A Simple Napster Style Peer to Peer File Sharing System

Linlin Chen
A20348195
lchen96@hawk.iit.edu

January 29, 2017

Abstract

In this project, I design a simple Napster style peer to peer file sharing system. More specifically, the system has two components: a central indexing server and the peers. The central server is responsible of indexing all files stored in each peer and receive any retrieval request from peer side. Looking up its local index database and send peers owning this file to the request peer. Also if any download requested received, the serer will help build two corresponding peers. The peer is a client, which shares its local files with all other peers. Also when other peer wants to download his local file, this peer will act as a server and send the file to the corresponding peer. In this report, I will mainly introduce the design idea and any discussions about trade-off or any potential improvements.

1 Introduction

Peer-to-peer (P2P)[1] computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes. In this project, I will implement a client-server P2P model, where a central indexing server is responsible for all the contents indexing and handling all resource requests from peers. It won't locally store any data but keep the records of all peers' resources. The peer, also called as client, owns all resources. It will share its resource with all other peers. Firstly it will index all files in the central server, and request any files from the server side. He can then choose one peer owning the desired file to download. The basic P2P network structure adopted in this project

[1] https://en.wikipedia.org/wiki/Peer-to-peer

Figure 1: The P2P network structure based on the client-server model, where individual clients request services and resources from centralized servers

is shown in Fig. 1. We assume each peer can act like a server when receiving any download request from other peers, which means direct communication link occurs between any two peers.

2 Main Features

In this project, the P2P file sharing system I design has the following main features:

1. **MD5 Checker:** The server index files based on the MD5 value, instead of file name. More specifically, since two files with the same file name can have totally different contents, so I distinguish two files based on their MD5 value. Every time a client request the server to index some file, he should also report the file's MD5 value for server to determine whether the same file has been indexed. Also, MD5 has also been used to check whether the file is downloaded successfully. When a client download a file from another client, he will check the integrity of the downloaded file comparing with original MD5 value. The MD5 value can help handle the situation that a client user edit some file, the server can still update the index database.

2. **Directory Watcher:** I realize an automatic sharing directory monitor, so that once user edit a file, or create a new file, or delete a existing file, the monitor will automatically inform the server. For example, once a user edits a existing file, the contents has been changed. Since I did not only rely on the

file name to distinguish two files, and distinguish based on the MD5 value, so the client will first unregister the original file, and register the modified file. Also, once a user creates a new file, the client will automatically register it in server. Deletion will make the client automatically unregister it in server. So my system does not require explicitly interface for edition, creation or deletion in console. User can do these operations directly in file system, and the directory monitor will handle the rest of things.

3. **Bandwidth Information:** I give each client a fixed bandwidth to simulate a more real situation. Once a client seed registering request server, he should also provide his bandwidth. This did a lot of help when another client retrieve this file, the server can display all peers owning this file along with their bandwidth, so that the client can choose one peer with maximum bandwidth to download the file.

4. **Automatically Update:** The system have automatically update mechanism. Any operations like creation, edition, or deletion of a file in the sharing directory, will be reflected to the server momentarily.

5. **Multiple Tasks:** Both the server and client can handle multiple requests at the same time. Also the system takes coherency, synchronization and consistency problems seriously into account.

6. **User-friendly UI:** The system has relatively user-friendly UI. When user want to look up some file, the server can return all possible indexed file names for better choose. Also to register a new file, the UI will list all possible files for user to select.

3 Design Idea

Here I introduce each module's design idea. I will go through the design of the whole system by introducing each function module.

3.1 Indexed Database Structure

Because I want to make the server more powerful, it would not only index the file name and corresponding client id. I want the server can distinguish two different files based on their contents, instead of the file name, so i use a relatively complex structure to store the indexed data. The basic data structure has shown in Fig. 2. Firstly, I use *ConcurrentHashMap* as the basic data template, because *ConcurrentHashMap* is a hash table supporting full concurrency of retrievals and

high expected concurrency for updates. [2] It can automatically handle with the concurrency problem and allows multiple retrievals at the same time, but only one modification at a time. It is also more efficient and faster than *HashTable*, which is able to handle with concurrency problem. So in my program, I adopted *ConcurrentHashMap* as my structure to store indexed data. For the *ConcurrentHashMap*, the key is the file name, and the value is *MD52indexvalue*, which is a class I defined. It records one MD5 value, and a *ArrayList¡ indexValue¿*, recording all peers' information having this file name and also the files' MD5 values are the same. It also records the file with this MD5 value's file size. For the *indexValue*, which is also a class I defined. It can record the peer id, peer ip address, peer's bandwidth.

With so complex structure, my program can support many different operations and functions. We will show later.

3.2 Different Utilities

For the facility, I define a set of utilities to make the programming easier. Please refer to the code package *Utilities* to see each file's function.

1. *FileUtility*: This class is defined to support a lot of file operations, like CheckMD5Value, Create files, delete files, list all files, *etc.*.

2. *FileDescriptor*: This class is used for client to transmit all files' information to the server when he wants to index some file. This implements the *Serializable* interface.

3. *PeerDescriptor*: This class is used for server to transmit all peers' information to the client, when the client wants to look up some file or download some file. This also implements the *Serializable* interface.

4. *RequestMessage*: This class is used for all clients or server to send any request message. It contains a *RequestMessageType* and *Object* data.

5. *ResponseMessage*: This class is used for all clients or server to send any response message. It contains a *ResponsetMessageType* and *Object* data.

6. *RequestMessageType*: This is *enum* type, which enumerates *REGISTER, UNREGISTER, LOOKUP, LISTALLFILES, DOWNLOAD, VERIFY, DISCONNECT*, each function could be easily inferred from the name. It is used in request message, to infer the message type.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

7. *ResponseMessageType*: This is *enum* type, which enumerates *SUCCESS, FAILURE, WRONGREQUEST, INTERACTION*. It is used in response message, to infer the message type.

3.3 Index Server

As said in introduction, the server is used to index all files in the clients and when receiving any look up request or downloading request, it will search the indexed database and feedback all the peers having this file. It does not locally store any data. Here I used multiple threading programming to ensure the server can concurrently handle with multiple requests from clients. In my project, I used socket programming to realize the communication between server and clients. Here I adopt port 8888 for server to listen any requests from clients. Once receiving any *RequestMessage* from clients, it will do the corresponding operations based on the *RequestMessageType*. Then I introduce each operation based on *RequestMessageType*.

1. REGISTER: For register request, the server will check whether this file name has been indexed. It not, it just directly create (key, value) pair into the ConcurrentHashMap; If so, the server need to determine whether the indexed files are exactly the same with this new file. It will first compare the new file's MD5 value with all indexed file with the same file name. If not found a match, it will create a new (key, pair) in *MD52indexvalue*. If the server found a match, it need to firstly to check whether it's a repeat register request from the same client. The server would not index the same file from the same clients for multiple times. So if this file has been registered, it would not register it again. If not, it then will add this to the ArrayList according to the MD5 value.

2. UNREGISTER: To unregister a file, the server will firstly look up whether this file has been registered. If not, nothing would happen; if so, then it will match the peer id and peer ip address with the requested server, then remove that entry.

3. LOOKUP: To look up a file, the server will search the whole indexed database. Differently here, it will firstly determine whether this file name has been registered. If not, it will return FAILURE; if so, then it need to compare whether there exist different types of file with this file name. For example, for file a.txt, there could be 2 types of the file, one 1KB and another 10MB, both have the file name a.txt. Then server will return a *INTERACTION* request with the client, to let the client user choose which file size to look up. Based

on user's choice, then the server will return all the peers' information with this kind of file. If there is only one type of file having this file name, the server will directly return all the peers having this file.

4. LISTALLFILES: This method is to facilitate the users. When user want to lookup one file or download one file, the server will firstly list all the files' name he has indexed for the users to choose. Users can actually input the new file name as well. So this method is used to list all the file names which have been indexed in the server.

5. DOWNLOAD: This method is used for client to download a file. The server will do very similar thing with LOOKUP, like if there exists different types of files having the requested file name, then it will let user to choose one to download; if not, it will directly return all the peers having this file. The response message contains all the peers' information.

6. DISCONNECT: This is sent by client to ask disconnect with the server. The server will close the socket with the client.

3.4 Client

For the client, I implement a folder watcher to watch every operation happened in the monitored folder. More specifically, it will monitor three kinds of operations — *creation, deletion* and *modification*. For a *creation* operation, the client will automatically send a register request to the server to index this newly created file; For a *deletion* operation, the client will automatically send a unregister request to the server to unregister this deleted file; For a *modification* operation, since in my program, the server will distinguish two files based on the MD5 value, instead of the file name; So once one file is modified, even though its file name remains unchanged, the MD5 value has changed. We treat these two files as two different ones. So once one file is modified, the client will firstly send an unregister request to the server, to unregister the old version file. Then it will register the newly modified file in the server. The file watcher makes sure all the files in the sharing folder are up to data.

I think there is no technical challenges to realize the register, unregister, lookup and download method. So I won't to cover these realizations.

One thing should be noticed is that the client can also acts like a server for other clients to download the file. Here I realize this using multithreading coding, to ensure the client can handle multiple requests from clients at the same time. A challenge here is that I did not want to transmit each client's port when they acts like a server, or hard coding the port number in the code. Because when a user

want to download one file, it need to look up from the server firstly, and then the server will return the all the peers' information having this file to the client. The peers' information contains each peer's client ID and IP address. So I use the client ID to help locate the port of each client. I set a *base_port*, 10000, and each client's port when they acts like a server would be $base\_port + clientID$. Because all the clients agree with the same base port, and they know the target client's id, so they can easily connect the socket communication with the target client.

4 Conclusion

Before concluding, I will discuss some potential improvements.

- **Authentication:** There is no authentication when one client connects to the server or another client. This could raise some security issues. For example, any one know the server's ip address and port number can implement a lot of attacks. Then same thing could happen to the client, as he also acts as a server.

  The solution to this one is to ask every client creates a user name and password. Each time they want to build connections with server or client, he need to provide the user name and password. If it passes the verification, then the server approves the request.

- **Replication**: In my program, I did not realize any data replication. There could be some potential problems. Firstly, if one client's data get lost and only he has this data, then other clients cannot access this data any more; or if one client gets off and as well, only he has this data, then other clients cannot connect to this client and data download request cannot be finished.

  The solution to this one is very simple. When one client using this service, the server will ask whether he wants to replicate some data. If he is willing to, then the server will distribute some data from other clients to this one, to make sure, all the data indexed in the server, has at least two replications.

This project is very interesting. I realized a peer to peer file sharing system and tried to add more features and tried to simulate a real situation. I am now very familiar with Socket coding, multiple threading coding and how to handle concurrency and event handling.

```
private static ConcurrentHashMap <String, HashMap<String, MD52indexvalue>> indexedDatabase
= new ConcurrentHashMap<String, HashMap<String, MD52indexvalue>> ();


        /*
         * define the value for indexed contents
         */
        private static class indexValue{
                int peer_id;     //peer id
                String ip_addr; //peer address
                int bandwidth;  //peer network bandwidth
                double filesize;        //size of the file

                public indexValue(int peer_id, String ip_addr, int bandwidth){
                        this.peer_id = peer_id;
                        this.ip_addr = ip_addr;
                        this.bandwidth = bandwidth;
                        //this.filesize = filesize;
                }
                public boolean containsPeer (int peer_id, String ip_addr) {
                        if (this.peer_id == peer_id && this.ip_addr.equalsIgnoreCase(ip_addr)) {
                                return true;
                        } else {
                                return false;
                        }
                }
        }


                /*
         * MD5 stored as key, index value as value
         */
        private static class MD52indexvalue {
                //md5 as key, peer info as value
                 private ArrayList< indexValue> md5_info;
                 private String MD5value;
                 private double filesize;

                 //number of peers having this file
                 private static int PEER_OWNING_NUM = 0;

                 public MD52indexvalue
                 (String md5value, int peer_id, String ip_addr, int bandwidth, double filesize) {
                        this.MD5value = md5value;
                        this.filesize = filesize;
                        md5_info = new ArrayList<indexValue> ();
                        indexValue iv = new indexValue (peer_id, ip_addr, bandwidth);
                        md5_info.add(iv);
                        PEER_OWNING_NUM ++;
                        //md5_info.put(this.MD5value, al);
                 }

        }
```

Figure 2: Data structure for Indexed Database