Ekin Akyürek
Mehmet Kaan Bozkır
COMP416 Project1: iSync Cloud

Problem Description:

The most general and shallow definition we can give to the given problem could be "to develop a unidirectional client-server relationship, which synchronizes the files in clients' local storages, based on the server storage". The main expected functionalities beside the multi-client support, were syncing all files, syncing particular files or checking if a sync is necessary for each client.

One of the most important features we had to develop is the multithreading property. One of the hardest parts of this assignment was that our "cloud server" had to support multiple clients simultaneously, therefore we could not just use one socket connection between server and the client, and we had to use another port for data stream, for each client. In the following parts of the report, all structures that we used will be explained further.

Overview of the classes we designed:

For this project, the backbone of our source codes were laid out by the TA's, therefore fortunately we did not have to design the project from scratch. We may begin by explaining the Server part of our source code.

We implemented variables from types Socket and ServerSocket. ServerSocket waits for requests from the network, and uses that request to revoke certain actions (e.g. starting a new thread) in the Server. We assigned a certain unused port number to this ServerSocket.

We also implemented the aforementioned multithreading property in ServerThread class. By writing "extends Thread", we can assure that the compiler knows this class is a multi-thread related to main Server class. We declared the other necessary variables in the Thread, these being the BufferedReader, to read Client requests (e.g. "sync all", "sync check" etc.), PrintWriter, to output the system messages and to send the specific port number of the data stream to the Client, `ObjectInputStream` for receiving hashmap from the Client, `BufferedOutputStream` to send files in packets.

The '`while(true)`' loop in the `main()` function of the Server class, the multithreading property is put to use. Firstly, at the line `ss.accept()`, the serverSocket waits for a connection to be made to it, and accepts it. We display a message, ensuring that the client-server connection is established. Then, by using a constructor, we declare a new ServerThread variable, and by using the function "`start()`", we run that thread.

The line `i=i+1` increases i by one at every iteration and that allows us to construct a thread, which has a specific client number "i". Therefore, we can assign a new datastream ServerSocket for each client.

After the connection between the server and client is established by using the `dataPort=dataStream.accept();`

line, the server now transfer files to each client via different sockets . We have defined four different commands for our Client-Server protocol; which are sync check, sync all, sync <file> and QUIT.

`sync <file>:` This function is a specific file syncing function. It receives the name of the desired file and searches for the equivalent of the file in the "Storage" folder of the server. If a file with the same name is found on the "Storage" of the server, then it receives the hash name of the file in the "Local" storage of the Client and compares that hash name with the hash name of the file in the "Storage" of the server. If the hash names are a match, it means that the file has not been updated at the server, and there is no need to update the file at the "Local". However, if the hash names are different, then it means that the file has been updated on the Server, but no changes are made on the Local. In this case, we send and replace the file from Storage to Local via `sendFile(ourFile)*;` line. If the file is not present at the Storage of the server, it sends a message to the client, saying that the file does not exist. We did not decide to add a deleting function here, to prevent the user from accidentally deleting a file on its local folder.

* Using `FileInputStream`,we first read the file in packets of 8096 bytes, then using `BufferedOutputStream`, we send the file to the Client. Also in the for loop, we make sure all of the file is sent to the Client properly.

`sync all` : This function is designed to completely update the Local folder with the content of Storage folder. It compares the file names as usual. If the file is not present in the Local folder, it sends that file to the Client, and likewise, it removes all files that are not present in the Storage folder, since those files should not be there after the synchronization. If the file names are the same, again, it compares the hash names to see if the file is the latest updated version; the file is updated if the hash names differ, and left if the hash names are the same.

`sync check:` This function does not actually interfere with the download and the uploading of the files, however, it locates the files that need to be updated, uploaded or removed at the "Local" storage of the client.

If the Client enters the input "OUIT", the server-client connection is exterminated.

Let us also examine the Client part of the code:

Similar to the Server part, we had to use variables of types Socket, BufferedReader and PrintWriter, for establishing socket connections, reading filenames and hashes and outputting messages. One important point we had to take into consideration was that the port number that we assign to our Socket variable s, had to carry the same port number as the server, demonstrated in the line `s=new Socket(address, 4445);`

The client part of the code is much simpler, since it is mainly used to fetch the commands of the Client and transmit those commands to the server. Firstly, a welcoming message is displayed on the console saying "` Enter Data to echo Server ( Enter QUIT to end):`", then the user at the client side is expected to give a valid command, such as "sync all", "sync 'file'", "sync check" or "QUIT".

If the client is asking to sync a specific file, and enters the command "sync <filename>", the program also identifies the desired file using the line:

```
else if(line.length()>5 && line.substring(0,4).compareTo("sync")==0
&& line.charAt(5)=='<' && line.charAt(line.length()-1)=='>')
```

and

```
String fileName =
line.substring(line.indexOf("<")+1,line.indexOf(">"));
```

Hence, we extract the name of the specific file.

Implementing hash generation function:

As suggested by our Teaching Assistants, we used the MD5 algorithm to generate hashes for our files. We later found out that this was an algorithm designed by an MIT professor in 1994, and it was used by many companies for hashing purposes.

We constructed a function called getHash(), which inputs the name of the file in String type, and outputs the byte array "digest", which is the hash name in byte array form. We also converted this byte array into hexadecimal human readable format. To get the hash names of files, we used the MessageDigest method, and specified the algorithm MD5 like demonstrated below.

```
private byte [] getHash(String filename) throws Exception{

      MessageDigest md = MessageDigest.getInstance("MD5");
      try (InputStream is =
Files.newInputStream(Paths.get("./Storage/"+filename))){
        DigestInputStream dis = new DigestInputStream(is, md);
      }
      byte[] digest = md.digest();

      return digest  ;
        }
```

Converting byte array hash into human readable format, hexadecimal:

```
StringBuffer hexString = new StringBuffer();
     for (int i=0;i<mdbytes.length;i++) {
          String hex=Integer.toHexString(0xff & mdbytes[i]);
          if(hex.length()==1) hexString.append('0');
          hexString.append(hex);
```

Results and Conclusions

An immediate result regarding this project was that we learned that Socket Programming includes extensive programming command over the language, since there are specific methods and functions that you can implement in certain ways. We have also faced lots of incompatibility problems due to the different IDE's that we used, and even after using the same IDE, we encountered errors related to the Java SDK version difference on the two computers, however, thorough troubleshooting allowed us to get over these problems.

We also saw that it requires a lot of effort to transmit big files over the network, since we thought it was more efficient to send the data in pieces. Thus, we can support for over 2GB files also. Another important property of the server-client model that we recognized was the security. In this project, we have not taken any measures to secure the data transfer between the server and the client. If we were to publicize our project, it would soon be raided by lots of security breaches. Therefore it is also important to sustain the safety of the operations as well as performing the data transfer.

In this project, we have researched and learned how to set up a basic Server-Client model and how to implement Socket Programming to create a cloud-like storage. Our model uploads, downloads, updates and removes files on the Local storage on Client's computer, and keeps the client updated if the client requests. It detects whether or not the Local storage files are altered, and if they are, replaces them with the original files from the server storage. We can support for 2