SyncViews: Toward Consistent User Views in Cloud-based File Synchronization Services

BAO Xianqiang*, XIAO Nong*, SHI Weisong+, LIU Fang*, MAO Huajian*, ZHANG Hang*
*National University of Defense Technology, Changsha, Hunan 410073, China
+Wayne State University, 5057 Woodward Ave, Detroit, MI 48202, USA
{baoxianqiang, zhanghang0259}@gmail.com, {nongxiao, liufang, huajianmao}@nudt.edu.cn, weisong@wayne.edu

***Abstract*—As more and more individuals have multiple computing devices, how to maintain the consistency of user files stored on multiple devices becomes a challenge. Cloud-based file synchronization services (FSS), e.g., Dropbox and iFolder, are promising approaches. However, we haven't seen any public documents discussing the consistency protocol, which plays a vital role in any FSS. This paper focuses on maintaining the consistency of user files in FSS. We define a new file abstraction layer called *user view*, which describes all the files and their metadata in a sync-folder. We propose a user view based file consistency protocol with a conflicts resolution mechanism for FSS. The consistency protocol automatically keeps user files synchronized among the clients and server. We implement our consistency protocol into a prototype called *SyncViews* and set up a LAN-based environment to evaluate its feasibility and performance. We measured the time consumption of each synchronous operation on SyncViews and compared with iFolder, an open source implementation. The results show that our protocol works well during the file synchronization processes and outperforms iFolder when the synchronized file becomes very large.**

Keywords-file synchronization service; consistency protocol; user view; cloud storage service

I. INTRODUCTION

Nowadays, more and more individuals have multiple computing devices. An individual may have a desktop at work, a desktop at home, a personal laptop, a smart mobile phone and even a Pad like iPad. All these devices bring great convenience of personal information management (PIM) in this mobile world. However, to manage some specific files which stored on various computers efficiently can become difficult [1, 6]. We describe some common problems as following:

First, can't always access the right documents on the right device. Some important documents may be stored on another device and you didn't carry the certain device with you. Second, can't always access the right version of a document. When you are working on a document you always want to access to the latest version of the file, but you might have saved the document on your office computer, home computer, laptop and perhaps even on a pad with different versions. So it will be confusing for you to keep track of the most current version. Last but not the least, can't always ensure data security. You could have your devices crash, reinstalled or even stolen, and lost weeks or months of work.

Cloud-based file synchronization service (FSS) [3, 4, 15, 16, 17] is a good solution to address these challenges, which is often promoted as a cloud storage service. The service can synchronize user files among a user's various computers automatically. As long as a user pay for the FSS, this service can allow user files automatically follow the user everywhere either on line or off line. A user can access the synchronized files from every owned computer or even through a standard web browser. As the number of computing devices owned to an individual increases, FSS can be a very useful daily information storage service. Many corporations have already launched a series of products, such as Dropbox [4], LiveSync [15], Everbox [16], DBank [17], iFolder [3], and so on.

However, most of the FSS providers didn't disclose the details of their products. Though iFolder[3] is an open source approach from Novell corporation, it can't work efficiently when the file size becomes large in our LAN-based environment and iFolder delivers all the conflicts to user, this mechanism sometimes may puzzle the user to decide which file to save. Thus, two most important challenges in FSS include:

First, the file synchronization consistency protocol, which maintain the consistency of user files efficiently. And user files not only stored on users' multiple computing devices, but also on the storage server in the cloud data center which provides FSS. Once a file has been updated, it will be synchronized to other clients and server sooner or later. In other words, all update operations should be detected and noticed to other clients and server.

Second, solving the conflicts may arise during file synchronization. Considering a user might modify the files on two or more clients at the same time, it could result a conflict when the files being modified on these clients are the same replications. Then, it becomes a challenge to decide which modified file to be synchronized to the server and other clients. We believe that the last received modified file is the latest version, and we use timestamps [18, 21] to detect conflicts intelligently and handle the conflicts on the server side by synchronizing the latest file to other clients.

Based on the discussion represented above, we introduce an abstraction layer of user files called *user view*, which defined as the snapshot of all files and directories stored in the sync-folder. We propose and implement a file synchronization consistency protocol with conflict resolutions based on user views. Specifically, the contributions of our work include as following:

- We propose a FSS model, which provides some details of the FSS as most of the providers will not disclose the details of their products.

- We introduce a new file abstraction layer—user view in FSS. We provide user views on multiple hosts including the server to keep user files always being synchronized.
- We design a user view based consistency protocol, which efficiently maintains the consistency between multiple clients and the server. The consistency protocol can automatically synchronize files in the background. And all the synchronous operations are transparent to users.
- We propose a conflicts resolution mechanism, which intelligently detects the possible conflicts during synchronous operations and handles the conflicts on the server side automatically for users.

Note that our protocol targets for users who want to synchronize files stored on various devices, typically on the computers using in different occasions, or the user wants to backup files remotely and always view the latest version without any redundant operations such as upload and download. We design an eventual consistency protocol [7, 9, 12] with automatic conflicts detecting and handling mechanism. The above techniques will be incorporated into a prototype called Sync-views, the evaluation results indicate that our consistency protocol works well in the LAN-based FSS and outperforms iFolder when the file size becomes very large.

The rest of the paper is organized as following: Section 2 depicts the detailed introduction of FSS and the service model. Section 3 introduces the concept of user view and discusses the consistency based on user views. Section 4 proposes and implements the consistency protocol with conflicts resolution mechanism. Section 5 presents our experimental environment and evaluates the feasibility and performance of our consistency protocol. Section 6 discusses the related works. Finally, we present the conclusion of this paper and some future works in Section 7.

II. FILE SYNCHRONIZATION SERVICE

We first give a definition of file synchronization service, followed by an abstract model of FSS.

A. *Introduction of FSS*

As we mentioned before, it can become difficult to manage files that a user has stored on various computers, so there is a great demand in FSS. And numerous file synchronization products have been launched by foreign and domestic companies.

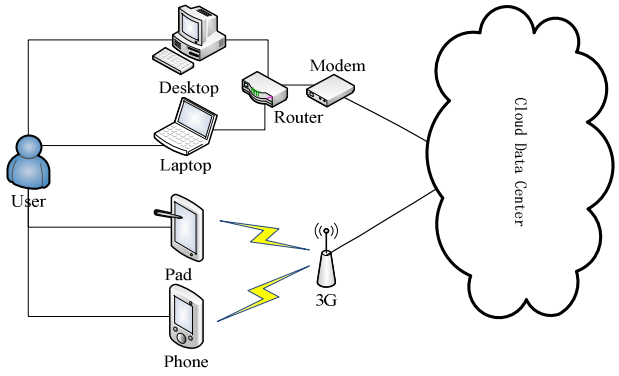
Here, we take Dropbox for an example to represent how a synchronization system works. First, a user need to install the Dropbox client on all his devices, then the user can designate a folder as the sync-folder, and the client will monitor the sync-folder automatically. Once a file in any sync-folder has been updated, the client can detect the update operation and synchronize the updated file to other clients at once. The synchronization processing always runs in the background to ensure file consistency among multiple devices.

Summarizing the characteristics of the related products, we define FSS as an online storage service, which can automatically synchronize user files stored on local host to remote cloud data center or other hosts. Most FSS providers use the pay-as-you-go business model, users pay the necessary storage space where they can access their synchronized files through any standard web browser. Also, a user can simply pay more to get more space when his/her needs increase. Thus, this service is always considered a cloud service, especially a cloud storage service.

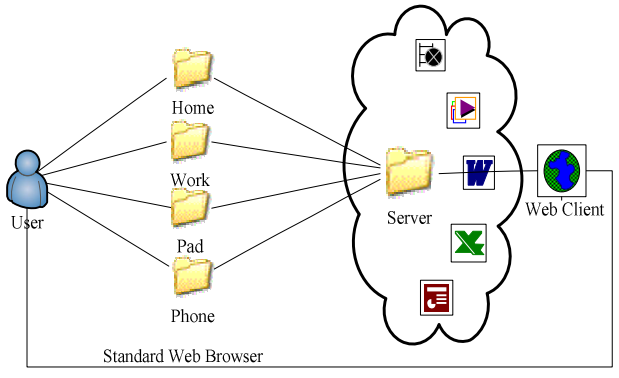
B. *A Model for FSS*

Suppose a user has a desktop at work, a desktop at home, a personal laptop, a smart mobile phone and even a Pad like iPad. The user pays the service space according to his/her needs or gets free space after registration. Then the user downloads the client software and installs it into multiple devices. After all the installations are finished, a folder should be designated as the sync-folder on each client, and all the sync-folders can correspond to the space bought from the service provider. So far, all files in the sync-folder will be monitored by a client.

If any file in the sync-folder has been updated, it will be automatically synchronized to other sync-folders. Update operations include create, modify, rename, delete operation, which should always be informed to the server and other clients in a certain period of time, in order to ensure all the user files can maintain consistency after the user has updated a file in any sync-folder. In addition, the user can access the synchronized files through any standard web browser and do some operations like upload, download. Also, the upload operation can update the server sync-folder and should be synchronized to every other client. Based on the description above, we give two models of FSS shown as Fig. 1.



(a)A general multiple devices model for FSS



(b)A general multiple sync-folders model for FSS

Figure 1. Two general models for FSS.

III. CONSISTENCY OF USER VIEWS

In this section, we introduce the notion of user view, then present a user views based file synchronization model. Details on how to maintain consistency will be described in the following section.

A. User View

We provide a new abstraction layer of user files—*user view* to maintain file consistency in our protocol. A user view is defined as the snapshot of all files and directories stored in a sync-folder, including names and attributes. As a user always has multiple sync-folders, the user can have multiple user views when using different devices, typically the server user view is generated for the user when accessing user files through web browser. We call all the user views generated on multiple devices as *SyncViews* when all the user views become consistent, and the synchronization system should always maintain user views being consistent according to the core function of FSS.

Logically, a user view includes three components: files and their attributes, directories and their attributions, and historical update operations of these files and directories. Here we treat directories in the same way as files (when a file is a directory, the file type is marked as directory), so a user view only includes two main components: files and their attributes and historical operations of user files (including the directories). We give a detailed description about user views as following:

Each file in a user view is associated with a six-element tuple: $file<f_{ID}, f_{user}, f_{path}, f_{type}, f_{size}, f_{time}>$, where f_{ID} is the unique file identifier in SyncViews; f_{user} represents the user name who owns this file; f_{path} is used to describe the file storage location on server; f_{type} is the file type, a directory is described as a directory type file; f_{size} is the file size, here we set it to N/A when the file is a directory file; f_{time} is the timestamp when the file is synchronized to server , and it is automatically attached to the file attributes by server when stores the file in sync-folder on data storage server.

Also, each historical update operation is associated with a six-element tuple: $update<u_{ID}, u_{user}, u_{cmd}, u_{path}, u_{oType}, u_{time}>$, where u_{ID} is the unique update operation identifier in SyncViews; u_{user} is the user name who owns this file; u_{cmd} is the update operation command, such as *create, modify, rename, delete*; u_{path} is the user file location on server; u_{oType} is the file type which has been updated; u_{time} is the timestamp when the update operation is synchronized to server , and it is also automatically attached to the historical update operation metadata by server when a file has been updated in any client sync-folder.

B. Synchronization Model based on User Views

A user view is always generated from a logged on client. Thus, there is a series of user views from office desktop (v_{work}), home desktop (v_{home}), laptop (v_{laptop}), smart phone (v_{phone}) and Pad (v_{pad}) for the user, even a user view (v_{server}) is generated when the user accesses files in server sync-folder.

We use V_{user} to represent the set of all user views that $V_{user}=\{v_{work}, v_{hone}, v_{laptop}, v_{pad}, v_{phone}, v_{server}\}$, and we abbreviate SyncViews to V_{sync}. Then, we can build another abstract model of FSS based on user views, shown as Fig. 2. V_{sync} can provide a global vision for file synchronization system to maintain file consistency among clients and server.

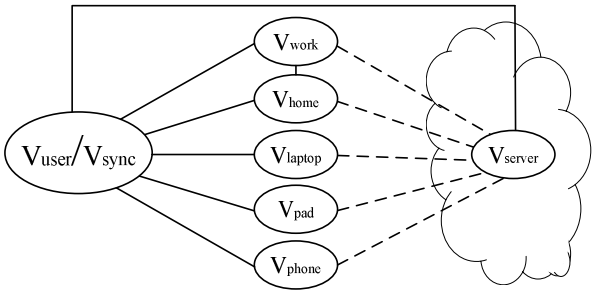


Figure 2. The user views based model of FSS. (V_{user} is different from V_{sync}, V_{user} doesn't focus on whether the user views have finished synchronization or not, while V_{sync} depends on the consistency of all user views.)

C. Consistency of User Views

First, the consistency of user views is defined as all the user views in V_{user} are the same, in other words, the consistency of user views means that a user can always access the same files in any sync-folder when logs on a client including the web client. It also represents as $V_{user} \equiv V_{sync}$, and $v_{work} \equiv v_{home} \equiv v_{laptop} \equiv v_{pad} \equiv v_{phone} \equiv v_{server}$, while $V_{user} \equiv V_{sync}$ is expressed as all user files in each sync-folder have finished synchronization, and all user views become consistent. $v_{work} \equiv v_{server}$ is expressed as all user files in sync-folder on office desktop have already been synchronized to the sync-folder on server, $v_{work} \equiv v_{home}$ and so on has a similar meaning.

Here we take an example to describe the detail of the consistency of user views. Supposed that $V_{user0} \equiv V_{sync0}$ (V_{user0} means V_{user} at a timestamp $time_0$, similar to others), so there is $v_{work0} \equiv v_{home0} \equiv v_{laptop0} \equiv v_{pad0} \equiv v_{phone0} \equiv v_{server0}$ at $time_0$. The user view v_{home0} can be generated for the user when logs on home desktop and the user haven't done any update operations after $time_0$. Then, the user updates a file in sync-folder on home desktop at $time_1$, v_{home0} for the user changes into v_{home1} immediately, while the other user views remain the same as before (user views at $time_0$). So the set of user views V_{user} becomes $V_{user0}=\{v_{work0}, v_{hone1}, v_{laptop0}, v_{pad0}, v_{phone0}, v_{server0}\}$, and V_{sync0} should be updated to $V_{sync1}=\{v_{work1}, v_{home1}, v_{laptop1}, v_{pad1}, v_{phone1}, v_{server1}\}$. As a result, $V_{user1} \neq V_{sync1}$ at $time_1$, it is expressed as that not all the user views are the same in both V_{user1} and V_{sync1}, which means the consistency of user views has been broken.

However, the consistency protocol needs to maintain the consistency of user views. Thus, the update operation should automatically be detected by the client on home desktop and the updated file will be synchronized to the server and other clients sooner or later in FSS. First, the updated file is synchronized to the server according to our design, so $v_{server0}$ for changes into $v_{server1}$ and there is $v_{server1} \equiv v_{home1}$. Then, other clients can get the updated file from server when logged on by the user, so v_{work0}, $v_{laptop0}$ and other user views change into v_{work1}, $v_{laptop1}$ and so on. Finally, there will always be $V_{user1}=\{v_{work1}, v_{home1}, v_{laptop1}, v_{pad1}, v_{phone1}, v_{server1}\}$ and the set of user views V_{user1} is satisfied with $V_{user1} \equiv V_{sync1}$ after the update operation from home desktop. To this end, all the user views become consistent again.

IV. USER VIEWS BASED CONSISTENCY PROTOCOL

To maintain the consistency of user files in sync-folders, every client interactively works with the server. Here we describe the process about the first consistency among clients and the server as following:

After a user has paid for the FSS, all user files' metadata represented as the first user view generated from the first logged on client (suppose it's the home client, and it is the first installed client), home client puts the first user view (v_{home0}) as well as all the file contents to the server immediately, then the first user view on server ($v_{server0}$) is generated. If the user logs on another client (suppose the work client) for the first time, the work client queries the server and gets all the files in server sync-folder to the work sync-folder, so the first user view on work desktop is generated. Similarly, other first user views will also be generated. According to the above discussion, first consistency of all user views can be formed.

A. Consistency Protocol Design

After the user views become consistent, maintaining the consistency of user views is required when an update operation arises in any sync-folder. At first, we give some definitions of the related operations in our protocol. The related operations include two update operations and two scan operations. First, we define the two update operations: a *local update* is defined as an update operation on a file in the sync-folder which is currently being used by the user; a *remote update* is defined as an update operation on a file in the server sync-folder. Then, we define the two scan operations: a *local scan* is defined as a logged on client scans its sync-folder periodically in case there is a local update operation; a *remote scan* is defined as a logged on client scans the server sync-folder (queries the historical update operations stored on meta-server) periodically in case there is a remote update operation.

Suppose that all user views become consistent at $time_0$, we describe the process of maintaining the consistency of user views when an update operation is detected by a client behind $time_0$ as following steps:

a) After a user has logged on a client (supposing the work client), an original version of work user view v_{work0} is generated. The user opens the work sync-folder and updates a file some time later. Then the user view for the user from work client changes from the original version v_{work0} into a second version v_{work1}.

b) The work client can detect the update operation when it has completed a periodical local scan of the work sync-folder and immediately generate an update entry in the form such as $update^c<u_{user}, u_{cmd}, u_{path}, u_{oType}, [u_{content}]>$ ($u_{content}$ is a optional parameter according to the update operations, such as a creation or modification entry includes $u_{content}$, other parameters are similar to the description in Section 3).

c) The work client sends the update entry to server as soon as possible. Then the server updates the server user view for the user according to the received update entry. We represent the update processing of server user view as the following: first, the server attaches a timestamp $u_{timestamp}$ to the update entry when received the entry, and the entry is changed into $update^s<u_{user}, u_{cmd}, u_{path}, u_{oType}, [u_{content}], u_{timestamp}>$. Then the server chooses a corresponding function to go judging by the parameter u_{cmd}. Then the server modifies the updated file's meta data on meta-server judging by u_{user} and u_{path}, and if $u_{content}$ isn't empty ($u_{content}$ is empty when u_{cmd} is marked with *'rename'* or *'delete'*), the upload content is stored as a new file in server sync-folder when u_{cmd} is marked with *'create'* or it covers the old file when u_{cmd} is marked with *'modify'*. So the server user view changes from the original version $v_{server0}$ into a second version $v_{server1}$, and $v_{server1} \equiv v_{work1}$. So far, the update on work user view has been synchronized to server, as well as to the web client.

d) After some time, the user has logged on another client (suppose the home client), also an original version of home user view v_{home0} is generated. The home client will always do a remote scan periodically. It sends a query command to the server, and the command is attached with the last remote scan timestamp $time_0$ returned from the server. Then the server queries the meta-server and returns all the historical update operations including the one has done in *a)* in the form of $update^s<u_{user}, u_{cmd}, u_{path}, u_{oType}, [u_{content}], u_{timestamp}>$ to the home client.

e) When the home client has received the update operation, it modifies the home sync-folder as same as the server does on the server sync-folder. Then the home user view can be changed from v_{home0} into a second version v_{home1}, and $v_{home1} \equiv v_{server1} \equiv v_{work1}$. Till now, the update on work user view has been synchronized to server and home client.

Similarly, the update will be synchronized to other clients by the same way. Thus, the consistency of all user views can be maintained through our protocol, shown as Fig. 3.

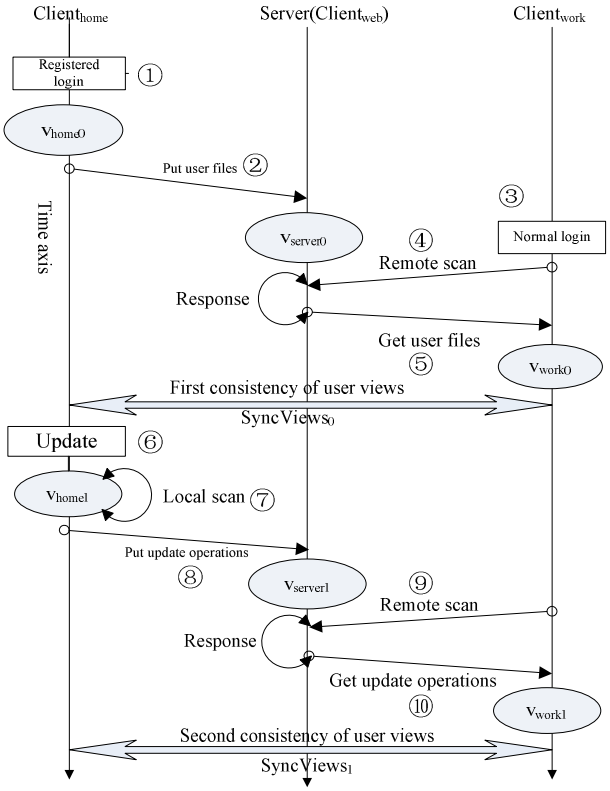


Figure 3. The details of the consistency protocol based on user views.

B. Interaction Interfaces

According to the above consistency protocol, the update operations include four operations: *create, modify, rename* and *delete*, so there are four types of synchronous operations in our consistency protocol: *create, modify, rename* and *delete* synchronous operation. In order to synchronize every updated file to the server and other clients, all the synchronous operations have been accomplished between a client and the server, which always start from a client to the server, finish from the server to other clients. So the design of the interaction interfaces between client and server is very significant in our consistency protocol, we design the following interaction interfaces listed in Table 1, ripe for the implementation of synchronous operations.

TABLE I. INTERATION INTERFACES

API	Parameters and Relative Descriptions	
	Parameters	***Function Description***
query	(user,prefix,timestamp,key)	find out updated files
list	(user,prefix)	list all user files on server
get	(user,uri,offset,length)	get operations or file data
put	(user,uri,oType,content)	send operations or file data
rename	(user,uriSrc,uriDst)	rename user files on server
delete	(user,uri)	delete user files on server

Then we represent the server-side structure in our FSS system, the server-side structure include three major components: control-server, meta-server and the storage server. The control-server handles every request from clients and sends the synchronous operations to meta-server and stores file content on storage server. After the synchronization processing has finished, control-server gets the response results from meta-server or storage server and transmits the results to the client, which started the interaction. According to the description above, we implement all the interaction interfaces on control-server and clients.

V. IMPLEMENTATION

The implementation of our consistency protocol is divided into the following four parts: *create*, *rename*, *delete* and *modify* synchronous operation, according to the four update operations. As our consistency protocol is implemented based on file granularity, the basic unit of the synchronous operations is a file. Review the definition of user view, a user view includes two major parts: files and their attributes and historical operations on user files. In our implementation, we use a table called *Snapshot* to store the metadata of files and their attributes, and a table called *ChangeRecord* to store the metadata of historical update operations. Both the *Snapshot* and *ChangeRecord* tables are stored on the meta-server.

Make a general hypothesis that there is a registered user A, and user A owns a home desktop, a work desktop and a personal laptop. Then, we describe the implementation details of the four synchronous operations as following:

A. Create Synchronous Operation

a) The user A logs on a client (suppose the home client), creates a file f_x in the home sync-folder. Then home client's periodically local scan can detect a new file f_x has been created, generates the entry $create_x<user_A, cmd_{new}, path_x, oType_x, content_x>$ then calls the *put* interface to send the entry to server.

b) The control-server receives the entry and attaches a timestamp for the recived entry, then the control-server generates a file entry $file_x<user_A, uri_x, type_x, size_x, timestamp_x>$ and insert it into the table $Snapshot_A$ through the *insert* function provided by meta-server. As well, a update entry $update_x<user_A, cmd_{new}, oType_x, timestamp_x>$ is inserted into the table $ChangeRecord_A$. At the same time, the control-server transfers $content_x$ to the storage server and $content_x$ will be stored as file f_x according to uri_x.

c) When user A logs on another client (suppose the work client), the work client's periodically remote scan can generates the entry $query<user_A, timestamp_{last}>$ ($timestamp_{last}$ is the return value of last query operation on server, the initial value is 0) and calls the *put* interface to send the query command entry to server.

d) When the control-server has received the query command, it calls the *query* interface in order to query the $ChangeRecord_A$ and the meta-server can send back all the update entries including the entry $update_x$ behind $timestamp_{last}$. Then the control-server sends all the update entries as the query results to work client.

e) After the work client has received the query results, the work client can call the *get* interface to download the new file f_x from server according to uri_x in the entry $update_x$, also other update operation can be done in the work sync-folder.

f) When user A logs on the other clients, here is the laptop client, the file fx can be created in laptop sync-folder similar to the work client.

B. Rename, Delete and Modify Synchronous Operation

The other three synchronous operations have similar procedures to the above create synchronous operation. However, the rename synchronous operation only changes the renamed file's metadata in $Snapshot_A$ and $ChangeRecord_A$ and the file content remains the same as before. The new file name will cover the old name in the related file entry when the meta-server updates $Snapshot_A$, and a rename entry will be inserted into $ChangeRecord_A$.

While the delete synchronous operation changes not only the $Snapshot_A$ and $ChangeRecord_A$, but also the file content. The file entry is deleted in $Snapshot_A$ and all the update entries related to the file are deleted in $ChangeRecord_A$. However, the file content is still stored in the server sync-folder on storage server, while the user A can't access the file through all the clients except web client. And the other clients delete the file in their sync-folders.

As the modify synchronous operations may result in file conflicts, it must deal with the conflicts represented in the following subsection D. And the more detailed description about the procedure of the modify synchronous operation is also described in the next subsection.

C. Conflicts Resolution

When a user has logged on two or more clients (suppose the work client and the laptop client) at the same time, and the user may attempt to update files both in work sync-folder and laptop sync-folder, the concurrent update operations can result in file conflicts while the modifying files are the same replication. We ignore the rare conflicts may arise during the create, rename and delete synchronous operation. Because a user is firmly impossible to create, rename or delete the same file in two sync-folders at the same time. While the user may always concurrently modify the same file on work desktop and personal laptop, such as editing a same source code file which will run on both two clients. So we mainly consider the conflicts arising in the concurrent modifications.

a) Detection of conflicts: When the user makes concurrent modifications to different clients, write-after-write file conflicts can arise while the same file replications are being modified. We describe a write-after-write file conflict in FSS as following: the user has finished modifying a file in laptop sync-folder, while the user also has finished modifying the same file in work sync-folder. Both the modify operations can be detected by clients, and the two modified file will be synchronized to server. The modified file later received by server will cover the earlier received one when the storage server saves the modified file. So the user misses the earlier file modification. We call the file modification missing as file conflicts in our consistency protocol.

We use timestamps to detect write-after-write file conflicts. According to our consistency protocol, all the modify operation entries will be attached a timestamp when received by the control-server. Before the control-server inserts a modify operation entry into the *ChangeRecord* on meta-server and saves the modified file content to storage server, the control-server queries the *ChangeRecord* table to find if there is a recent modify operation entry (the same user and the same file) in *ChangeRecord*, then determines whether a conflict is arose with the timestamp of the latest modify operation entry.

b) Resolution of conflicts: Our consistency protocol can detect the write-after-write file conflicts through timestamps comparison and also provide means to resolve such conflicts. We take a radical approach to deal with the detected conflicts that we only present the conflicts to server and always synchronize the later received file to other clients, so the files in sync-folders on clients are always the latest version (We believe a file later submitted by the user is always more up-to-date when other environment like networks is similar to clients). All conflicts are resolved in the server side, and the algorithm details are shown as Fig. 4.

When the control-server has detected a conflict (step 4 in Fig. 4), it saves the earlier received file as an old version $file_0$ before covered by the later received file (step 5 and 6). Then, the control-server generates a file entry $file_0$ of the old version and inserts the entry $file_0$ into *VersionRecord* (a table stored on meta-server only for web user view, step 7). As well, then the control-server inserts the modify operation entry according to the later received file entry $file_0$ into *ChangeRecord*, updates the *Snapshot* with the modify operation's timestamp $time_1$ and stores the file on storage server (step 8, 9 and 10).

Algorithm: modify(user, uri, $time_1$, content, ACK)
Input:
user: the user who modified the file.
uri: the location to store the modified file on storage server
$time_1$: the timestamp attached by control-server when receives a entry
content: the modified file's content
Output:
ACK: an ACK send to the client started the interaction
1. $time_0 \leftarrow$ ChangeRecord.query(user, $time_1$, cmd)
2. $flag_{conflict} \leftarrow$ false
3. if(($time_1$ - $time_0$) < TIME) then
   //TIME is the threshold for conflicts estimation
4. $file_{temp} \leftarrow$ read(user, uri)
5. $file_0 \leftarrow$ write(user, uri_0, $file_{temp}$)
6. VersionRecord.insert(user, $entry_{file0}$)
7. $flag_{conflict} \leftarrow$ true
   end if
8. file $\leftarrow$ write(user, uri, content)
9. Snapshot.update($entry_{file}$, $time_1$)
10. ChangeRecord.insert($entry_{file}$, cmd, $time_1$)
11. ACK $\leftarrow$ $flag_{conflict}$

Figure 4. The modification conflicts resolution mechanism.

VI. Experiments And Evaluation

A. The Experiment Enviornment

We set up the experiment environment in a real local network environment. The clients include two desktops and a laptop, and the two desktops are wired connected via HuaWei router while the laptop is connected to the same router through wireless network. The servers include the control-server, the meta-server and the storage server, and they are connected via RuiJie switch. The HuaWei router is also connected to the RuiJie switch, so all the clients and servers are connected in a LAN shown as Fig. 5, the main configurations about the clients and servers are listed in Fig. 5.

We implement all the interaction interfaces with Java language on the server side, and provide standard RESTful interfaces based on HTTP, while the client is implemented with C# language.

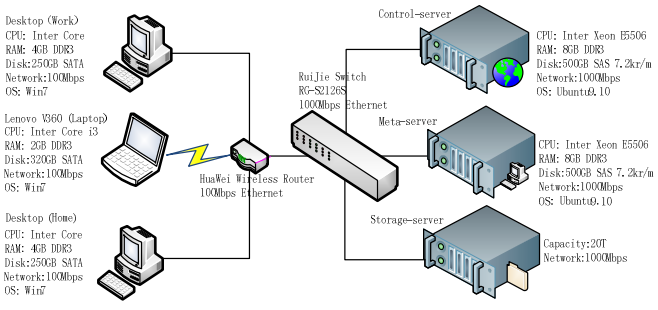


Figure 5. The LAN-based experiment environment.

B. Results and Evaluation

We use a series of files range from 100KB to 1GB with different types to test the four synchronous operations in our prototype SyncViews and iFolder on three clients—a laptop and two desktops, the different file sizes are listed in Table 2. We test the consistency among these clients and test the latencies of the four synchronous operations on each client.

TABLE II. THE TESTED FILE SIZE AND TYPE

size (MB)	0.1	0.5	1	5	10	50	100	500	1000
type	txt	doc	pdf	mp3	mp3	rar	wmv	rmvb	rmvb

a) The latencies of create synchronous operation: The result is shown in Fig. 6 (a). When the file size < 10 MB, both SyncViews and iFolder has a very similar latencies. When the file size > 100 MB, both the latencies of SyncViews and iFolder increase rapidly. However, iFolder's creation latency increases more sharply than SyncViews that while the file size ≥ 500 MB, the delay of create synchronous operation of iFolder is nearly three times than SyncViews. It is may mainly because iFolder uses the HTTPS to transfer the user files that increases its latency when the file size becomes very large.

b) The latencies of modify synchronous operation: The result is show in Fig. 6 (b). When the file size ≤ 5 MB, we use a doc type file with different sizes and we add 1 KB content to the end of each file, the latencies is not distinguish. When the file size ≥ 10 MB, we use a rar type compressed file (the rar compressed files with different sizes are included lots of pdf, doc, ppt and etc files) and we delete a 1MB file in the compressed file. As our consistency protocol is implemented based on file granularity, the latencies of modify synchronous operation is very similar to create synchronous operation. There is an abnormality that when file size increases to 1 GB, iFolder takes much more time to synchronize the modified file to the server, even than create a 1GB new file in the sync-folder. This is mainly because iFolder uses the Rsync [22] algorithm, and the iFolder client should download the old synchronized file from the server sync-folder first, then generate file difference which represents the new modified file and synchronize it to the server.

c) The latencies of rename synchronous operation: The result is show in Fig. 6 (c). In SyncViews, the latencies of renaming files with different size are similar, needing 2~3 seconds to rename a file. While in iFolder, the latency increases when the file size becomes larger.

d) The latencies of delete synchronous operation: The result is show in Fig. 6 (d). Both the SyncViews and iFolder need a short time between 1~3 seconds to synchronize a delete operation to server, and the latencies change a little when the file size varies.

Since all the results are tested in a LAN-based network, we can prove the feasibility of our consistency protocol based on user views. Meanwhile, we evaluate the performance of our prototype SyncViews compared with iFolder by test the latencies of the four synchronous operations. Above all, the results shown as Fig. 6 indicate that SyncViews outperforms iFolder especially when file size becomes very large.

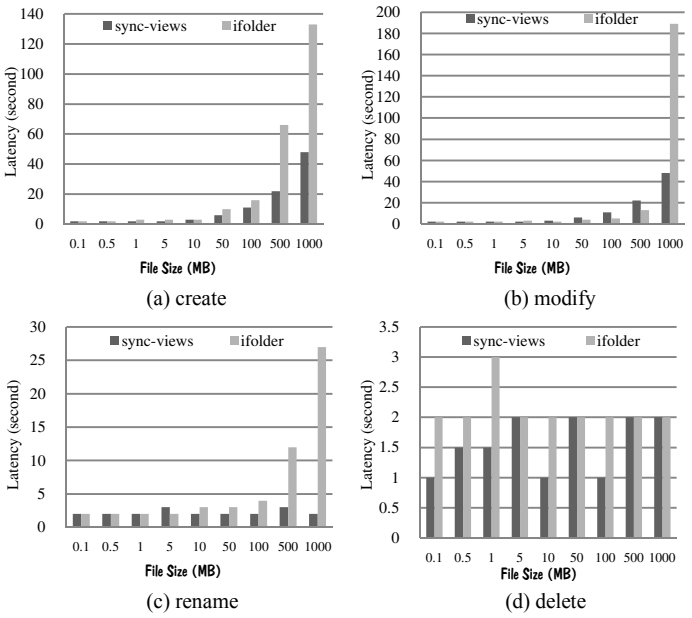


Figure 6. Latencies of the four synchronous operations.

VII. RELATED WORK

The discussion of related work is mainly divided into three parts: FSS, abstract view, consistency protocol. We represent the details as the following:

FSS: Currently, it's common for many people computing with multiple devices. David et al. [1] did some interviews with some people from academia and industry using multiple devices. Their findings indicate that a user's information and activities may span multiple devices. And it's a challenge for the user to manage information and activities across multiple devices efficiently [6]. Automatic file synchronization is a feasible approach to address the challenge, and iFolder [2] from Novell is an open source approach to provide FSS. However, iFolder can't work efficiently when the user file size becomes very large in our LAN-based environment. And iFolder delivers all the conflicts to user, this mechanism sometimes may puzzle the user to decide which file to save.

Abstract view: The abstract view is often used to represent a specific relationship property in data storage at different layers. Karlson et al. [5] proposed a user's conceptual view—*versionset* to represent a set of digital items, defined as a set of files with different versions of the same document, which only focusing on a single document. AFS [11] proposed an abstract view—*whole-file caching*, which only focusing on cache management. Cegor [2] provides an abstract layer called *semantic view* to make its distributed file system for heterogeneous network environment (Cegor) adaptive. Our approach proposed another layer of abstraction different form the above abstract views, called *user view*, in order to maintain file consistency in our protocol.

Consistency protocol: There is a great deal of prior work in maintaining the file consistency in a certain storage system [2, 9, 11, 13, 14, 15]. However, a lot prior works focus on strong consistency for data storage system and can't adapt to maintain the consistency in FSS. We only discuss some of

them, which may adapt to FSS here. The bayou architecture [9] maintains weakly data consistency to maximize a user's ability to read and write data, even while completely disconnected from the rest of the computing environment. And a reconcile consistency protocol is proposed in Cegor [2], which assures the modifications made by on client are guaranteed to be visible to another client. While we concentrate on the consistency requirements for a user computing with multiple computers, we proposed an eventual consistency [7, 9, 12] protocol based on user views and give a timestamp based conflicts resolution mechanism, and implement them into a prototype.

VIII. CONCLUSION AND FUTURE WORK

FSS is very promising cloud storage service in our daily life. In this paper, we propose a consistency protocol and implement a prototype called SyncViews. In our consistency protocol, we introduced a new abstraction layer of user files—*user view*, in order to maintain the file consistency during the synchronous operations. This layer of abstraction is closely related to the user behavior. Thus, we implement our protocol into four synchronous operations: *create, modify, rename, delete* according to the user behaviors in sync-folders. As conflicts may arise during the *modify* synchronous operation, we propose a conflicts resolution mechanism based on user views.

Our evaluation is conducted in a LAN-based environment, and the results show that our user-views based protocol work efficiently in FSS. We also evaluate the four synchronous operations' time consumption when file updates arise in sync-folders, and we compared the time consumption with iFolder, the results indicate that SyncViews outperforms iFolder when the synchronized file becomes very large, and has a similar performance to iFolder when the file size is small.

In our future work, a high concurrency supported file consistency protocol will be designed to provide efficient service when the number of users increases sharply. Also, a more efficient file synchronization algorithm will be proposed to decrease the data transmission during the *modify* synchronous operation. Last but not the least, our future work includes establishing a better file-sharing subsystem in our protocol in order to let users to share their files among different accounts, not only by a same account through web browser.

ACKNOWLEDGMENT

We thank associate professor LAI Mingche for discussing lots of issues with him. This work is partially supported by National Natural Science Foundation of China grants NSFC 6073013, NSFC 60903040 and Program for New Century Excellent Talents in University NCET-08-0145.

REFERENCES

[1] David Dearman, Jeffery S. Pierce. It's on my other computer!: computing with multiple devices. Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems, Florence, Italy, April 05-10, 2008.

[2] W. Shi, H. Lufei, and S. Santhosh. Cegor: An adaptive, distributed file system for heterogeneous network environments. Proceedings of the Tenth International Conferences on Parallel and Distributed Systems, 2004, 10: 145-142..

[3] iFolder. http://www.kablink.org/ifolder

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

[5] Amy K. Karlson, Greg Smith, Bongshin Lee. Which Version is This?: Improving the Desktop Experience within a Copy-Aware Computing Ecosystem. Proceeding of the twenty-nineth annual SIGCHI conference on Human factors in computing systems, Vancouver, BC, Canada. May7-12, 2011.

[6] Ravasio P., Schar S.G. and Krueger, H. In pursuit of desk-top evolution: User problems and practices with modern desktop systems. ACM TOCHI 11, 2 (2004), pp156-180.

[7] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In DSN'10, June 2010, pp181–190.

[8] Huang CQ, Xu FY, Hu XY. Massive data oriented replication algorithms for consistency maintenance in data grids. ICCS 2006, Part I, LNCS 3991, pp 838–841.

[9] Demers A. J., Petersen K., Spreitzer M. J., Terry D. B., Theimer M. M. and Welch B. B. The bayou architecture: Support for data sharing among mobile users. In Proceedings IEEE Workshop on Mobile Computing Systems & Applications (Santa Cruz, California, August-September 1994), pp. 2–7.

[10] Agrawal N., Bolosky W.J., Douceur J.R. et al. A five-year study of file-system metadata. ACM Trans. Storage 3, 3 (2007), 9:1-9:32.

[11] J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In Thirteenth ACM Symposium on Operating Systems Principles (Asilomar Conference Center, Pacific Grove, U.S., 1991), vol. 25, ACM Press, pp. 213–225.

[12] W. Vogels. Eventually consistent, Commun. ACM 52(1) (2009) 40–44.

[13] T. Kraska, M. Hentschel, G. Alonso, D. Kossmann. Consistency rationing in the cloud: pay only when it matters, Proceedings of the VLDB Endowment (PVLDB) 2(1) (2009) 253–264.

[14] E. Anderson, X. Li, M. Shah et al. What consistency does your key-value store actually provide? Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[15] Windows Live Sync. http://sync.live.com/

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

[17] DBank. http://www.dbank.com/

[18] J.H.Howard.Using reconciliation to share files between occasionally connected computers. Proceedings IEEE Workshop on Workstation Operating Systems,Napa,California,October 1993,pages 56-60.

[19] A. Lakshman, P. Malik. Cassandra - A decentralized structured storage system, in: 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.