here, because we believe that modify operations change the content of a file, while rename operations change a file's metadata. Thus they are two different kinds of operations that can occur simultaneously and can be merged.

When we synchronize a delete operation: First search the server to see if there is a file with the same FID. If there is no such file then do nothing. Otherwise compare the version numbers of the deleted file and the file on the server-side. If the two version numbers are consistent then delete the content and metadata of the corresponding file on the server-side. If the two version numbers are not consistent then a ***delete-modify conflict*** arises. Delete-rename conflict is not considered here, as rename operations will not change the content of a file, thus we believe that the renamed file is the same as the file to be deleted.

When we synchronize a rename operation: First search the server to see if there is a file with the same FID. If there is no such file then a ***rename-delete conflict*** arises. If there exists such a file then compare the OldPath of the client file with the path of the file on the server-side. If the OldPath is equal to the server path, then rename the server file with the NewPath. Otherwise compare the NewPath of the client file with the path of the server file to see whether they are consistent. If they are consistent then do nothing. if not, then a ***rename-rename conflict*** arises. Similarly, we do not consider rename-modify conflict here.

B. *Conflict Resolution*

In this sub-section we will discuss how to resolve the six kinds of conflict we've mentioned before.

File name conflict: Alert the user that the file name has already been occupied. And request the user to change the corresponding file name.

Modify-delete conflict: Warn the user that the file has been deleted on the server-side, and query the user whether to create this file again on the server-side or delete the local file. If user chooses the former option, then the server creates this file again, assigns a new FID to this file, sets the version of this file to 1, and updates the metadata on both sides. However if the user chooses the latter, then the client deletes the local file along with the file metadata.

Modify-modify conflict: Warn the user that there are conflicting files on the server-side. Display the details of the conflicting files so that the user can select to save two files at the same time or just to save one. If the user chooses to save the two files at the same time, then the local file will be automatically renamed (we can add a version mark to the original file name) and uploaded as a new file. After this the server assigns the uploaded file a new FID, sets the version of this file to 1, and updates the metadata on both sides. If the user chooses to save the local file, then the file content on the server-side is updated, and the file version number on the server-side is increased by 1, and the client metadata is updated accordingly. If the user chooses to save the file on the server-side, then we can do no operation and wait for the file to be synchronized to the client during download synchronization.

Delete-modify conflict: Warn the user that there is an update about the file to be deleted, and query the user whether they determine to delete the file. If the user determines to delete the file, then the server deletes the corresponding file content and metadata. Otherwise we do no operation and wait for the file to be synchronized to the client during download synchronization.

Rename-delete conflict: Resolution to this conflict is analogous to the one to the modify-delete conflict.

Rename-rename conflict: Alert the user that the file name has been changed on the sever-side, and query the user which path to keep.

V. EXPERIMENTS AND RESULTS

Our experimental environment includes a metadata server, a sync server, a storage server, and two laptop computers, which are connected in a local area network.

We simulate the situation in which people use a file synchronization system to test our prototype. We assume that in the daily file operations of an average person modify operations account for 80% of all file operations, rename operations account for 10%, create operations and delete operations account for 5% respectively. The number of the files that need to be synchronized is supposed to be around 100, and each file has an equal probability to be updated. During the offline operation between two synchronizations, the average updating frequency of each file is basically about 10 times. We set an operation sequence in accordance with the proportion of each file operation to simulate a user's operation.

Different network environments will affect synchronization time delay. Thus, we take the number of operations that we need to complete in a synchronization process as the measurement criteria. We compare the number of operation entries that need to be synchronized when using the log replaying method [14] and our proposed log merging algorithm. The test results are shown in figure 3 and figure 4.

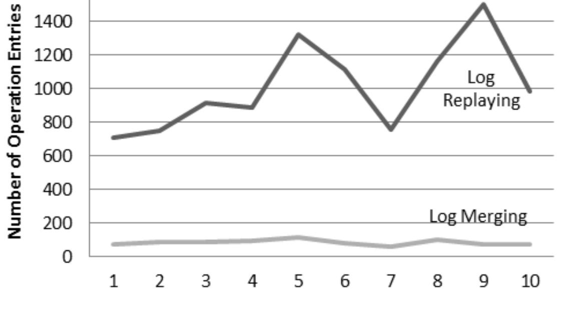

Figure 3. Contrast of total number of operation entries

Figure 3 shows the total number of operation entries that need to be synchronized when using two algorithms respectively. The x axis represents the 10 results we get. As it can be seen from the figure that, for the same file scale and operating frequency, the number of operations to be synchronized using log merging algorithm is much smaller than the one using log replaying method, and it is relatively balanced each time.

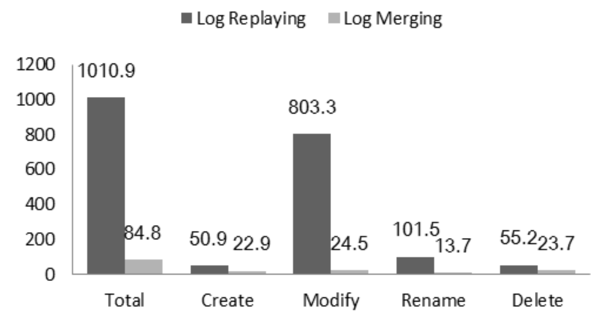

Figure 4. Contrast of average number of each operation entries

Figure 4 shows the average number of operation entries that need to be synchronized when using the two algorithms respectively. By using our log merging algorithm, we find that the number of operation entries will be significantly reduced, especially for modify and rename operations. Among all kinds of operations, the number of modify operations decreases the most, because it takes the largest proportion and reduces to at most one for each file.

VI. Related Work

Data synchronization technologies have been used in failure recovery of storage system for a very long time. A widely used method is log replaying [14, 15]. Just as its name suggests, in this method, the operation log is sent to the device that needs to be synchronized, and the synchronization process is performed strictly according to the operation sequence recorded in the log. However there may be many redundant operations in the log that will consume a large amount of time and network flow, and may even lead to some unnecessary conflicts. Our approach includes a log merging algorithm which can greatly reduce the number of operations that need to be synchronized.

Conflict resolution is an essential component of a file synchronization protocol. In SyncViews [15] file conflicts are detected based on timestamps, and a file that was later received by the server is considered to be the newest. This approach may miss some updates when there are offline operations. In contrast, our conflict detection and resolution mechanism considers all the situations that may occur in a two-way synchronization.

VII. Conclusion and Future Work

File synchronization is an effective way to conduct cross-device data management. In this paper we proposed a two-stage file synchronization protocol which combines a log merging algorithm with a view comparing algorithm. In addition, we presented a conflict resolution mechanism based on our proposed protocol, which addresses conflict detection and resolution in a two-way synchronization scheme. Our experimental results show that by using our method, the number of operations that need to be synchronized is significantly reduced when compared with the log replaying method.

In our protocol, the local update operations are paused during two-way synchronization, which will cause certain time of service stop. In our future we will address this problem and will build our synchronization system in a real cloud environment.

Acknowledgment

This work was supported by a grant from the Hubei Provincial Department of Education scientific research programs for youth project (No. Q20133003)

References

[1] Alzain, Mohammed, Soh Ben, and Pardede Eric, “A survey on data security issues in cloud computing: From single to multi-clouds”, Journal of Software, v 8, n 5, pp.1068-1078. 2013.

[2] David Dearman and Jeffery S. Pierce. “It's on my other computer: computing with multiple devices”, In Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems (CHI '08). ACM, New York, NY, USA, pp. 767-776.

[3] Dongbo Liu and Xiao Peng, “PM&E-CP: Performance monitor and evaluator for cloud platforms”, Journal of Software, v 8, n 4, pp.761-767, 2013.

[4] Plaxo. www.plaxo.com..

[5] Google Browser Sync. www.google.com/tools/firefox/browsersync/

[6] Dropbox. http://www.dropbox.com/

[7] Everbox. http://www.everbox.com/

[8] Kuaipan. http://www.kuaipan.cn/

[9] iFolder. http://www.ifolder.com/ifolder

[10] J. Plaice and W.W. Wadge, “A New Approach to Version Control”, IEEE Trans. Software Eng., pp.268-276, 1993.

[11] Amazon S3. http://aws.amazon.com/s3/

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google file system”, In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, pp. 29-43, 2003.

[13] D.Borthakur, “The Hadoop Distributed File System: Architecture and Design”, The Apache Software Foundation, 2007.

[14] Huang Lu, Hai-Shan Chen and Ting-Ting Hu, “Survey on resource allocation policy and job scheduling algorithms of cloud computing”, Journal of Software, v 8, n 2, pp.480-487. 2013.

[15] Bao Xianqiang, Xiao Nong, Shi Weisong, Liu Fang, Mao Huajian and Zhang Hang, "SyncViews: Toward Consistent User Views in Cloud-Based File Synchronization Services", *Chinagrid Conference (ChinaGrid)*, pp.89-96, Aug. 2011.

Chao Liang received the PhD degree from Institute of Computing Technology Chinese Academy of Science in 2002. Now he is the Chief Architect of Lenovo Mobile Communication Technology Ltd. His current research interests include mobile cloud computing and cloud storage.

Luokai Hu received the MS degree from Wuhan University, China, in 2006 and PhD degree in State Key Lab of Software engineering at Wuhan University in 2011. He is currently a post doctor of Lenovo Mobile Communication Technology Ltd. And he is also a lecture at Hubei University of Education. His current research interests include Semantic Web and cloud computing.