

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik

Masterarbeit

Data Synchronization and Local Data Access across Peer-to-peer Social Networks

Bing Gao
Matrikelnummer: 6643802
E-Mail: binggao@uni-paderborn.de

Paderborn, den 8. Mai 2015

vorgelegt bei
Prof. Dr. Christian Scheideler
Jun.-Prof. Dr.-Ing. Kalman Graffi

Ehrenwörtliche Erklärung

Michael SzydloHiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den 8. Mai 2015

Bing Gao

Inhaltsverzeichnis

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Problem Definition . . . 2
1.2.1 Definition of synchronization application . . . 2
1.2.2 Which kind of synchronization application? . . . 2
1.2.3 Data storage . . . 4
1.2.4 Mechanism of synchronization Process . . . 4
1.2.5 Summary of problem definition . . . 5
1.3 Goals . . . 6
1.4 Outline . . . 7

2 Background **9**

2.1 Previous Work on Data Synchronization Application . . . 9
2.1.1 Dropbox . . . 9
2.1.2 FreeFileSync . . . 9
2.1.3 Allway Sync . . . 10
2.2 Current System Environment . . . 10
2.2.1 Pastry overlay . . . 10
2.2.2 Framework . . . 10
2.2.3 Plugin/GUI layer . . . 10
2.3 Challenges of Implementation a Sync Application in the p2p social network . . . 11

3 Algorithms **13**

3.1 Rsync – Synchronizing between Two Files . . . 13
3.1.1 Definition . . . 13
3.1.2 Algorithm principles . . . 13
3.1.3 Core algorithms applied to Rsync . . . 14
3.2 Merkle Tree – Synchronous Data Consistency Algorithm . . . 17
3.2.1 Definition . . . 17
3.2.2 Hierarchical computation . . . 18
3.2.3 Retrieving a merkle hash tree . . . 19

4 Implementation **23**
4.1 Decisions on Algorithm and Available Tool . . . 23
4.1.1 Rsync . . . 23
4.1.2 Jarsync . . . 23
4.2 Strategy Design . . . 24
4.2.1 Deal with user online status . . . 24
4.2.2 Deal with missed sync content during user's offline period . . . 25
4.2.3 Owner view syncing process . . . 25
4.2.4 Participator view syncing process . . . 26
4.3 Strategy Implementation . . . 27
4.3.1 Implementation of Rsync . . . 27
4.3.2 Modification detection with DLL . . . 28
4.3.3 Combination of Rsync and Merkle Tree . . . 30
4.4 User Authorization to Access a Sync Object . . . 38
4.4.1 Join sync group by invitation . . . 38
4.4.2 Request to join a sync group with *group ID* . . . 38
4.5 Data Structure Implementation . . . 40
4.5.1 Core data stored in the framework . . . 40
4.5.2 Data structure used in syncing process . . . 40
4.6 Data Transmission via Framework . . . 41
4.6.1 Sync content transmission . . . 41
4.6.2 Big file transmission . . . 41
4.7 Web View Implementation of File Sync Status . . . 44
4.7.1 Sync group list view . . . 44
4.7.2 File sync status view . . . 44

5 Evaluation **47**
5.1 Outcomes . . . 47
5.1.1 User activities . . . 47
5.1.2 Consistency between outcomes and goals . . . 48
5.2 Evaluation Environment . . . 48
5.3 Illustration of Test Cases . . . 49
5.4 Data Set . . . 51
5.4.1 Modifying file . . . 52
5.4.2 Adding file . . . 52
5.4.3 Deleting file . . . 53

6 Conclusion **55**
6.1 Summary of Work . . . 55
6.2 Future Work . . . 55

Abbildungsverzeichnis **57**

Literaturverzeichnis **59**

1 Introduction

On account of some remarkable advantages, such as security, decentralization and extend-ability, peer-to-peer (p2p) technology has been more and more popular in recent years. Some p2p systems were developed in various utilities like file sharing, communication system and social networking. Through the joint efforts of many people, now we have a p2p-based framework which provides the basic functionality for social networking. Many plugin-based applications, like friends, chart, group and so on, are implemented base on the functionality provided by the framework layer. Nowadays, the user demand of social networking has never stopped from growing. In order to meet the various needs of users, and to provide a interesting and convenient way for data sharing, a new application for data synchronization is implemented based on an existing p2p-based social network. This application allows user to specify a local folder from his/her local disk, and share it with other users in the network. Anyone in the sync group can make changes (add/delete files, edit existing files) on their specified local sync directories. Then these changes can be detected and synchronized among different user machines.

1.1 Motivation

A data synchronization application, is useful not only for interactions within one single user on different machines, but also for interactions among a group of users. It provides an approach to local data accesses, manages user authorization for the synchronized object and synchronizes among databases with real-time data updates. We believe that a good synchronization application can greatly facilitate people's lives. Imagine that if there were no synchronization applications such as Dropbox or Allway Sync, you could only transfer data use other ways like using USB drive or sending compressed files via E-mail. The convenience of data transmission would be limited to a large extend. When it comes to social networking, data synchronization adds more fun for users who want to share local data and keep to be updated from others. For example, a user wants to share slides, lecture-notes and self-summarized materials of some lecture which locates on his own local disk with his classmates, he wants the others to make some correction if there is any mistakes related to the content, and better the others also share their materials which could be helpful. In this case, a fast, accurate synchronization tool plays an important role in his friends circle. As far as I know, some existing popular social networks like Facebook, Twitter or Instagram do not provide such kind of functionality which allows users

to synchronize data among different machines or with other users. Therefore, the idea of implementing a synchronization application in our existing p2p-based social network became an exciting decision.

1.2 Problem Definition

1.2.1 Definition of synchronization application

When we decide to implement a data synchronization application, the first problem should we consider is, what is synchronization application.
A synchronization application should have the ability to synchronize multiple types of files among multiple peers. The structures of these files and sub-folders under multiple sync directories should keep consistent. According to user's operations, data modification can be detected, the sync content can be calculated fast and synchronized to the corresponding files on all other remote sync directories.

1.2.2 Which kind of synchronization application?

The second important problem to be considered before implement a synchronization application is, which kind of synchronization application it should be. Basically there are 3 kinds of classifications.

1-way-sync or 2-way-sync

1-way-sync is synchronization process in which files are expected to be changed in one certain local directory (source) only. In other remote directories (targets), files can only be synced unconditionally. The source has read/write access to targets, but targets have no read/write access to source. This synchronization process always proceeds in one direction.

2-way-sync is also known as bi-directional synchronization. In the process of 2-way-sync, any local directory can have read/write access to others. Each directory can be assigned as a source as well as a target. When a file under any local directory has been modified, the directory is set as source, other remove directories are target. Files in all directories are updated in both directions.

In our social networks, in order to motivate users to participate in interactions, we choose to implement a 2-way-sync application. Therefore, users should be able to get the updated files from others, at the same time, they can also edit the files under their local directories and upload the changes to update the remove directories.

1-to-1 sync or 1-to-more sync

In 1-to-1 sync process, at any point of time, the sync content which is calculated from source side is transferred from one source to one target at most. In other words, the source can be only occupied by one target at most for data transmission. The other directories can only wait until the source are released from the current target.

Diversely, there can be more targets which are being synchronized by a source at the same time in 1-to-many sync process. Once the sync content is calculated and generated from the source side, it is transferred simultaneously to all the other remote directories. Then the remove directories update the files under them with the received sync content. So in 1-to-many sync process, all targets receive the same sync content and update their files with it.

In our p2p system environment, data is transferred from one peer to other peers through the framework. The sync content is generated by comparing the file on source side (local file) and the corresponding file on each target side (remote file). This comparison process should be 1-to-1 because these remote files may differ from others. Thus the sync contents generated by comparing a source with different targets may be different. Each of these sync contents should be used to update the corresponding remote file. Hence 1-to-1 sync process is applicable in our situation. This 1-to-1 sync process can be extended to 1-to-more sync process if only we repeat it until all the remote directories receive their sync contents. This is the way we organize multiple peers to synchronize data.

Sync via server or Source-target-role alternating

Synchronizing data via server, is the mechanism of most existing synchronization applications. There is always a whole backup of synced database in server. Whenever user edits his local sync directory, the sync content is calculated by some algorithm and uploaded to server. From there the sync content is transferred to remote directories. Some sync applications such as Dropbox have improved this mechanism by directly transferring sync content from source to targets and server. Thus the targets can get updated faster. But the main principal has not been changed: a real server structure is needed to backup databases and to coordinate the synchronization process among directories.

A source-target-role alternating synchronization process uses the mechanism of setting roles automatically according to user's operation. It does not require a server structure. If any local directory under which data has been modified, it is set as source. Other remote directories are set as targets. The sync contents will be generated and transferred among the source and the targets. The roles of source and target are alternating among all the directories. Whenever there is data modification has been detected, the source-target-role setting up mechanism is activated. This kind of synchronization is very flexible. It implemented a self-management mechanism among directories. No server structure is necessary.

Based on the above advantages, source-target-role alternating sync became the choice for our synchronization application in the p2p social network. This is precisely what the feature of p2p system. Any peer itself is a supplier and consumer of resources, without the need of centralized coordination by a server structure or stable host.

According to the above discussions, a 1-to-1, 2-way and source-target-role alternating synchronization application meets our requirements.

1.2.3 Data storage

In our p2p-based social network, applications use a decentralized infrastructure to store the encrypted data of the users. To implement a synchronization application in this system, an important problem should also be considered: data storage for synchronization process.
As we do not have a real server structure which can backup the database and coordinate the synchronization process in our p2p-based social network, the data storage problem needs to be considered according to the system environment. In order not to increase the burden of data storage in framework, we should not store any part of the database which is being synced in the framework. Only authorization data which contains user IDs of a sync group as well as their corresponding sync directories should be stored in the framework. Additionally, the information of synced folder should also be stored and displayed on GUI.

1.2.4 Mechanism of synchronization Process

A good 2-way synchronization application in a p2p social network should have the following features:

- Any peer in a community has equal rights from the view of modifying the synced database.
- Modification of files under directories should be able to be detected fast and accurately.
- The files should be able to get updates from remote directories even it is currently being used or modified under the local directory.
- Support for reading/writing from/to directories which locate on different operating systems.
- The sync content missed during user's offline period should be detected and transferred right after user logs in again.

In order to achieve these features, a mechanism is considered as follow:

1. Users participate in syncing a folder are organized in a group. The user IDs contained in the group are authorized to access the synced database.
2. Modification on local directory is detected by comparing the old and new values which can represent the states of the files. The structure of the value should be reasonable to indicate the folders and files structure under each directory.
3. Each local file which is detected to be modified is compared with the corresponding files under other remote directories by an efficient algorithm. The calculated outcomes by the algorithm is the sync content which needs to be transferred to the remote directories.
4. After receiving a sync content, the corresponding file under each directory starts to be rebuilt with the sync content. Thus the file is updated.
5. For different operating systems, the representations of local paths are different. Therefore, users should be able to specify there own paths to where the synced folder will be located.
6. As for retrieving the missed data modification during logged out period, one of other online peers which is in the updated status, its sync database is selected to be compared with the one of which the peer has just logged in again. The modified files should be detected, sync content should be generated and transferred to the newly logged in peer. In this way, the missed sync content can be retrieved even user logs out for a while.

From the above points, we know that to implement this synchronization mechanism, two core problems need to be solved: building a status value structure for a directory; And with this building method, the files with different values should also be easily found. An efficient algorithm which is used to calculate sync content by comparing two files under two directories is necessary. Thus, the choice and implementation of an algorithm became the primary task in the implementation of data synchronization application.

1.2.5 Summary of problem definition

The problem definitions involved in implementing a synchronization application are summarized as:

First of all, the application we are implementing is a 1-to-1, 2-way and source-target-role alternating synchronization application.

Secondly, we do not backup the whole database in any server structure, not even part of it. Only authorized groups which contain user IDs and the specified paths are necessary to be stored in the framework.

Then a mechanism with which we can build status values for directories and find the modified files by comparing these values is needed to be implemented.

Additionally, an efficient algorithm needs to be selected and applied into the synchronization process. In order not to increase the burden of data transformation through framework, the size of sync content which is calculated by this algorithm should be as small as possible.

Finally, to consider the user behaviors in a social network, the sync content which is missed during user logs out period needs to be calculated right after user logs in again.

1.3 Goals

After defining the problems, we need to consider what exactly functionality should the synchronization application have. What operations can users make and what effects will be there after making these operations. Here we illustrate a use case diagram to explain the goal functionality of this synchronization application. In this data synchronization application, users are classified as two identities: owner who initiates to sync a folder and creates a sync group, and participator who accepts invitation from owner or who is allowed to join a sync group after requesting. One owner together with one or more participants constitute a sync group .

As shown in Abbildung 1.1:

Owner: after log in, the owner can specify a local folder, create a new sync group and invite other users. For every user, there is a sync group list which contains information of all the sync groups the user has created or participated. The owner can check status of each synced folder from the list, reset local path for any synced folder, modify data (add new files, delete files and edit existing files) under his/her own local path and synchronize these modifications with other online users. For the owner, deleting sync group from list means deleting the sync group from framework. Participants will not be able to synchronize data after the owner deletes the sync group. Of course the owner can also invite more users from his friend list by sending them invitations. If the owner wants to stop synchronizing data with some people in the sync group, he can also remove them from the group. Then the removed users will not get any updates.

Participator: the users who have accepted the invitation from the owner are added to the sync group as participators. For users who are not invited by the owner but still they want to join a sync group, they can send request the owner with knowing the sync group ID. They are added to sync group only after the owner accepts their requests. Same with the owner, participators are able to check status of each synced folder on their sync group lists, reset local path for any

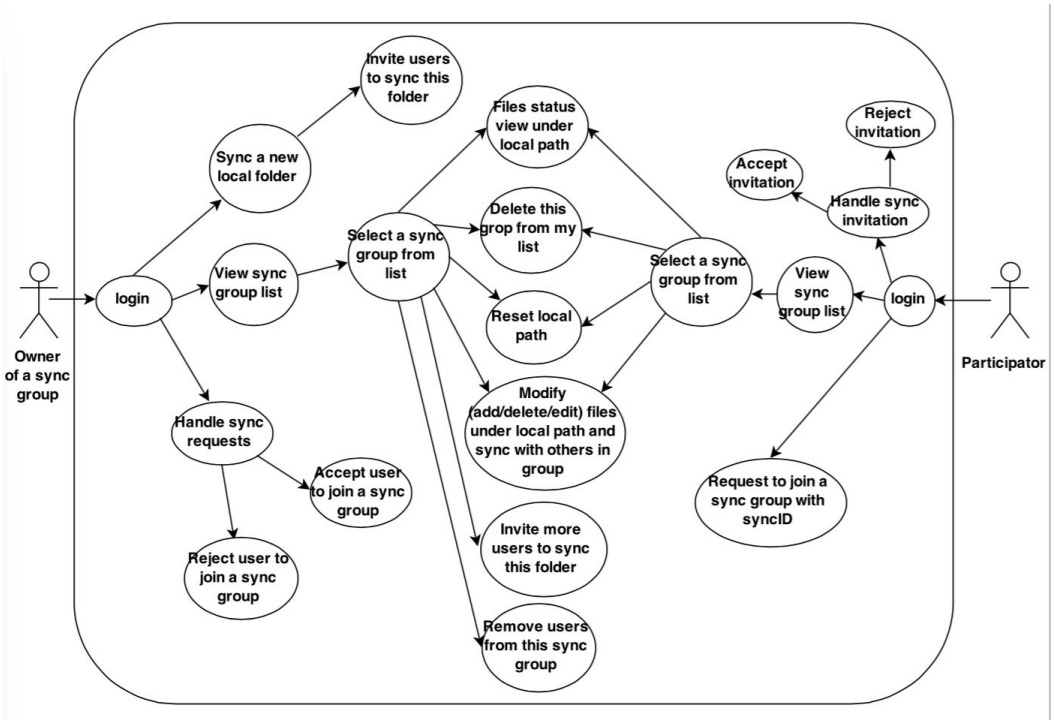


Abbildung 1.1: Use Case Diagram of Synchronization Application in a p2p-based Social Network

synced folder, modify data under their own local paths and synchronize these modifications with other online users all over a sync group. But differs with the owner, if a participator deletes a sync group from his/her sync group list, s/he only leaves from the group, that means s/he is stopped from synchronizing the local folder with other users, but the sync group still exists in the framework, other users can still participate in synchronization.

This is basically the goal functionality of our data synchronization application. To summarize, our goal in this implementation is to provide a simple user-friendly interface through which users can achieve data synchronization functionality by making appropriate operations.

1.4 Outline

This paper is intended to provide a detailed introduction about how did we implement a data synchronization application across the p2p-based social network, what functionality dose it provide, what technologies has been applied and so on. The structure and outline of this paper are organized as below:

In chapter 1, we mainly discuss about the background knowledge, some previous work on data synchronization, the current environment and achievement of the p2p-based social network and challenge of building our own data synchronization application across the system.

In chapter 3, we discuss two high performance algorithms used in data synchronization and management.

Chapter 4 describes the implementation process during the development.

In chapter 5, we evaluate the application with achieved results, test cases and some samples of data sets.

Finally in chapter 6, we conclude the thesis from two aspects: summary of the work in this thesis and some more work need to be done in the future.

2 Background

In this chapter, we mainly describe the background knowledge involved in the process of implementing a data synchronization application. It will be discussed in the following sequence: first of all, we introduce some successful applications as previous work on data synchronization technology. Then the current system environment of our p2p-based social network is analyzed. At last, we discuss about the challenges of implementing a data synchronization application according to the current situation.

2.1 Previous Work on Data Synchronization Application

Data synchronization technology is applied to synchronize data among multiple devices or users. The files or folders under the synced directory are automatically copied to remote directories. Soon afterwards the modification made under each directory can be detected and synchronized with other directories. There are many synchronous software which have been developed based on data synchronization technology. Among them, we give brief introduction for Dropbox, FreeFileSync and Allway Sync.

2.1.1 Dropbox

Over the years Dropbox has been the first selection of data synchronization and backup for many users. It has also been recognized as one of the most popular cloud storage service. It provides services like data synchronization among different devices which signed up with a same user ID and sharing folders with other users by sending them links. Various clients are available for different operating systems. The most important characteristic of Dropbox is easy to use. A powerful server structure provides guarantee for database backup. When user modifies database on any machine, the difference is generated by RSync algorithm. If there are other online machines signed up with the same user ID, difference will be synchronized to other machines. At the same time, the database backup on the server is also updated.

2.1.2 FreeFileSync

FreeFileSync is a free open source software. It also provide files and folders synchronization service for users. It is a program which compares files in databases on different machines or directories, copy and sync files among these databases. It

keeps versions of deleted and updated files, in order to restore data. It can detect the conflicts and renamed files or folders. There is no server structure which can keep a backup for the whole database. Files are replaced/copied/deleted from source to target.

2.1.3 Allway Sync

Allway Sync is folder synchronization software similar with FreeFileSync. In addition to providing data synchronization service, Allway Sync also offers support for FTP, scheduling and automatically syncing. This software can be installed on as many computers as user needs. It realized multiple-folder synchronization by tracking a database for file modifications and deletions.

2.2 Current System Environment

Before start developing an application, we first give a brief introduction for the current system environment in this section. The existing environment of our p2p-based social network is described from the following aspects: pastry overlay, framework and technologies applied in plugin/GUI layer. The system provides very powerful functions from these aspects. But we do not give detailed introduction here. Only the related technological support for building our new application is described.

2.2.1 Pastry overlay

Overlay is built based on FreePastry. It provides the basic functionality of peer-to-peer system: generating node IDs, port forwarding with NAT, building/join a ring.

2.2.2 Framework

The framework layer provides the kernel functionalities of social networks. It provide support for data storage, access control, security management, message dispatcher and App store which manages plugin/GUI layer.
The mechanism of creating encrypted items which support an access control to users are available in data storage. This encrypted item is *SecureStorageItem*. And data structures like *DistributedLinkedList* have been implemented. *DistributedLinkedList* can be used to represent a collection class list in a distributed manner.

2.2.3 Plugin/GUI layer

The Plugin/GUI layer is the top layer in the social network architecture. Many applications like groups and chart are built based on the Plugin/GUI layer. Each plugin of these applications is OSGi-based.

Technologies used in GUI

The technologies used in GUI are:

- HTML5: user interface.
- AJAX: used to create interactive web applications.
- JQuery: javascript libary, is compatible with CSS.
- Bootstrap: Front-end development framework for HTML, CSS and JS.
- Knockback: Communication of information between the large amounts of data models.

Web provider and Rest handler

Web provider is a plugin which allows to access the API of other plugins. REST Handler is a class used by the Web provider plugin to access API by using AJAX call.

2.3 Challenges of Implementation a Sync Application in the p2p social network

The challenges of implementing a data synchronization application are:

1. There should not be any centralized server structure used to manage synchronization process.
2. No backup of synced database stored in framework.
3. Quick detection of which node made the modification on his database.
4. Quick detection of which find exactly under the directory has been modified.
5. Size synced content transferred via framework should be as small as possible.
6. Recover the sync content missed by user during log out period.

Summary of the Chapter
In this chapter, we have illustrated some successful previous work on implementing a synchronization application, the current system environment which could be used by us to develop a new plugin is analyzed, challenges of building a sync application based on this p2p system are described. As discussed in section 1.2 of chapter 1, an efficient algorithm plays a central role during the data synchronizing process. Thus, we will introduce two efficient algorithms used in data synchronization application in the next chapter.

3 Algorithms

In this chapter, we mainly we enumerate the algorithms which are considered to be applied into our application.

3.1 Rsync – Synchronizing between Two Files

Rsync algorithm is used for generating differences by comparing two files, then apply these differences to update a remote file. This algorithm was invented by Andrew Tridgell for synchronizing files under Unix/Linux. Rsync is also applied to the synchronizing process of Dropbox. It minimizes the amount of data transmission through network for synchronizing files. In this section, we do not introduce how to the use Rsync algorithm, only describe the principles of the core algorithm and problems can be solved by it. How to use the algorithm in detail and the implementation method will be introduced in the next chapter which elaborates the implementation process. The content of this section is referenced from the paper of Tridgell (1998).

3.1.1 Definition

First of all, let's have a look at the problem to be solved by Rsync. There are two similar files **A** and **B** on two different machines α and β. Now we want to synchronize file **B** according to the content of file **A**. So file **A** denotes the source and file **B** denotes the target. Of course, we could directly copy file **A** from machine α, transfer it to machine β and replace file **B**. But our intention is to minimize the data transmission through network. We only want to send the different parts of **A** compared to **B**. So we need to compare these two files to get the diff. But these two files are on two different machines. They can not be compared without transferring one of them onto the machine of another. And we do not want to transfer the whole file cause this goes against our original intention. Thus, there must be some way, let these two files be compared without transferring any one of them. Therefore Rsync algorithm was invented.

3.1.2 Algorithm principles

Based on the precondition (two files on two machines) of the problem definition in last subsection, the principles of Rsync algorithm are described as the following 5 steps in sequence:

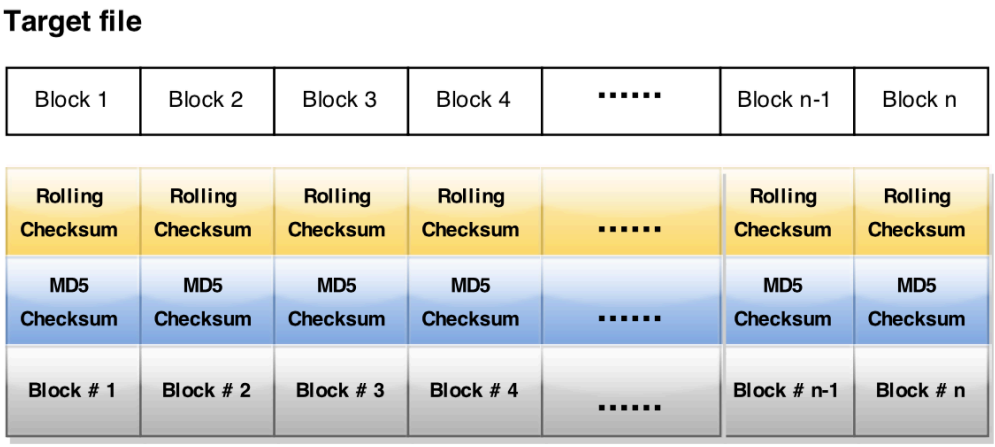


Abbildung 3.1: File Block Diagram (quoted from the blog from Hao (2012))

1. Split file **B** into same size data blocks of size **S** bytes from β side.
2. For each data block of **B**, calculate two checksums: a weak rolling checksum and a strong MD4 checksum (described later).
3. β sends these checksums to α.
4. α checks file **A** for every **S** bytes (at any offset) which have the same weak and strong checksums as a block of file **B**. The exclusions the different blocks.
5. α generates different blocks and sends them to β. β rebuilds file **B** with these diff blocks.

3.1.3 Core algorithms applied to Rsync

Algorithms applied in Rsync are as follows:

1. Rolling checksum and MD4 checksum of a data block
   In step 1, we first split the file into blocks, for example each block 512 bytes (the last block is smaller than 512 bytes). Then we get checksums from each block by doing two kinds of calculations:
   a) Rolling checksum: weak checksum, a 32 bits checksum generated by algorithm adler32 invented by Mark Adler.
   b) MD4 (Message-Digest Algorithm 4) checksum: strong checksum, a 128 bits checksum generated by algorithm MD4 (now is MD5) hash.

The Abbildung 3.1 shows the file blocks and checksums information of target file. Why do we need two kinds of checksums? Because only checking the weak checksum generated by adler32 is not accurate enough, there is a high probability that the same checksum can be generated from two different blocks. So it is necessary to use a strong algorithm to make sure that these two blocks are same. That is to say, if the weak checksums are different, of course these two blocks are different, otherwise we have to check the strong checksums, if the strong checksums are also same, then only these two blocks are same. In a word, adler32 is for recognizing the different blocks, MD4 is for finding the same blocks.

2. Transformation of checksums
   After β computes the checksums for all blocks of file **B**, these checksums are organized in a list. Three parameters for each block are contained in the list: block number, weak rolling checksum and strong MD4 checksum. Then this list will be sent from β to α.

3. Checksum searching
   Once the target (α) receives the checksum list, it stores this list into a 16 bites hash table. Use rolling checksum to hash. The length of the hash table is 2^{16}, so the hash of rolling checksum is an integer value between 0 and $2^{16} - 1$.
   Here by the way, we would like to explain, it was said to sort the rolling checksums in some articles. These articles are all quoted from the original author, but there are misunderstandings. It was not to sort the rolling checksums, but to hash these rolling checksums of the target file to a 2^{16} long hash table.

4. Comparison
   This is the key algorithm used on source file.

   a) Select the first block (we assume 512 bytes) which contains from the first byte to 512th byte of the source file, do rolling checksum calculation. Search the calculated result in the above hash table.

   b) If it is found, it is possible that the target file contains a same block with source file. Then compare the MD5 checksums. If MD5 checksums also match, this block is same with target block, record the block number.

   c) If it is not found from the hash table, no need to compare MD5 checksums, this block contains different data compared with the block of target file. Then algorithm goes to *a*), backward offset 1 byte, select the block which contains from the second byte to 513th byte of the source file, continue to calculate rolling checksum and search the result in hash table.

   d) Iterate this process until all the same blocks were found. Then the other blocks in hash table are the diff blocks need to be transferred to machine β.

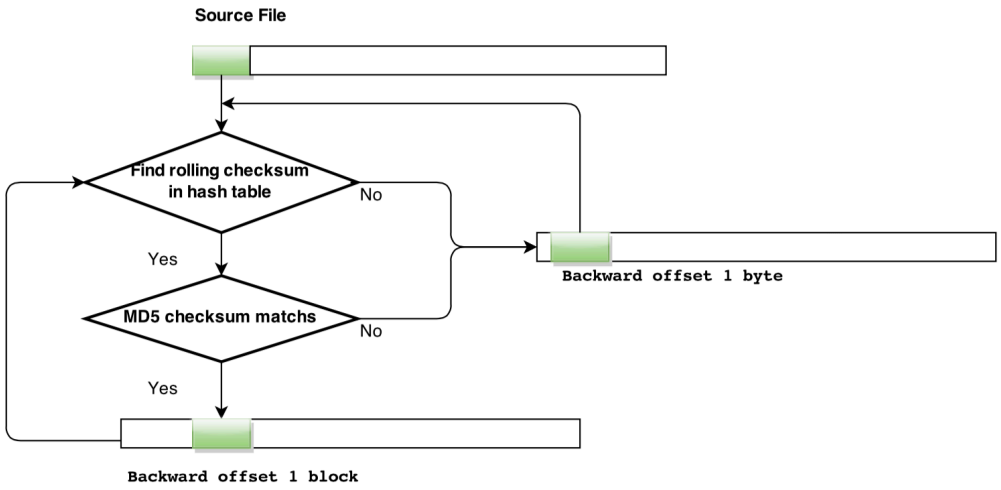


Abbildung 3.2: Blocks comparison: rolling checksum (quoted from the blog from Hao (2012))

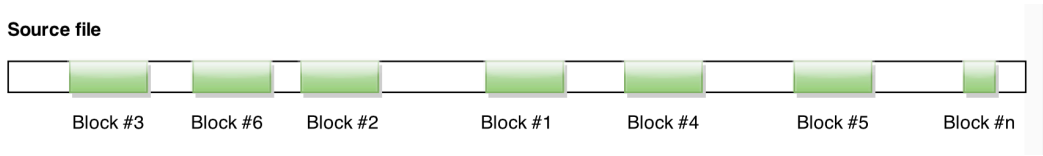


Abbildung 3.3: The same and diff blocks of source file compared to target file (quoted from the blog from Hao (2012))

This process is shown in Abbildung 3.2:
So in this way, all the same blocks of target file **B** contained in source file **A** are found out, the left scattered blocks are the diff content needs to be sent to β. Abbildung 3.3 illustrates the same blocks (green) and excluded the diff content.

Finally, at the source file side, we get a list of diff data blocks by Rsync algorithm. Then the source machine α compresses the list and sends it to machine β. After receives the list, β rebuilds file **B**. The synchronization is completed.

More details of Rsync algorithms are in the original paper from Tridgell (1998). It is also available online from site: https://rsync.samba.org/tech_report/.

3.2 Merkle Tree – Synchronous Data Consistency Algorithm

Merkle Tree algorithm is for checking the consistency of synchronous databases. It was named after Ralph Merkle who patented it in 1979. This algorithm can be used to build a hash tree data structure which is used to verify any kind of data modification. It provides a efficient and secure mechanism to maintain consistency of multiple databases. The most pivotal two steps contained in Merkle Tree for data synchronization among databases: building Merkle Hash Tree and retrieving the Merkle Hash Tree. In this section, we first define the problem can be solved by Merkle Tree. Afterwards we elaborate algorithm principles. Then we give introduction for the mechanism of building and retrieving a Merkle Hash tree. The content of this section is referenced from the paper of Szydlo (2004).

3.2.1 Definition

Synchronizing a data collection among multiple nodes is a common case for synchronization applications. For example, we synchronize a collection of files and folders under a root folder among machines. In order to maintain the consistency of these files, we should regularly do comparison to these databases, find the modified files and update the corresponding files on other machines. The following problems need to be solved: how to detect the existence of data modification in a database, how to quickly find out the modified data and how do we keep the consistency of the database structure.
Here we introduce a definition: **data block**.

A **data block**: the unit which can be detected to be modified and transferred for data synchronization. A set of data blocks are generated by splitting files by specifying a block size.

Actually the problems need to be solved are related to each other. If we are able to manage the database structure by a reasonable mechanism, generate value for each data block and folder, then by analyzing the values we can know if there is data block has been modified in the database and which one exactly is it has been modified.
Here we use hash to label the value of each data block. According to the structural characteristics of files and folders under a directory, we use a tree structure to store the hash values of data blocks of files or folders under the directory. Such a tree structure is called **Merkle Tree**.

Merkle Tree: a hash tree structure, each leaf node in Merkle Tree is labeled with a hash value, hash of each non-leaf node (parent) is labeled with the hash of the labels of all its children nodes. Hash values of nodes are labeled sequentially from bottom to up, until the root node is reached. Hash value of the root node is the root hash of this Merkle Tree. By detecting these hash values, we can verify data modification under the directory.

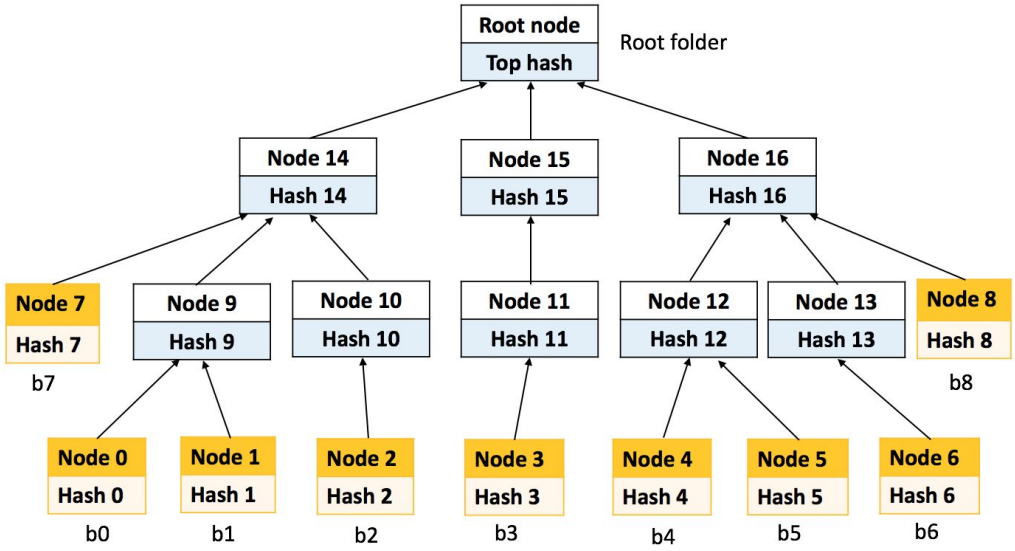


Abbildung 3.4: Merkle Tree hierarchy of data blocks and folders under a root folder

3.2.2 Hierarchical computation

For better understanding, we assume there are two similar databases **A** and **B** on two different machines α and β. Under the root folders of **A** and **B** respectively contain 9 data blocks. These blocks are **b0**, **b1**, **b2**, ..., **b8**. The hierarchy of these data blocks and folders can be displayed as Abbildung 3.4.
Based on the above assumption, the principles of computation in a Merkle Tree are described as follow:

1. This data structure is tree, it can be binary tree or multi-way tree (here we analyze the case of multi-way tree).

2. Every leaf node of Merkle Tree is labeled with a hash of the corresponding data block.

3. Every non-leaf node of Merkle Tree, its value is a hash of all its children' values.

As shown in Abbildung 3.4, value $v0$ of leaf node 0 is hash 0, value $v0$ of leaf node 1 is hash 1. The value $v9$ of parent node 9 is computed as: $v9 = hash(v0, v1)$. By such way, the value $v14$ of parent node 14 is $v14 = hash(v7, v9, v10)$. Then the value of

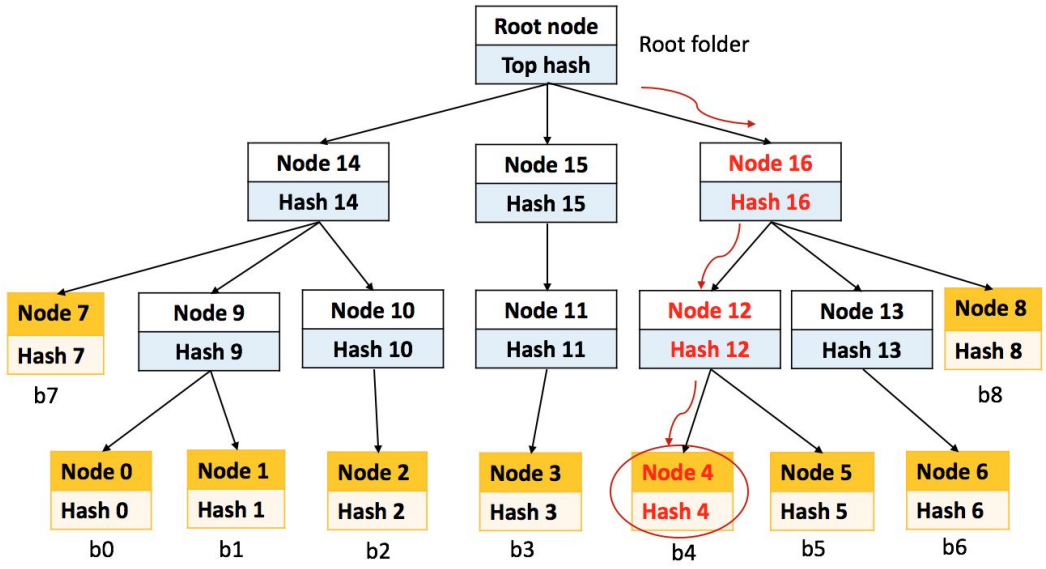


Abbildung 3.5: Comparison and retrieval process to a Merkle Tree

root node is $tophash = hash(v14, v15, v16)$. This is the hierarchical computation relationship in Merkle Tree.

3.2.3 Retrieving a merkle hash tree

If there is any data block modified on machine α, how do we retrieving it to find the modified data block by comparing two Merkle Trees on α and β? This comparison and retrieval process is shown in Abbildung 3.5.
For example we modify data block $b4$ on machine α. The retrieval method is described as follow:

1. First compare top hash of two Merkle Trees. If top hash are different, retrieve their children: nodes 14, 15 and 16.

2. Only value $v16$ of node 16 is different. Retrieve its children: nodes 12, 13 and 8.

3. Only value $v12$ of node 12 is different. Retrieve its children: nodes 4, 5 and 6.

4. Only value $v4$ of node 4 is different, and node 4 is a leaf node. Access the information of data block $b4$.

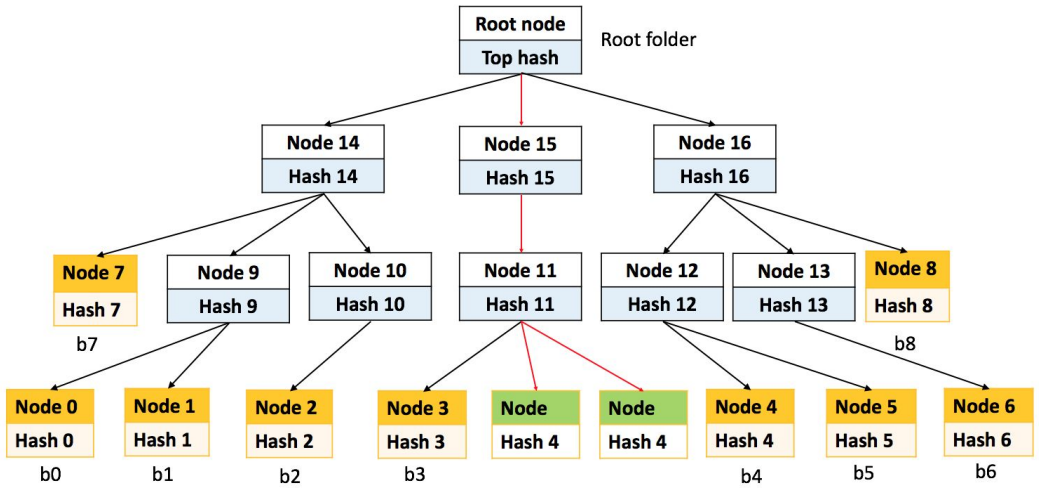


Abbildung 3.6: Retrieve a Merkle Tree to find the blocks of added file

5. Retrieving ends.

If machine α adds a file under the root folder, for example add a file which contains two data blocks under node 11 which represents a folder, the comparison retrieval process is shown in Abbildung 3.6:
Similarly, the deleted data block can be retrieved in Abbildung 3.7:
The above processes are the methods to find the modified, added and deleted data blocks by comparison and retrieving Merkle Trees. The next step is transferring the modified and added blocks from machine β to machine α, rebuild database B with these blocks and delete the blocks with has been deleted from A on machine β. This is the way to keep consistency of two databases on machines.
From the above description, it is learned that for synchronizing a large size database, using Merkle Tree algorithm can improve the efficiency of data verification.

Summary of the Chapter

Two efficient algorithms Rsync and Merkle Tree are introduced in this chapter. Both of them have their own advantages from different aspects. Rsync accurately calculates the difference of two files and transfer it between files, the amount of data transmission is minimized by Rsync algorithm. Merkle Tree algorithm provides a very efficient way to manage a large database and find the modified data blocks

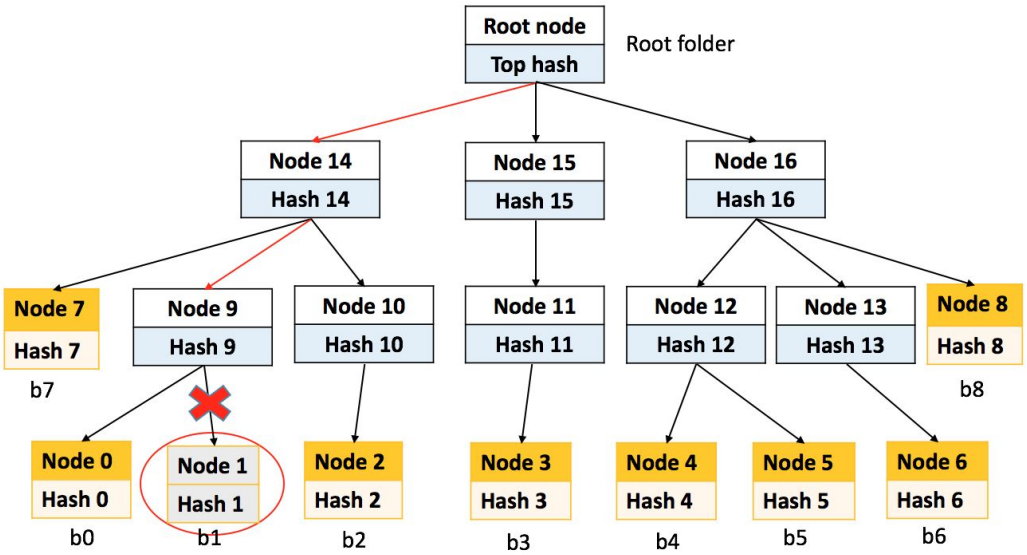


Abbildung 3.7: Retrieve a Merkle Tree to find the deleted block

by comparing the local tree and the remote tree. The choice between these two algorithms will be introduced in details in the next chapter: chapter 4.

4 Implementation

In this chapter, we describe in detail the implementation process of the data synchronization application. The process is discussed from the following aspects: selection of algorithm and tool, proposed strategy design, implementation of the strategy, implementation of data structure, data transmission via framework and file status web view implementation.

4.1 Decisions on Algorithm and Available Tool

4.1.1 Rsync

We introduced two kinds of algorithms in Chapter 3. Rsync can be used for generating diff content which is used to rebuild the target file. Merkle tree organizes data blocks of files in a tree structure, updates the target database by replacing the modified data blocks, adding new data blocks and deleting non-existent data blocks according to source database. From the discussion in section 1.2 in Chapter 1, we know that the application we need to implement is a 1-to-1, 2-way and source-target-role alternating synchronization application. In the initial stage of data synchronization (after invited users accept invitation), the synced database is copied from source machine to target machines by data transmission through framework. After the initial stage, users start to modify the files under their own directories. In order to achieve real-time updates on these databases, we should detect changes on source machine and send updates to target machines frequently. This decreases the dependency on the data transmission capability of the framework. In order to increase the transmission speed of the framework, the size of diff content transferred among machines should be as small as possible. Rsync algorithm meets this condition. The diff content generated by rolling checksum in Rsync is much smaller than the diff data blocks in which are statically spitted from a file in Merkel Tree algorithm. If we can manage each file in synced database properly, so that other corresponding files can be found on other machines, Rsync algortihm is a good choice to be applied into the syncing process of each pair of files.

4.1.2 Jarsync

In the last subsection, we decided to apply Rsync algorithm in our sync application. Now let us search for a suitable Java implementation of Rsync algorithm. Jarsync

is a implementation of Rsync algorithm. It provides a compression library for Java platform. Jarsync is covered under the GNU (General Public License) which guarantees our freedom to use it to make our own software.

About Jarsync

The Jarsync library implements Rsync algorithm, provides simple API for using it. It is used to synchronize two files under two directories on different machines. It implemented a client-server-role allocation method. The client sends its checksum list to server. Server compares delta according to the checksum list and send delta to client. Client rebuild the file with the delta. This is the sync process implemented in Jarsync. What we need to do is implementing a roles allocation mechanism which can automatically detect which file under which directory has been modified, assign it as server, other corresponding files are targets, call the method of Jarsync API, let these files do 1-to-1 sync in a loop.

The API

The API of Jarsync library provides three basic functions:

1. Checksum Generation
2. Delta Generation
3. File Patching

In additionally to these three functions, the API of Jarsync also provides streaming implementations which are used for data transmission. This is implemented by a event-listener-streaming form for each operation.

4.2 Strategy Design

To achieve the goal shown in Abbildung 1.1 in the section 1.3 of Chapter 1, here we illustrate two diagrams which describe about working mechanism of this application from the views of owner and participator of a sync group.

4.2.1 Deal with user online status

When users are offline, their files under directories should not be able to be synced. Only the online users out of the sync group can participate syncing. Thus, a user status detector is used to manage the online users out of a sync group. At any point of time, if there is user modifies the database under hie/her sync directory, the online users out of the sync group should be found quickly. Then the machine who did modification takes the server role, other online machines are targets. The server select one from these targets, do 1-to-1 sync with Jarsync API. Afterwards, other un-synced users are synced in a loop until all these online users are synced. So the users inside a sync group should be classified as shown in Abbildung 4.1.

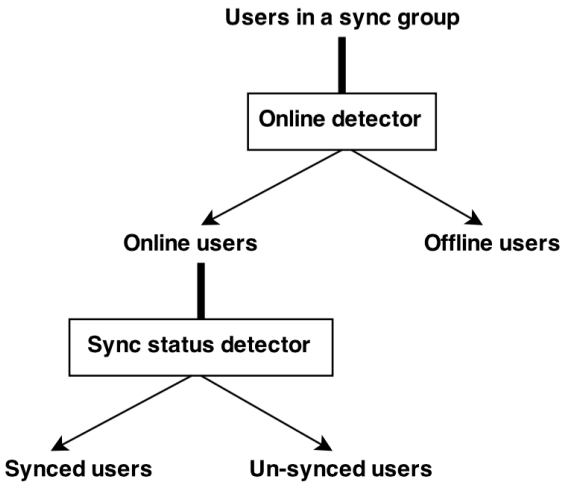


Abbildung 4.1: User classification in a sync group

4.2.2 Deal with missed sync content during user's offline period

User may log out for some time after s/he joins some sync group. So s/he may miss some synced content made by other users in sync group. How do we deal with the offline users? From the above subsection 4.2.1, we know there are a set of online users who have been synced by others. When a user logs in again, the local directory takes the role of client, one of the online user who has already been synced out of the sync group takes the the role of server, they are set to do 1-to-1 sync by Jarsync API. So as long as a user ID is in the group, s/he is authorized to sync the local folder with other users in the same group. Whenever s/he goes online, sync circle will automatically start, new hash tree is generated, files under his/her local directory are updated by comparing with another online user's files. In this way, the missed content will be written into user's sync directory even user logs out for a while.

4.2.3 Owner view syncing process

Abbildung 4.2 illustrates a flow chart of data sync process from owner view. 1-to-1 sync process starts after there are online users who accept invitation. And this process will persistently proceed in a loop until all the online users out of the sync group have been synced. An online detector procedure is used to find online status of the sync set. And this online status of sync group is updated whenever someone logs in and logs out.

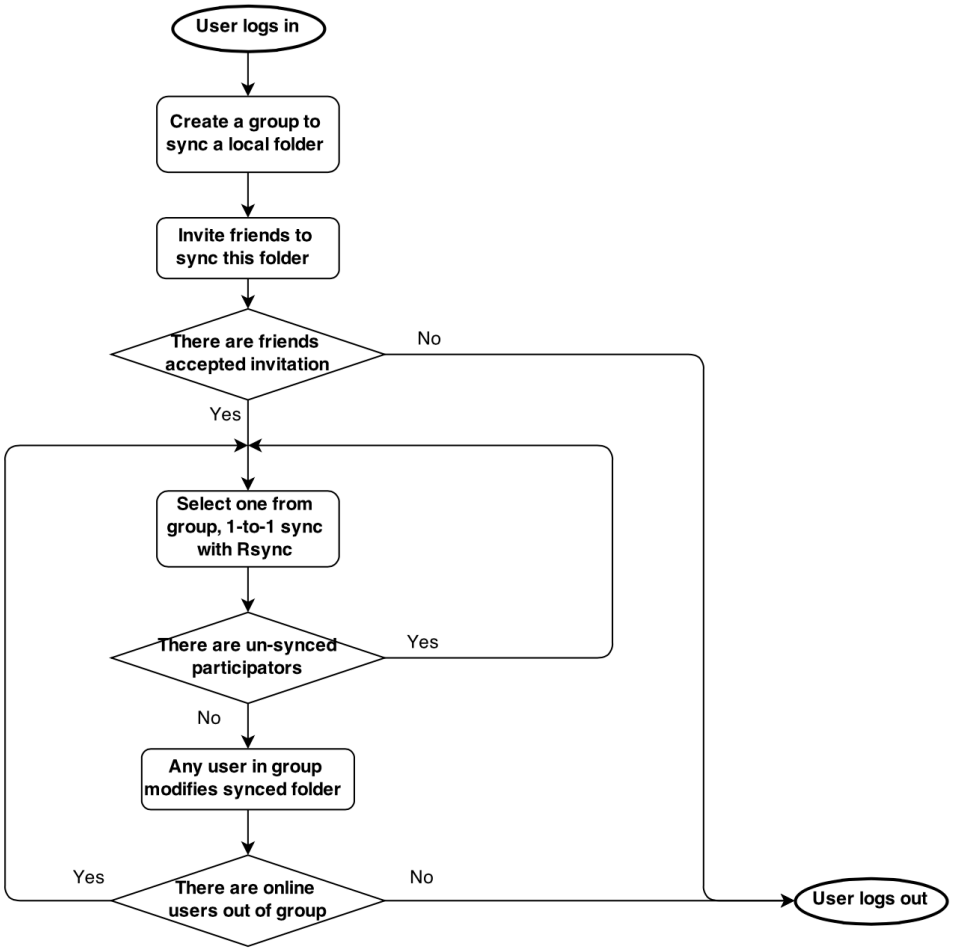


Abbildung 4.2: Flow chart diagram of data sync process from owner view

4.2.4 Participator view syncing process

Abbildung 4.3 illustrates a flow chart of data sync process from owner view. After copied the database from the owner, the participant can also modify (add/delete/edit file) the database under his/her own local directory. The modified data will be synced to other machines in a loop by 1-to-1 sync process.

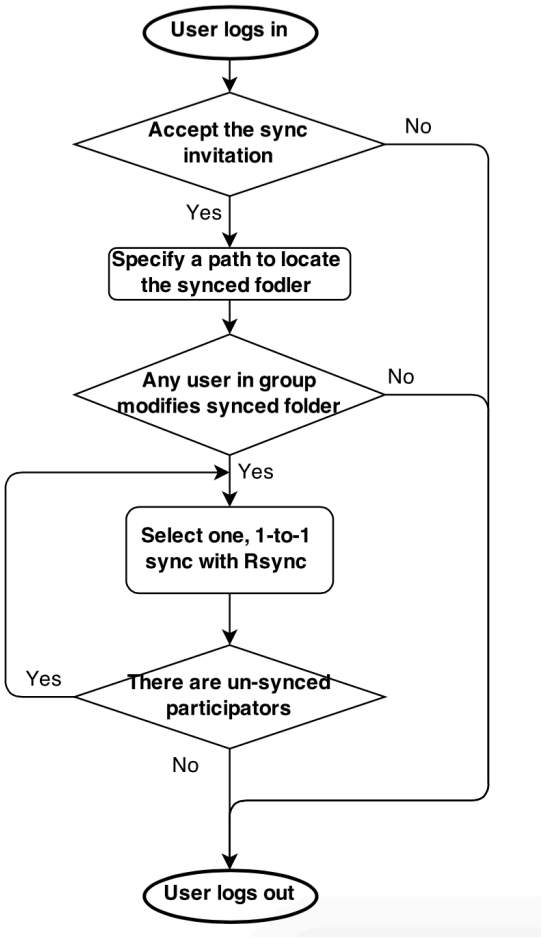


Abbildung 4.3: Flow chart diagram of data sync process from participator view

4.3 Strategy Implementation

This section describes the implementation process of the data synchronization application. First we describe how did we implement the Rsync algorithm using Jarsync API. In subsection 4.3.2, we implemented a solution: use *DistributedLinkedList* (*dll*) to store files' status in the database. But some problems generated from using this *dll* structure. Thus, in subsection 4.3.3, we abandon *dll* and apply the Merkle Tree algorithm into our application.

4.3.1 Implementation of Rsync

Class *JRSync* in plugin is the implementation of Rsync utilities. It calls Jarsync API by following methods:

- *getSums()*: computes checksums of a file.

- *computeDelte()*: generates a list of delta.
- *rebuildFile()*: rebuild a file with a list of delta.

4.3.2 Modification detection with DLL

Synchronizing two files with the implementation of Rsync algorithm is done. Now we should consider how should we detect files status under a directory. So we need some data structure in which we can store status of files and keep on updating the status. And by frequently comparing the status with the file status under local directory, we can find which find under the local directory has been modified/added/deleted. In subsection 2.2.2 of Chapter 2, we have discussed that the data structure *DistributedLinkedList* can be used to represent a collection class list in a distributed manner. So we implemented a file status detecting method with *dll*.

Modification detection mechanism with DLL

The mechanism of detecting files' modification status with *dll* is as below:

1. When the user specifies to sync a folder, read and store the files' status (*«filename»*, *«modification date»*) in a *dll*.
2. Periodically read files' status under the sync directory of each online user's machine.
3. If there is modification date of any local file differs with *dll*, it has been modified. Take a loop to sync (by calling *JRSync*) the corresponding files under other directories. Then update *dll*.
4. If there is any (*«filename»*, *«modification date»*) found under local directory, but it doesn't exist in *dll*, it is an added file. Simply copy the file to other directories. Then update *dll*.
5. If there is any (*«filename»*, *«modification date»*) can not be found under local directory, but it exists in *dll*, it is deleted from the local directory. Delete the corresponding files under other directories. Then update *dll*.
6. This read-compare-sync process loops over all the online user machines in each sync circle.

The working mechanism diagram is illustrated in Abbildung 4.4.

Problems generated from using *dll*

After the above mechanism was implemented, the applications could provide the following functionality for users to sync a folder:

- Edit existing files and sync them with others.

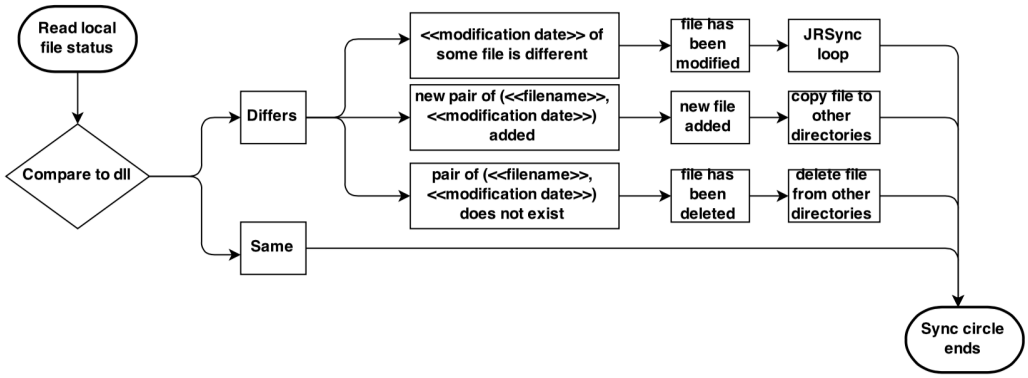


Abbildung 4.4: Working mechanism of a sync circle by detecting file status with *dll*

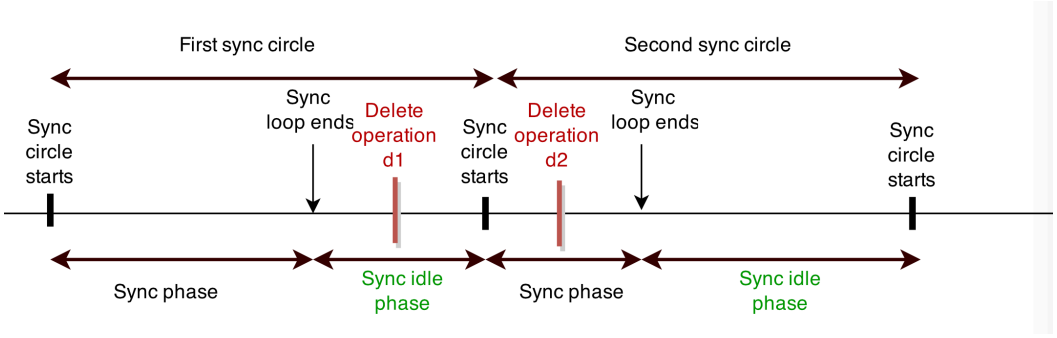


Abbildung 4.5: Reason of problematic deleting files with *dll*

- Add new files under local directory and sync them with others.

Also two problems generated with this mechanism.

1. Deleting files
   We find that the functionality of deleting files does not work properly. Sometime the deleted file is added back after user delete it. User has to do the deleting operation for several times then only the files can be deleted from local directory and the remote directories.
   The explanation of this problem generation needs the illustration of diagram shown in Abbildung 4.5.

   As shown in Abbildung 4.5, we set a sync circle for syncing a database. For example 10 seconds. Then we first detect for modified files status by using *dll*. Sync phase will start until the all the other machines have been checked and synced. We say sync loop ends at this point of time. The period from sync loop ends to the next sync circle starts is sync idle phase. If the delete

operation $d2$ occurs during sync phase as shown in the diagram, the deleted file will be added back as a new added file from one of other machines. The reason is, at the time point of deleting file, the detecting works is done and the *dll* has already been updated. The deleted file can not be detected in current sync circle. And during sync phase there is a machine which is holding the server role, it is syncing other machines in a loop. The server does 1-to-1 sync to the client on which a file is deleted, thus the deleted file is added back. Oppositely, the delete operation $d1$ which occurs during sync idle phase of a sync circle works fine. Because the deleted file can be detected and the *dll* can be updated right after the next sync circle starts, then the corresponding files will be deleted from other directories.

2. Sub folder under directory
   The files and folders structure is limited by the structure limitation of *dll*. It stores a list of file status. So only a list of files under each directory, their status can be read. No sub-folders can be detected or synced to other directories.

4.3.3 Combination of Rsync and Merkle Tree

We can not control the user delete operation occur at which point of time. So the deleting file problem can not be solved if we use *dll*, at least till now we have not got a solution. For the sub-folder problem, there may exist sub-folders and files under them, even deeper sub-folders and files under directory, of course the structure of them can not be managed with a *dll*. A tree structure is need. Thus this remind us about the Merkle Tree algorithm which is introduced in section 3.2 of Chapter 3. We can modify the Merkle Tree structure a little bit and make use of it to build a hash tree structure for the structure of files and folders under the sync directory.

Simplified Merkle Tree Algorithm

It has been discussed in the subsection 4.1.1 of this chapter, the size of diff content generated by Rsync is much smaller than the size of diff blocks which are detected to be different by Merkle Tree. But Merkle Tree implemented a very good value computation and modification detection mechanism for tree structure database. So Merkle Tree can be simplified and used to implement a database retrieval mechanism. Files are not needed to be spited into blocks. There leaf nodes of the tree structure are files themselves. The mechanism is described as follow:

- Hierarchical value computation:

  1. Assign a value to each leaf node which represents a file.
  2. Compute value of node from the bottom up. Computation method see subsection 3.2.2 in Chapter 3.

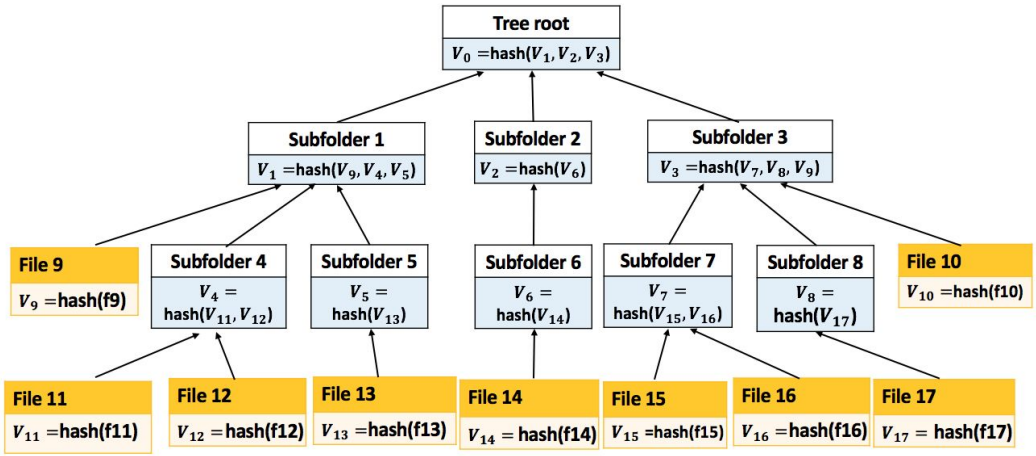


Abbildung 4.6: Simplified Merkle Tree structure used in our sync application

- Retrieve for modified files.
  To detect modified/added/deleted files, retrieve the Merkle Tree and compare it with another Merkle Tree. Retrieval method see subsection 3.2.3 in Chapter 3.

- Sync over the online user machines.
  Any detected file which has been modified, it takes the server role, calls *JR-Sync*, takes a loop to do 1-to-1 sync with the corresponding files on other directories.

Compared to the Merkle Tree discussed in section 3.2 of Chapter 3, each leaf node in our diagram of Merkle Tree structure donates a file, other parent nodes donate a folder. This structure is shown in Abbildung 4.6.
This was the mechanism of using Merkle Tree to detect the modified/added/deleted files by retrieving Merkle Tree.

Implementation of Merkle Hash Tree

Based on the principles of Merkle Tree algorithm introduced in section 3.2 of Chapter 3, the class *Hash.java* in package *org.lifesocial.plugin.sync.merkle* implemented the

way of building and retrieving a Merkle Hash Tree for a sync directory with the following methods:

- *hashFile()*: to hash a file (leaf node).
- *hashNode()*: hash of a folder (parent node) computed by all files under it.
- *isSame()*: check if value of a node has been changed.
- *update()*: update a hash tree.

File utilities

The class *FileUtil.jave* implemented the following utilities of files:

- *copy()*: copy files in the form of *(InputStream, OutputStream)*.
- *getBufferedReader()*: buffers InputStream.
- *getFileInputStream()*: get InputStream of a file.
- *pass()*: transfer data.
- *save()*: save file operation.

Adjustment of strategy with Merkle Tree

After implementing the data structure Merkle Hash Tree and utility functions of file, now we need to consider, how can we implement the mechanism mentioned above. Some parts of the strategy need to be adjusted, because we need to solve some more problem:

- Storage and comparison of Merkle Hash Tree:
  The hash trees generated from *Hash.java* are stored in a hidden folder *.sync* under each local sync directory. Because there is no centralized control of comparing different hash trees of multiple databases, the modification of a database has to be found by self-comparing. So for each local directory, there are two hash trees: a old hash tree and a new hash tree. The new hash tree is Merkle Tree generated for the current database, and old hash tree is Merkle Tree generated last time. Whenever sync process gets over by generating a new hash tree, the old hash tree is replace by the new one, and gets ready for the next comparison. The method *saveHash()* in class *SyncLogic.java* implements the storage of the hash tree.

- Trigger the sync process: user saving action
  Now there is an existing hash tree, we need a method to trigger the generation of a new hash tree, then compare it with the old one to find modified files, then loop over sync group with 1-to-1 Rsync, this is the sync process. It occurs

by generating a sync session with methods *startSyncSession()*, *doSync()* and *stopSyncSession()* in class *SyncLogic.java*. But how to trigger a sync process? Here we use the user action of saving a file to trigger it. Thus, whenever a user edits a file and saves it, a new hash tree is generated to be compared with the old one. The edited file can be found, then it starts to sync other corresponding files by calling *JRSync* loop over online users in a group. The method *save()* in class *FileUtil.java* implements this process.

- Detect added/deleting files: fixed sync circle
  By user action saving file we can find the modified files, what about added/deleted files? If user adds/deletes a file under local directory, there is no action which can trigger the generating of a new hash tree and comparison. Thus we still need to set sync circle to periodically generate new hash tree, do comparison and update hash tree stored locally.

Fixed and unfixed sync circles

First it is needed to be explained, why is the user's action saving file needed to trigger the sync process. Anyway there is new hash tree generated and being compared in the beginning of each fixed sync circle (here in our application we set the circle 10 seconds). The modified file (if there is any) will be found without any doubt. Why do we still need user action to trigger sync process? This is aimed to avoid the occurrence of modification conflicts. See Abbildung 4.7.
In *A* of Abbildung 4.7, imagine two users modify the same file inside of one sync circle, if there is no user's saving action could trigger syncing process, both of these two modifications would be detected in the beginning of next sync circle. Which one should be the sync source among them? The conflict occurs. But as shown in *B* of Abbildung 4.7, if user saving action triggers sync process: generate a new hash tree, compare with the old one to find the modification file, then loop over sync group with 1-to-1 Rsync. This sync process occurs very fast due to the high efficiency of the algorithms Merkle Tree and Rsync. At the same time when this sync process ends, new sync circle starts, sync circle is no longer fixed as long as there is file being edited. When there is nobody edit files, fixed sync circle starts again to detect the added/deleted files. Thus, in order to avoid data modification conflicts as much as possible, user can trigger sync process by saving file.

Sync process messages management with Message Queue

Here we manage the sync process with a message queue. When a data modification is detected, the source-targets communicate with each other by message, these messages are organized in a FIFO mode. When a sync circle ends, message is out of the queue, the next first one is ready to proceed. In this way, we record and organize the frequently multiple modifying operation from different machines. This queue structure is implemented in class *MessageQueue.java*.

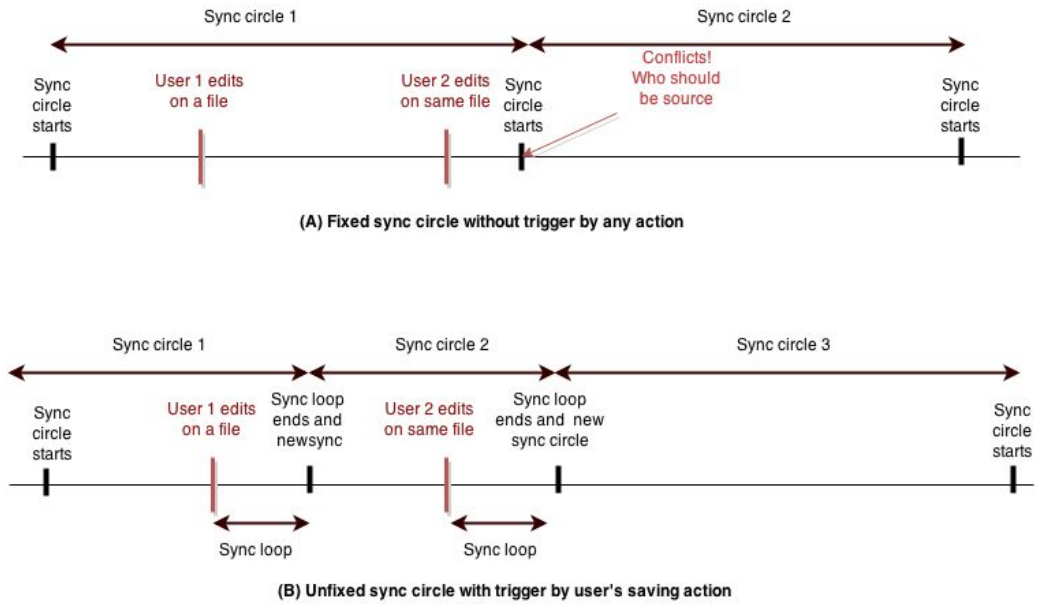


Abbildung 4.7: Fixed and unfixed sync circles

Modification detection mechanism with Merkle Tree

With the above solutions of these problems, the modification detection mechanism is modified as shown in Abbildung 4.8.

Sequence Diagram

Abbildung 4.9 illustrates the core sequence diagram of detecting of file modification and syncing process with Rsync between a source and a target.

Sync utilities implementation

Some kernel utilities of sync process are implemented in class *SyncUtils.jave*:

- *getRelPath()*: get file path.
- *findByName()*: get file by filename.
- *getOtherPeers()*: list of other online peers out of sync group.
- *getDllIDForFileTrack()*: track a synID of specified userID.

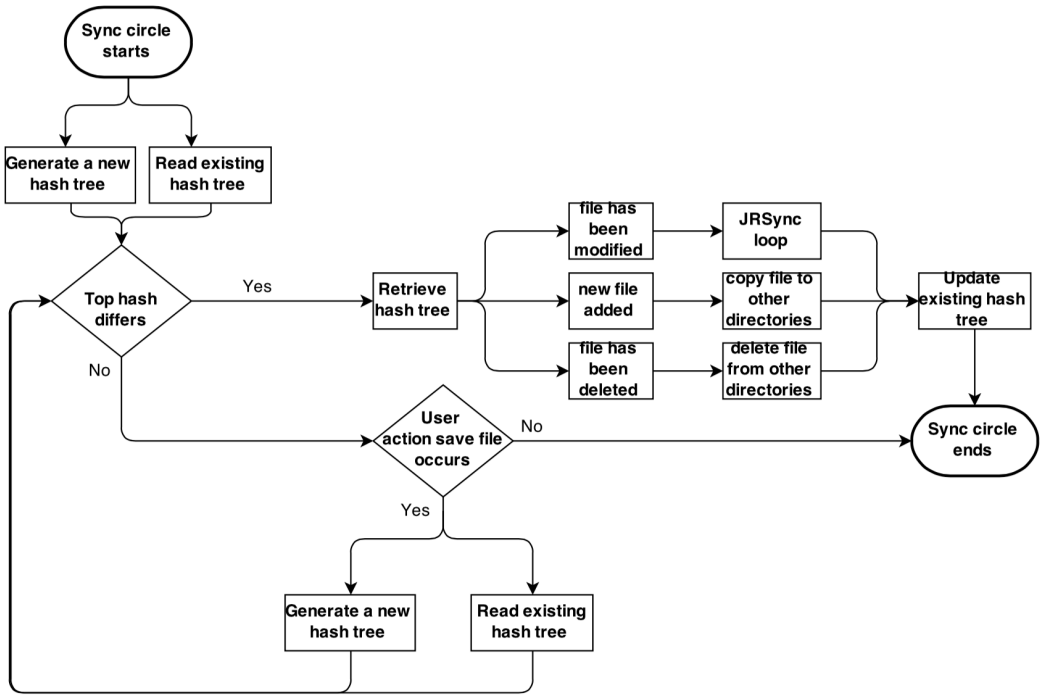


Abbildung 4.8: Working mechanism of a sync circle by detecting file modification with Merkle Hash Tree

- *getDllIDForSyncRef()*: get DLL id of some sync reference.
- *getLocalFile()*: read access to local files.
- *getMySync()*: collection of sync groups of a userID.
- *fineDeletedFiles()*: return a set of deleted files.
- *saveWholeFile()*: copy added files from remote directory.

Achievement with Merkle Tree

Except syncing edited and added files, now the application is also able to delete files from remove directories. Compared to detecting file status by *dll*, modification detection by Merkle Tree algorithm provides a more efficient way to find the edited/added/deleted files. At the same time, it allows users to have sub-folders under their sync directories. No matter how deep the file path is, it can be found with retrieving the Merkle Hash Tree. With Merkle Tree algorithm, we solved the problem in the above subsection 4.3.2.

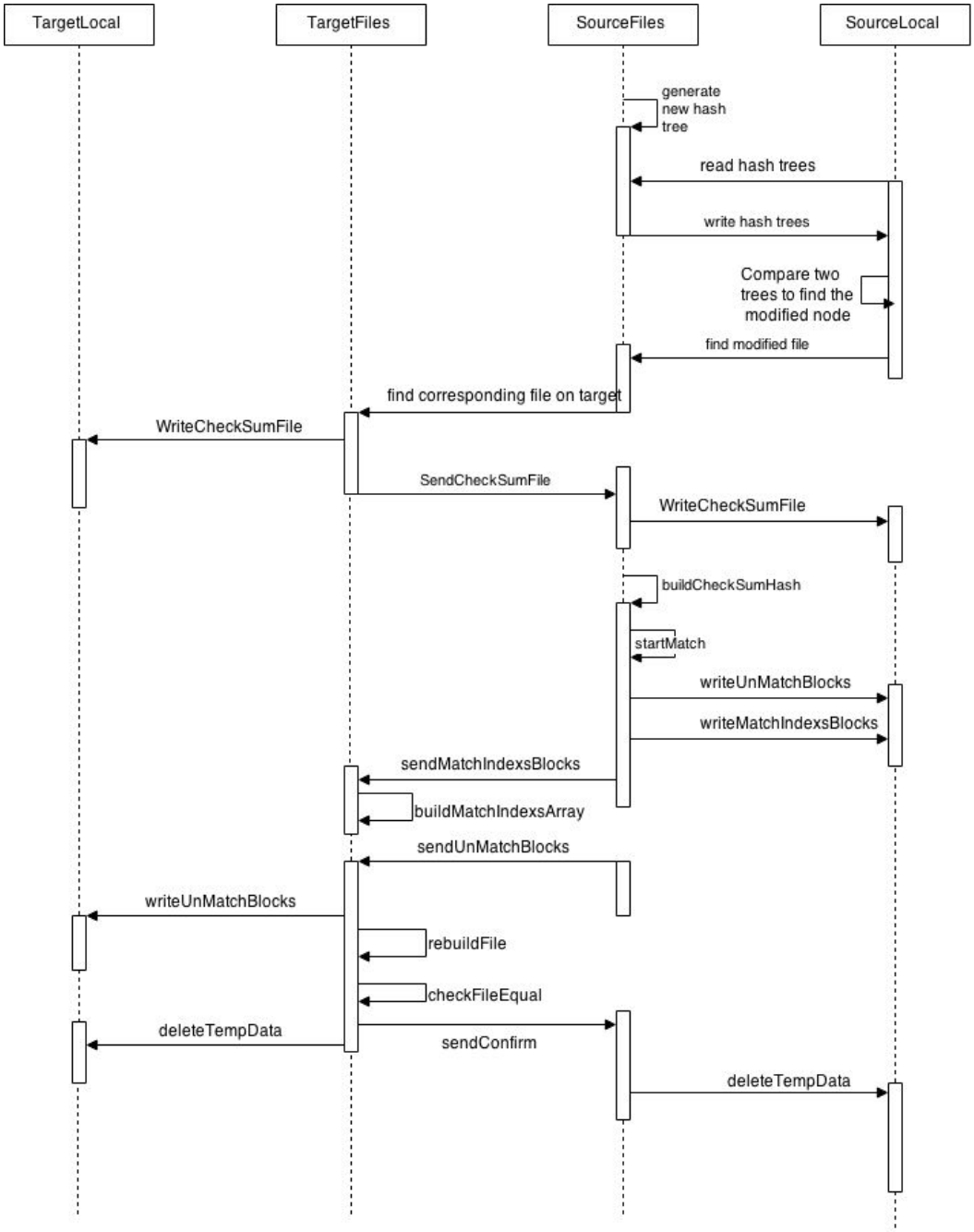


Abbildung 4.9: Sequence Diagram of Combination of Merkle Hash Tree and Rsync

Up to here, we have made full use of two efficient algorithms for data synchronization: Merkle Tree and Rsync. Merkle Tree algorithm works on local machine to detect data modification of local sync directory. Rsync algorithm is used to compute diff content of two files and to rebuild target files. We believe that both of these algorithms have improved the efficiency of data synchronization from different aspects.

4.4 User Authorization to Access a Sync Object

When a owner create a sync group, this group is encrypted and stored as *SecureStorageItem* in the framework. This is implemented from the view of security. The owner can invite users from his/her friend list to join a group. The read/write access to user's data on local disk has to be authorized. Thus, the invitation from owner of a sync group is sent by using Notification plugin, user is added to sync group only when s/he accepts the invitation. Considering other users who are not on the friend list of the owner, s/he may also have interest to join the group, request to join a group with knowing the group ID is also available. User is added into group right after the owner accepts the request.

4.4.1 Join sync group by invitation

Abbildung 4.10 illustrates the work flow of user authorization in a sync group. In this case, user are added to sync group by accepting the invitation from the owner of group. The owner can invite any user from his/her friend list and send them invitation by notification message.

4.4.2 Request to join a sync group with *group ID*

Abbildung 4.11 illustrates the work flow of user requests to join a sync group. Considering some people may not be on the friend list of the owner of sync group, but they may also have interest to sync the object. Requesting to join some specified sync group with group ID and owner name is also available. Why user name is necessarily? From the perspective of security, a sync group is stored as *SecureStorageItem* in the framework, it is not allow for any one to join the group without permission. So a notification message has to be send to the owner, then user will be added into the group when owner accepts the request.

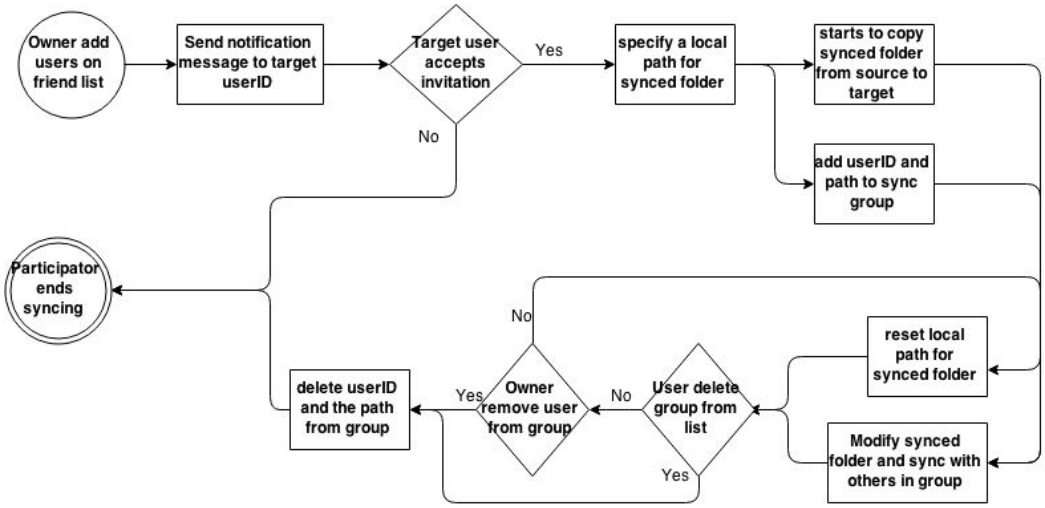


Abbildung 4.10: Work flow of user authorization of in a sync group

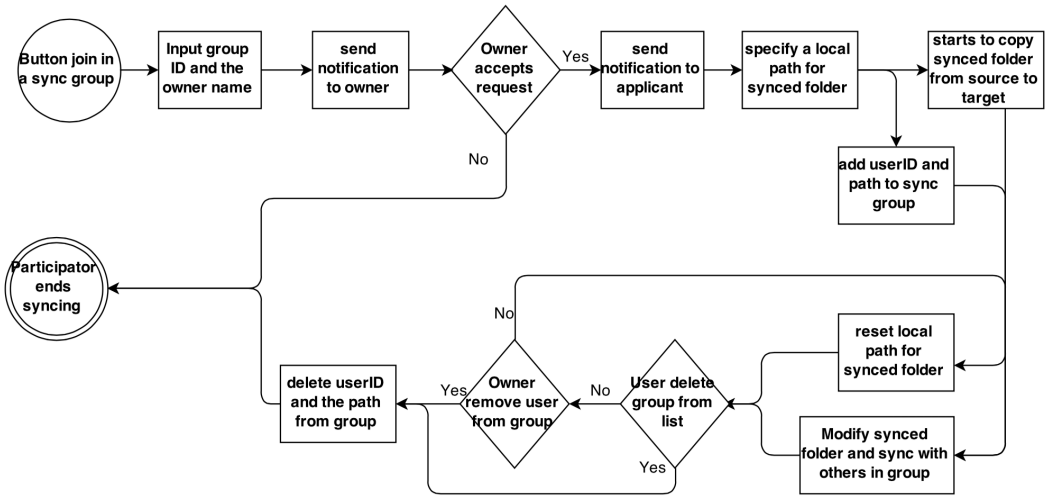


Abbildung 4.11: Work flow of user requests to join a sync group

4.5 Data Structure Implementation

We implemented a hash tree structure in class *Hash.java* which has already been discussed. Additionally, we implemented some data structures in package *org.lifesocial.plugin.sync.data* and data structures in package *org.lifesocial.core.data.sync* which need to be stored in framework.

4.5.1 Core data stored in the framework

There are three kinds of data structures stored in the framework: file information, sync group and sync reference.

- *FileInfo.java*: defined a web view information (filename, size, status and so no) of a file, which used to display on HTML GUI.
- *SyncGroup.java*: stored as *SecureStorageItem*, implements *Serializable*, used to authorize users of a sync group, stores all information of this sync group.
- *SyncReference.java*: implements *Serializable*, used to set/get user's local path for a sync object.

4.5.2 Data structure used in syncing process

Data structure used in syncing process:

- *MsgFileSync.java*: sync plugin message, extends *AbstractBasicMessage*.
- *Node.java*: defines of a node (represents for a file or folder in the local database) in a merkle hash tree.
- *PayloadFileDelta.java*: delta generated from Rsync.
- *PayloadFilePartData.java*: file transmission ratio: x out of y bytes synced.
- *PayloadFileSum.java*: file checksums generated by Rsync.
- *PayloadFilePushRequest.java*: request for sending file, target answers if it has the file: *PayloadFileSum* request for diff contents. If target does not have the file: *PayloadFileWholeRequest*: request for the whole file.
- *PayloadFileWholeData.java*: send whole file.
- *PayloadGetSyncFileList.java*: request files over view: local hash tree, the target compares with its hash tree, return *PayloadGetSyncFileListReply*.
- *PayloadGetSyncFileListReply.java*: files over view, includes the files need to be deleted from remote side.
- *PayloadInitSync.java*: request to start a sync session.

- *PayloadInitSyncAnswer.java*: answer to the request. OK to start a session if idle, no if it is being a source or target.
- *PayloadStopSync.java*: stops a sync session.
- *StoredFile.java*: whole file temporarily stored in the network when whole file needs to be transferred.
- *SyncFileList.java*: files over view used in *PayloadGetSyncFileListReply*.
- *SyncFileRecord.java*: single item in *SyncFileList*.

4.6 Data Transmission via Framework

4.6.1 Sync content transmission

According to the sync process discussed above, data need to be transferred via framework are: whole file by the operation *PayloadFileWholeData* and delta by the operation *PayloadFileDelta*. We called them sync content. The sync content transmission is implemented in class *FileUtil.java*. The mechanism of transferring sync content via framework is described as below:

1. After sync content is generated, we write it into a *OutputStream* with method *save()*.
2. Method *pass()*: write len bytes from the specified byte array starting at offset off to this *OutputStream*.
3. Copy the *OutputStream* from source into *InputStream* for a target with method *copy()*.
4. *getFileInputStream()*: get file input stream.
5. *getBufferedReader()*: create a buffering character-input stream that uses a default-sized input buffer.
6. *getRawBufferedReader()*: create a buffering stream if the character encoding is not supported.
7. *readString()*: create a *InputStreamReader* for reading the input stream.

4.6.2 Big file transmission

Some time we need to transfer whole file through the framework. We found that the transmission of big files (bigger than 100 MB) through the framework is very problematic. Transferring 1 MB of data takes about 3,4 seconds. The system reports the following error message: *Memory is almostly exhausted! Object size calculation is stopped! The object size is set to 0!*. So we actually could not transfer big file



Abbildung 4.12: Work flow of transferring a big file via framework

through the framework. Because of the characteristic of the sync plugin, we get the default reaction timeout message when it takes too long time but no response from receiving data. Abbildung 4.12 illustrates the work flow of transferring a big file from source side.
Abbildung 4.13 illustrates the work flow of building file on target side.
As shown in Abbildung 4.12 and 4.13, we deal with big file transmission through the following steps:

1. We use 500 KB as unit (because this takes less then 1 second to be transferred in LAN), split the file to pieces.
2. Give serial numbers to these pieces.
3. Transfer these pieces one by one in sequence from source to target, target

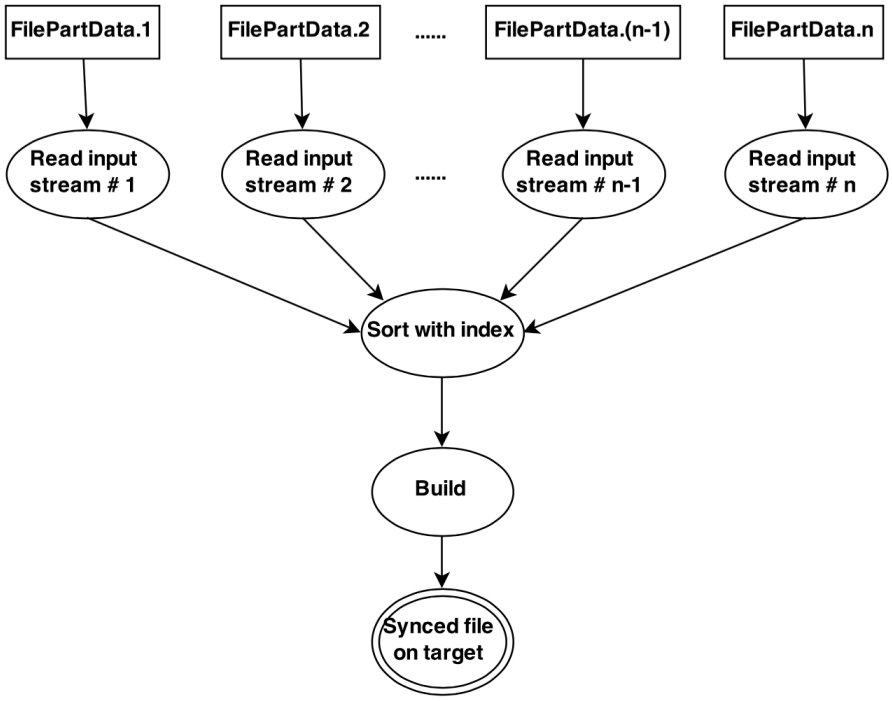


Abbildung 4.13: Build file with parts of splited file

sends a ACK message (Acknowledge of receiving some piece) to source after it receives the piece.

4. Target sends the next piece only after it receives the ACK message from target.

5. Repeat steps *3* and *4* until all pieces are transferred.

6. Build file with these pieces and their serial numbers on target side.

This process is implemented in class *SyncLogic.java* with methods *sendWholeFile()* and *sendWholeFileRequest()*.
By the way, the data transmission message from source to target uses method *messageDispatcher.sendMessageWhithoutStoring()*. Because the transmission message provided in the framework is one-way. So here we import the method of *sessionId* to build two-way conversation which is similar with TCP. This is implemented with method *handleMessage()* in the class *SyncPlugin.java* and the field of *sessionId* record in class *MsgFileSync.java*.

4.7 Web View Implementation of File Sync Status

Many synchronization application like Dropbox implemented a file management program with file explorer by exchanging the icons. Through this users can know about file status in a synchronization process. But this kind of file status displaying program has to be developed for different operating systems (even different versions of the same operating system). In our case, a web view sync status design is more future-proof on the long term. As we had a overview about the technologies used in building HTML GUI for sync plugin in section 2.2.3 in chapter 2, now we introduce about implementing a sync status web view in HTML GUI.

4.7.1 Sync group list view

Abbildung 4.14 is a screenshot which illustrates a web view of user's sync group list. In sync group list view, method *getMySyncList()* in *SyncPlugin.java* is implemented to collect the sync references of a user ID. For each sync group on the list, following information are displayed: index, name of the sync group, sync group ID, current status of the local sync directory, name of owner, share option (only available for owner), delete option and the local path of synced folder.
Here the sync groups created by the user or in which user joined as a participator, these two kinds of group information are collected in the list. For a owner, sharing option is available for inviting more friends to join sync, or remote some users from the sync group.
Deleting option is used to delete the sync group from user's sync group list. For a participator, deleting means s/he leaves from the group and gives up to sync with others. For the owner, deleting means s/he delete the sync group from framework, the synced folder will not be synced any more.
Local path can be reset by user. The status of all files under the local directory can be shown by clicking on the local path.

4.7.2 File sync status view

By the user operation clicking on a local path of a sync directory, the page of files under directory is entered. On this page, user can check the status and info of each single file or subfolder. Abbildung 4.15 is a screenshot which illustrates a web view of sync status of files under the sync directory.
As shown in Abbildung 4.15, on the top field of the page, sync group info, local path and users in the group are shown. The number of files have been synced and the total number of files need to be synced are shown in the form: x out of y files synced. Then a paginator is set to display a very large number of files. 20 files are displayed on each page. User can go to other pages to check files' status. On the file info list, type (file or folder), status of file, name of file, size of file and modification date are shown for each file under the directory. By clicking on the folder icon, files

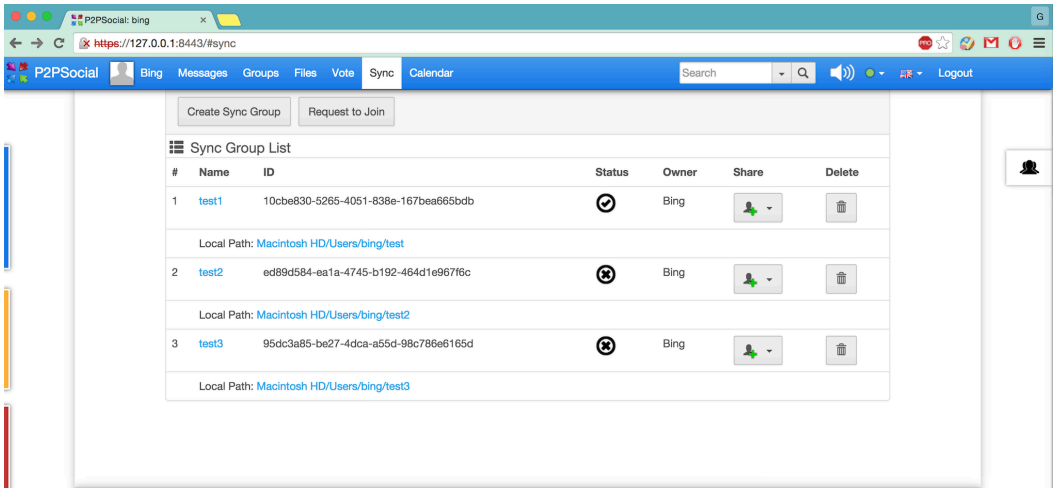


Abbildung 4.14: User's sync group list view

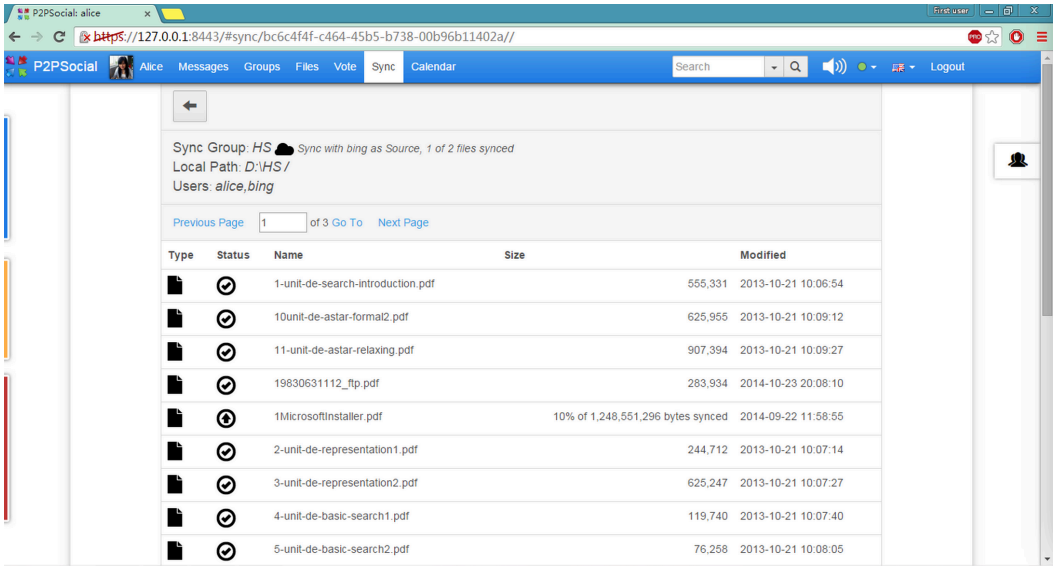


Abbildung 4.15: Status web view of files under local directory

under subfoldlers can also be seen.
Under the size info column, in addition to the total size of file, the proportion of file which has been synced is also shown in the transmission process of big files.
User operation click on button go back, go back to sync group list view.
Current status of each local sync directory is exchanged between 4 icons shown in Abbildung 4.16.
These icons respectively donate: syncing with other users, synchronous completion status, checking for modification, disconnect with other users.



Abbildung 4.16: Status icons for local sync directory



Abbildung 4.17: Status icons of files/folders under a sync directory

The icons donate syncing status of each file under the sync directory are shown in Abbildung 4.17.
Here these icons respectively donate: file status up-to-date, uploading sync content to others and downloading sync content from others.
The sync proportion info which needs to be shown on HTML page is implemented by calling method *FS.syncCurrentFilePercent()* of class *SyncLogic.java* in sync plugin. And the communication method provided by Knockback and Bootstrap between HTML GUI and sync plugin is implemented by *sync.js* in the plugin web provider.

Summary of the Chapter
In this chapter, we introduced the process of implementing the data synchronization application according to the following order: selection on algorithm and tool, strategy design, strategy implementation, problem with using *dll*, solution with combining Merkle Tree and Rsync algorithm, some adjustment with the strategy implementation, user authorization implementation, data structure and transmission mechanism during sync process, a file sync status web view design on HTML GUI. This is basically how did we implement the sync application according to the goals we discussed in section the section of introduction. As the desired functionality of the sync application has been implemented by the above sequence, we will evaluate the functionality for the next step in chapter 5.

5 Evaluation

In this chapter, we evaluate this synchronization application from the following aspects: consistency between the outcomes and the goals discussed in section 1.3 of Chapter 1, test cases of the functionality and performance analysis.

5.1 Outcomes

In this section we give a over view about the outcomes of the synchronization application by illustrating users' effective activities. Then check if the outcomes can meet the requirements of goals.

5.1.1 User activities

The effective user activities of using the synchronization application are:

- Create a new sync group, specify a local folder to be synced, add description about the sync group.
- User accepts invitation, set local path to locate the synced folder, then starts downloading files from a source machine.
- Any user insider the sync group can add/delete and modify files under local sync directory, the modification can be synced to other online machines.
- For the owner of a group: invite users from his/her friend list to join the sync group by send them invitations, remove users from group.
- Reset local path is available for any user in the group.
- Visualized status of the synced folder and the files/folders under it are shown to users.
- For a participant, s/he can delete sync group from his/her sync group list, after that s/he won't get any update from other updates.
- If a owner deletes a sync group, the synced folder stops to be synced.

5.1.2 Consistency between outcomes and goals

Now let us look at the use case diagram Abbildung 1.1 in section 1.3 of Chapter 1. Compared to the above list of user effective activities, we can see that all these requirements are met. In addition, another functionality like visualizing file syncing status is also implemented.

5.2 Evaluation Environment

The test cases and data sets used in the later sections are based on the testing results on at most 5 machines on which 5 users are registered and join one/more sync groups. The evaluation environment can be seen in Abbildung 5.1. Among these machines, operating system of 3 machines is windows 7, 1 machine is with Mac OS, 1 machine is with Ubuntu.

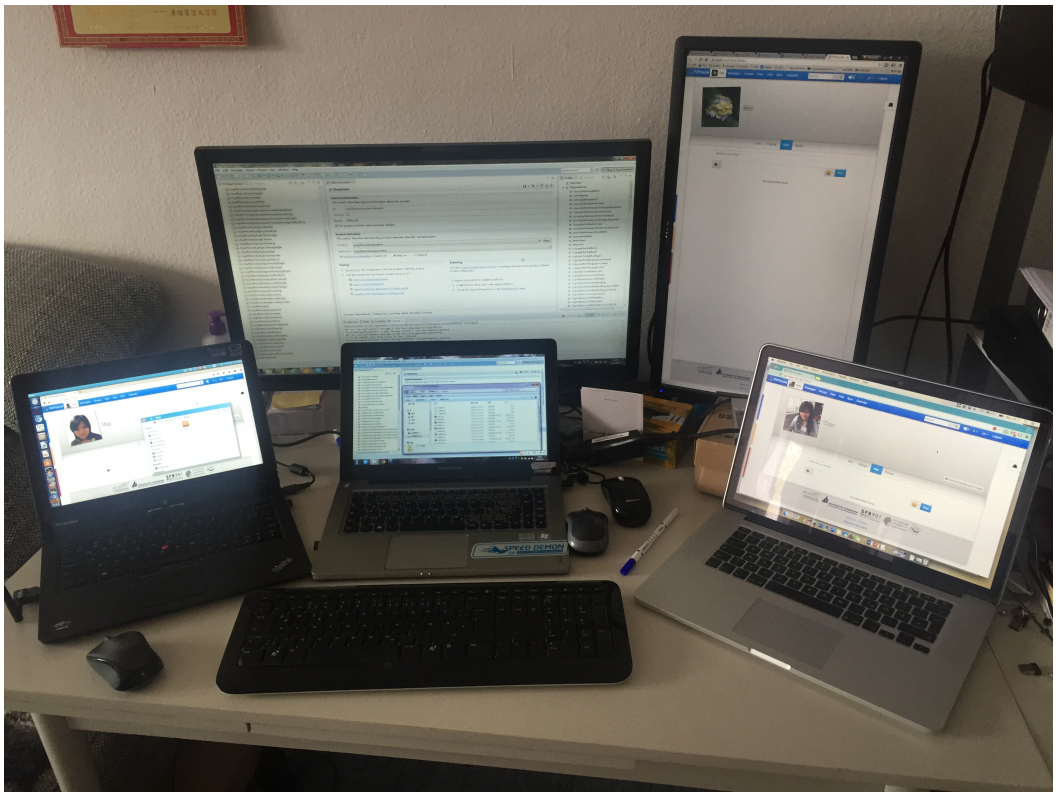


Abbildung 5.1: Evaluation Environment of the Synchronization Application

5.3 Illustration of Test Cases

Tabelle 5.1 shown below is the test cases which are tested based on functionality provided by the data synchronization application.

Test Case :	**Data Synchronization in 2 peers**
Description :	Register users A and B, A initiates to sync a local folder and invites B B accepts invitation
Operations :	+ add files from A + add files from B + delete files from A + delete files from B + edit files from A + edit files from B
Exception :	Synchronization of adding, deleting and editing files between 2 peers work fine.
Result :	Gut

Test Case :	**Data Synchronization in 3 peers**
Description :	Register users A, B and C A initiates to sync a local folder and invites B and C B and C accept invitation
Operations :	+ add files from A + add files from B and C + delete files from A + delete files from B and C + edit files from A + edit files from B and C
Exception :	Synchronization of adding, deleting and editing files among 3 peers work fine.
Result :	Gut

Test Case :	**2 (multiple) Sync Groups in One Peer**
Description :	Register users A, B, C, D and E A initiates to sync 2 local folders respectively with two groups A syncs folder 1 with B and C A syncs folder 2 with D and E
Operations :	+ add files to folder 1 from A + add files from B and C + delete files under folder 1 from A + delete files from B and C + edit files under folder 1 from A + edit files from B and C + add files to folder 2 from A + add files from D and E

	+ delete files under folder 2 from A
	+ delete files from D and E
	+ edit files under folder 2 from A
	+ edit files from D and E
Exception :	2 sync groups do not interfere with each other.
Result :	Gut

Test Case :	**Big Files Transformation**
Description :	Add a file bigger than 100 MB
Operations :	+ add files from A
	+ add files from B and C
	+ delete files from A
	+ delete files from B and C
	+ edit files from A
	+ edit files from B and C
Exception :	Synchronization works fine on these 3 different Operating Systems.
Result :	Gut

Test Case :	**Data Synchronization among Different Operating Systems**
Description :	Register user A on Windows 7
	Register user B on Linux (Ubuntu Kylin)
	Register user C on Mac OS X Yosemite
	A invites B and C to sync a folder
	B and C accepts invitation
Operations :	+ add files from A
	+ add files from B and C
Exception :	Big files can be transferred via framework.
Result :	Gut

Test Case :	**Zero-sized Folder Synchronization**
Description :	Specify a initial empty folder to sync with other users
	Register A specifies an empty folder and invites B to join syncing
	B accepts invitation
Operations :	+ add files to empty folder from A
	+ add files from B and C
	+ delete files from A
	+ delete files from B and C
	+ edit files from A
	+ edit files from B and C
Exception :	Hash can be built and updated for an empty folder during synchronization process.
Result :	Gut

Test Case :	**A list of Big Files Synchronization**
Description :	Register users A, B and C

	User A syncs a list of big files, size of each file is bigger than 1 GB
	B and C join to sync the folder
Operations :	+ add more big files from A
	+ add more big files from B and C
	+ delete files from A
	+ delete files from B and C
	+ edit files from A
	+ edit files from B and C
Exception :	Big files can be transferred via framework, modification on big files can be synced.
Result :	Gut
Test Case :	**A Very Long File List with 200 Files**
Description :	Register users A, B and C
	User A syncs a list of 200 files
	B and C join to sync the folder
Operations :	+ add more files from A
	+ add more files from B and C
	+ delete files from A
	+ delete files from B and C
	+ edit files from A
	+ edit files from B and C
Exception :	Large number of files can be synced.
Result :	Gut

Tabelle 5.1: Illustration of Tast Cases

5.4 Data Set

In this section we analyze the performance of the synchronization application through the analysis of data sets.

This sync application is a totally new application provides data synchronization functionality to multiple users (according to the size of sync group). Obviously, with the expansion of the group's size, it is hard to say every synchronous content can be accurately transmitted via the framework to remote machines. To meet the goal of achieving accurate and fast data synchronization during social interactions, it is necessary to subsample same performance parameters to a set of interactions that are likely occurring among real users.

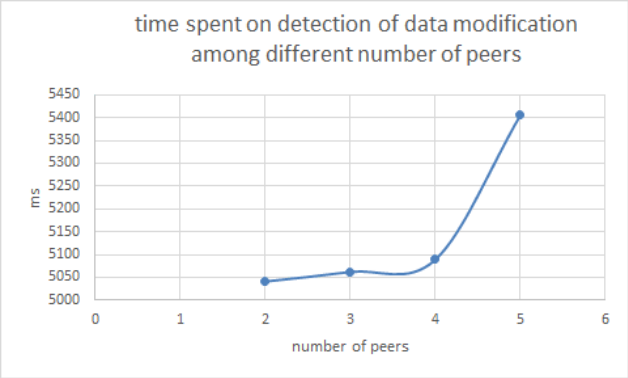


Abbildung 5.2: Time spent on detection data modification among different number of peers in a sync circle

5.4.1 Modifying file

When there is no data modification occurring among peers, still the sync circle is running to generate hash trees on each peer. The time spent on each sync circle is shown in Abbildung 5.2. Each value is calculated by the average value of five sets of data. In this case, time is spent on searching for online users of each sync group for a user ID, detecting for sync request from other users and build/compare/update local hash tree.

When their is data modification detected on any one in the group, usually it is assumed that the delta calculated Rsync is less than 500 KB. In this situation, some more time is spent on retrieving hash tree, getting file's path, computing delta, sending delta and rebuilding file. Still we calculate by the average value of five sets of data among different numbers of peers as shown in Abbildung 5.3. Unit of time used here is millisecond (ms).

5.4.2 Adding file

Time spent on syncing newly added file depends on the size of file. Now we set up 3 peers, sample some files with different sizes to check the speed of transferring files among there 3 peers. The transferring speed is calculated as shown in Abbildung 5.4. Here the unit of transferring speed is Mbps, unit of file size is kilo-byte (KB). From Abbildung 5.4 we can see, file size which is smaller then 1000 KB (1 MB) can be transferred with high speed. But files bigger than 1 MB are all transferred with almolst same speed: $1.2 Mbps$. That is because, we split big files with the unit of 500 KB. So files smaller than 500 KB can be transferred fast. Files with big size are split, transferred one be one and built on target side. Basically during each sync circle, 3 pieces can be transferred to a target. So the speed is quite constant.

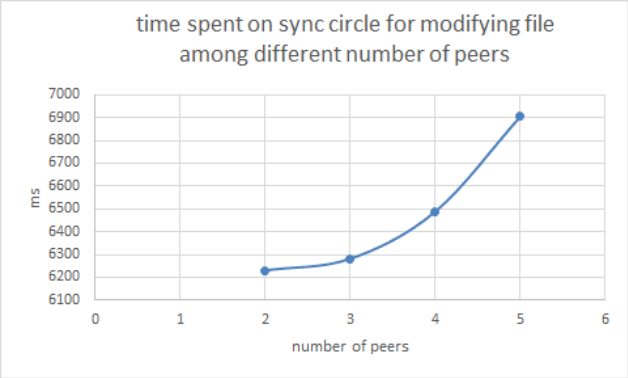


Abbildung 5.3: Time spent for syncing modified files among different number of peers in a sync circle

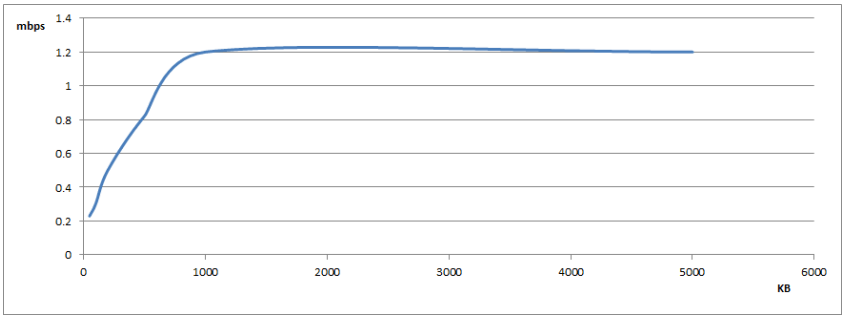


Abbildung 5.4: Transferring speed of newly added files among 3 peers

5.4.3 Deleting file

Subsamples of time spent on detecting a deleted file and delete the corresponding files on targets are shown on Abbildung 5.5. This time is basically spent on retrieving hash tree to find deleted file, getting deleted file path and deleting remote files from other online target machines.

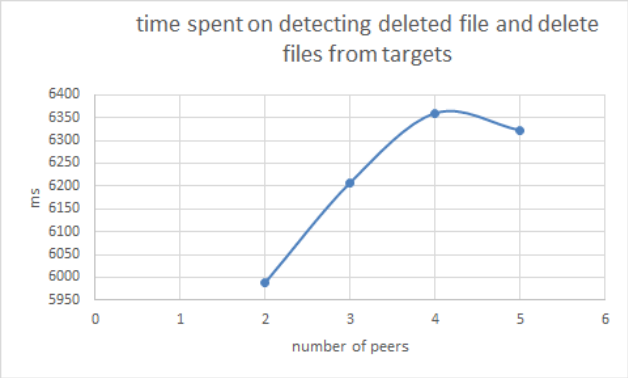


Abbildung 5.5: Time spent on detecting deleted files and deleting remote files in each sync circle

Summary of the Chapter

In this chapter we evaluated the synchronization application from the following aspects: comparison between the goals defined in the first chapter and the outcome of implementation, brief introduction of the evaluation environment, illustration of test cases and subsamples of some data sets during testing process. This is basically how did we evaluate the sync application according to the existing condition. In a word, the functionality of the synchronization application is not only implemented but also working fine as it was expected.

6 Conclusion

In this chapter, we conclude the thesis from two aspects: summary of the work in this thesis and some more work need to be done in the future.

6.1 Summary of Work

In this thesis, we have introduced the development of a data synchronization application built based on a p2p social network. The algorithms used in implementing the application and the implementation process are described in detail. And a simple way to evaluate the application is also provided. This application can provide a more interesting and convenient functionality for data sharing among multiple users. It have been realized that user can sync a local folder with friends and get data access from remote machines. The combination of the two efficient algorithms Merkle Tree and Rsync makes the syncing process more accurate and fast. On account of the feature of non-centralized database backup, this can be an useful and secure application in social networks.

6.2 Future Work

As future work, some more problems should to be considered. For example, in this application we assumed that the delta generated by Rsync was smaller than 500 KB. Cause Rsync works very efficiently with calculation and transferring of delta which is small enough. If user save file frequently during the process of modifying file. Each time the generated delta is small enough to be synced fast to others. But what if user modified a large size of data which is bigger than 500 KB? In this situation, using Rsync to calculate and transfer delta may not be a good solution. Cause the computation of hash sum and delta consumes too much CPU time, the sync process will be slowed down. In this case, simply copy the source file and replace the target file is even faster than the process of calculating, transferring delta and rebuilding target file. So this problem needs to be considered and solved maybe with some other algorithm rather than Rsync. This can a important future work.

At the same time, the problem of data modification conflicts is also needed to be figured out. In our situation, user's saving file operation starts a sync process. It is assumed that at each point of time, there is only one user's saving operation detected and claimed for the source role. But what if there are multiple users who are editing

on the same file on their local directories and they frequently edit and save the file. It could be possible that two user's saving operations were detected at the same point of time. In this situation, some mechanism is needed to be implemented to decide who should be source. And the edited content of local file should not be deleted by being synced from other users. This problem can also be solved as a future work. To make a more perfect data synchronization application, there are more details like the two problems mentioned above to be reconsidered.

Abbildungsverzeichnis

1.1 Use Case Diagram of Synchronization Application in a p2p-based Social Network . . . 7

3.1 File Block Diagram (quoted from the blog from Hao (2012)) . . . 14
3.2 Blocks comparison: rolling checksum (quoted from the blog from Hao (2012)) . . . 16
3.3 The same and diff blocks of source file compared to target file (quoted from the blog from Hao (2012)) . . . 16
3.4 Merkle Tree hierarchy of data blocks and folders under a root folder 18
3.5 Comparison and retrieval process to a Merkle Tree . . . 19
3.6 Retrieve a Merkle Tree to find the blocks of added file . . . 20
3.7 Retrieve a Merkle Tree to find the deleted block . . . 21

4.1 User classification in a sync group . . . 25
4.2 Flow chart diagram of data sync process from owner view . . . 26
4.3 Flow chart diagram of data sync process from participator view . . . 27
4.4 Working mechanism of a sync circle by detecting file status with *dll* 29
4.5 Reason of problematic deleting files with *dll* . . . 29
4.6 Simplified Merkle Tree structure used in our sync application . . . 31
4.7 Fixed and unfixed sync circles . . . 34
4.8 Working mechanism of a sync circle by detecting file modification with Merkle Hash Tree . . . 35
4.9 Sequence Diagram of Combination of Merkle Hash Tree and Rsync . 37
4.10 Work flow of user authorization of in a sync group . . . 39
4.11 Work flow of user requests to join a sync group . . . 39
4.12 Work flow of transferring a big file via framework . . . 42
4.13 Build file with parts of splited file . . . 43
4.14 User's sync group list view . . . 45
4.15 Status web view of files under local directory . . . 45
4.16 Status icons for local sync directory . . . 46
4.17 Status icons of files/folders under a sync directory . . . 46

5.1 Evaluation Environment of the Synchronization Application . . . 48
5.2 Time spent on detection data modification among different number of peers in a sync circle . . . 52

5.3 Time spent for syncing modified files among different number of peers in a sync circle 53
5.4 Transferring speed of newly added files among 3 peers 53
5.5 Time spent on detecting deleted files and deleting remote files in each sync circle 54

Literaturverzeichnis

[Hao 2012] Hao, Chen: *The core algorithms of Rsync.* http://coolshell.cn/articles/7425.html, 2012

[Szydlo 2004] Szydlo, Michael: *Merkle Tree Traversal in Log Space and Time.* Bedford, RSA Laboratories., 2004

[Tridgell 1998] Tridgell, Mackerras: *The rsync algorithm.* CA: Department of Computer Science, Australian National University., 1998