CS 550 Programming Assignment 1

Ankur Saikia (A20445640)

Design Document

The application consists of a **server** and a **client** and can be used to download files from the server directory to the client directory.
The protocol used to share data is TCP/IP which uses packets to transfer the data and is a message oriented form of communication.

Server design:

The server uses multiple processes to concurrently connect to multiple clients. We instantiate a **socket** and then **bind** it to the user provided Port. After binding we listen to clients using the **listen**() command. This kind of socket is called a passive socket since the connection with client is not yet done. The server needs to call **accept**() in order to be able to connect to a client.
Now, for our server, after listen() command, we call an infinite while loop which consists of the accept() command.
The moment a client connects to the server, we call **fork**() which creates another process and then call **connectToClient**() method from the child process. This **frees the main process** to accept another clients trying to connect to the server. The server also sends the file listing with their md5's to the client.
The **connectToClient**() method receives client commands and acts accordingly. It receives the file numbers to send as requested by the client and sends then sequentially or parallely which is also based on the user input. After completely sending the files, the server waits for client to validate the md5s and then send back any file numbers that need to be re-downloaded. If no files need re-downloading the process exits. When files are sent sequentially, threads are not required. Additionally, we also don't need to send additional headers in the data sent to client. But when files are to be sent in parallel, we create as many threads as the number of files. In this scenario, we also include a header (which is the file number in our case) while sending data to client through TCP. The client would be receiving multiple data streams at the same time and in this case and would have to differentiate which packet belongs to which file based on the header.
In each of these cases, the logic is as follows:

- Client sends the number of files to download to the server.
- Client then sends the download logic: sequential or parallel (If 1 is sent then sequential else parallel)
- Now the logic changes for sequential and parallel.
- If the download logic is sequential, the server would wait for the client to sent the file number.
- Once a single file number is sent, the server would in return send back the file size.
- Once the client receives the file size, it starts a while loop to receive file data in chunks of size 1024. The server has a similar logic, but in this case it opens the file and sends file data in chunks of size 1024.
- If the download logic is parallel, the client would first send all the file numbers required and the server would in turn send back the sizes of all the files.
- Once the client has received all the file sizes, it would create those files and keep them in memory.

- The server would then spin up threads for each file and start sending the file data. The client similarly would check the remaining data and if there is pending data it would create a thread to receive it.
- There is an additional for loop which determines whether re-downloads are required. The whole logic of downloading files starts again if client sends the number of files remaining again.
- For both of these cases, we log the requests in log.txt. The time to send is logged in time.txt.

Running the server:

make run-server PORT={port_number}

example:

make run-server PORT=50017

Client design:

Each client is a single threaded process. Every time a client is connected to the server, the client receives the files list which also contains the file md5's. The user is then asked to input which files to download and also how to download those files.
Once the user input is provided, the client would send the file numbers to the server. The server would send back the size of each file.
After receiving the file size, the client would run a while loop till size is 0 and all the file data is received.

- When the files are to be downloaded in a sequence, file numbers are sent in sequence and the second file number is only sent if the first file has completed downloading.
- For parallel downloads, we request file sizes of all the files at once and store them in a list. We have an infinite loop that checks the sum of sizes in each iteration using the **isPending**() method. If the sum is greater than zero, that means there is still pending data to be received and hence we create a new thread to receive the data.
- We use locking mechanisms when downloading files parallely. There are two locks we use.
- The first lock is process level which is used when we subtract the file sizes from the list of sizes or when we calculate isPending().
- The other lock is file level, which is present for each file. Whenever we determine a file to write to, we hold the lock before writing. The we write to the file and finally release the lock.
- For both parallel and sequential downloads, after the download is complete, we check the md5 and compare it with the one which was received along with the file list. If the md5's donot match we retry again on user intervention.

Running the client:

make run-client PORT={port_number} FOLDER_INDEX={folder_index} AUTO={automated?}

{**port_number**} : This port number should be same as that of server

{**folder_index**} : Setting this to any index would result in the data being saved in the respective client folder. e.g. **FOLDER_INDEX=2** would save the client downloaded files in ***ClientFolder_2*** folder.

{**automated?**} : **AUTO=1** & **AUTO=2** would make the client not ask for user input and would download all files from server directory, serially and parallely respectively. **AUTO=0** would ask for user input when running the client

example:

make run-client PORT=50017 FOLDER_INDEX=1 AUTO=0

Design Considerations:

For maintaining a list of files in directory:

- This is done only once at the start of the server, in order to update the files list again, we need to restart the server. By doing this we can maintain a single data structure for all the processes or clients. Else we need to instantiate a new data structure each time a client connects to the server and pass it to the child process in the connectToClient() method.
- This was selected because it uses lot less space than maintaining unique data structure for each client.
- The whole data structure is sent to client on connect. This data structure also contains the md5 as calculated by the server. After the client downloads the file, it will independently calculate themd5 hash for the file and compare it to the one sent by the server.

Logging

- Logs are saved to **log.txt**. Information saved: FileName, Size (in bytes), TypeOfDownload (Sequence or Parallel), ClientDescriptor, ClientNumber (since start of server)
- Time to send are saved in **time.txt**. The times are in seconds.
- The errors are logged in the standard output of the console where it is running.

Re-downloads by client

- Re-downloading by client is done through user intervention. After downloading all the requested files, the client compares the md5's and creates a new list with the file numbers that require re-download. On user confirmation, the client again requests the server for the files and downloads them.

Scaling to 64 clients

- This is done through the **stepTest.sh** script. This script runs the server and connects clients starting from 8 clients till 64 clients increasing 4 clients in each loop.