**File Server and Client**

Programming Assignment 1

Abheek Mondal

A205438046

CS 550

Monday, 20 February 2023

This assignment is a client-server mode file downloading system. It involves the creation of a client and server program. The server hosts a list of files for the client to download. The client will get the file list from the server and then request for downloading one or more files of them from the server. Scale the implementation of client to evaluate the performance of the server.

There are three main program files that I created:

**Server.py**

‘server.py’ code is a simple implementation of a file server that listens for incoming client connections on a specified IP and port, and responds to two types of commands: **get\_files\_list** and download.

When a client sends the **get\_files\_list** command, the server responds with a list of all the files in a specified directory along with their sizes. When a client sends the download command followed by a file name, the server checks if the file exists in the directory and sends the file to the client if it does.

The server also computes the hash value of the file using the SHA-256 hashing algorithm before sending it to the client to ensure the integrity of the file during transmission.

**ClientThread** class extends **threading.Thread** class. Each time a new client connects to the server, a new instance of the **ClientThread** class is created to manage the client's requests in a separate thread. This allows multiple clients to connect and download different files simultaneously without blocking each other.

The **ClientThread** class contains a run method that handles the client's requests. When the client sends a "download" command with a filename, the **ClientThread** class checks if the file exists in the server's file directory. If it does, it reads the file's content and sends it to the client. If the file doesn't exist, it sends an error message to the client.

The **get\_file\_list** method is used to retrieve a list of files available on the server. It is called when the client sends a "**get\_files\_list**" command to the server.

The Server class has been modified to listen for incoming connections and create a new **ClientThread** instance to handle each client request in a separate thread. The start method is responsible for setting up the server socket and listening for incoming connections.

The **send\_file** method is a helper method that sends the file to the client. It is run on a separate thread for each download request. When a download command is received, we create a new thread and pass the **client\_socket** and **file\_path** to the **send\_file** method. The **send\_file** method then sends the file to the client. This allows multiple clients to download different files at the same time.

**client.py**

This code is a simple server program that allows clients to download files from a server directory. The server class has a "start" function that listens for connections and creates a new thread for each client. Each client thread manages the client's requests, which include asking for a list of files available for download and downloading a specified file.

The "**ClientThread**" class manages the requests from a single client. The "run" function is called when a thread is started and waits for commands from the client. If the command is "**get\_files\_list**", the thread retrieves a list of files available for download and sends it back to the client. If the command is "download", the thread checks if the specified file exists in the server directory and sends the file data back to the client if it does.

The "**get\_file\_list**" function in the "**ClientThread**" class retrieves a list of files in the server directory and their sizes. The "Server" class has an IP address, port, and directory path attributes that are used to create the server socket and specify the server directory. The program starts the server by creating a "Server" object and calling its "start" function.

The client program has the following commands:

1: Get list of files hosted by the server

2: Download file(s) from the server

3: Exit the program

When 1 is entered, the client will get a list of files hosted by the server and print them to the console.

When 2 is entered, the client will prompt the user to enter one or more filenames to download, separated by commas. The files will be downloaded one at a time and their checksums will be printed to the console. The total download time will also be printed to the console.

When 3 is entered, the client program will exit.

**subprocess\_test.py**

This code is downloading a file using multiple subprocesses. The download file function records the time taken to download a file and writes it to a file. In the main function, a list of files is created, and for each file, a subprocess is created to download the file. The subprocesses are added to a list of threads. After all the subprocesses have been started, the download time is recorded for each thread. Finally, the total time taken for all the downloads is calculated and recorded.

**Program Analysis**

server.py

The server.py program is designed to create a server that listens on a specified IP address and port number for incoming client connections. Upon receiving a connection, a new thread is created to handle the client request. The program provides two main functionalities: get the list of files in the server database and download a specific file from the server.

The program uses the socket library to establish a network socket and listen for incoming connections. It creates a new ClientThread object for each incoming connection, passing the client socket, client address, and the server file directory as arguments. The ClientThread class defines the run() method that manages the client request. The method first receives data from the client, splits it into tokens, and determines the type of command. If the command is to get the list of files, it calls the get\_file\_list() method to retrieve the list of files and sends it back to the client in JSON format. If the command is to download a file, it checks if the file exists in the server directory and sends it back to the client if it exists. Otherwise, it sends an error message to the client.

The program uses threading to manage multiple client requests concurrently. However, it does not limit the number of threads that can be created, which may result in performance degradation or resource exhaustion under heavy load.

client.py

The client.py program is designed to connect to the server.py program and request either the list of files or a specific file. The program prompts the user to enter the server IP address and port number, as well as the command type (get\_files\_list or download) and the filename (if downloading a file).

The program uses the socket library to establish a connection to the server and sends a command string containing the command type and filename (if applicable) to the server. It then waits for a response from the server and prints the response to the console.

The program does not have any error handling or input validation, which may cause it to crash or produce unexpected results if the user enters invalid input.

subprocess\_test.py

The subprocess\_test.py program is designed to test the subprocess module in Python. It creates a specified number of subprocesses, each of which runs a specified command (in this case, the client.py program with randomly chosen command arguments). The program records the start and end time for each subprocess and writes the total time to a file.

The program uses the subprocess module to create a new process for each subprocess. It passes the command and arguments as a list of strings to the subprocess.Popen() method, which returns a subprocess object. The program uses the wait() method of the subprocess object to wait for the subprocess to finish before recording the end time and writing the download time to a file.

The program uses threading to oversee multiple subprocesses concurrently, which may improve performance by allowing multiple subprocesses to run in parallel. However, it does not limit the number of threads that can be created, which may result in performance degradation or resource exhaustion under heavy load. Additionally, the program does not perform any error handling or validation of the command arguments, which may result in unexpected behavior if invalid arguments are passed to the subprocess.

**Time Analysis**

|  |  |
| --- | --- |
| File size (bits) | Time Taken (Seconds) |
| 128 | 0.00100 |
| 512 | 0.00101 |
| 2000 | 0.00104 |
| 8000 | 0.00099 |
| 32000 | 0.00135 |

This analysis indicates a clear linear correlation between file size and download time, with one exception. Surprisingly, the file with a size of 8000 bits took less time to download than any other file, although this may have been due to chance. Upon repeating the test, the same trend was observed. The underlying cause of this outlier is difficult to ascertain, and could be due to a number of factors, such as background processes or resource allocation. However, without conducting a retest in an isolated system with only this program running, it is impossible to confirm any potential causes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | 128 | 512 | 2k | 8k | 32k |
| 2 | 0.166 | 0.164 | 0.164 | 0.164 | 0.166 |
| 4 | 0.268 | 0.268 | 0.269 | 0.267 | 0.266 |
| 8 | 0.482 | 0.483 | 0.487 | 0.486 | 0.482 |
| 16 | 0.924 | 0.914 | 0.912 | 0.911 | 0.912 |

The analysis of the graph indicates that there is a positive correlation between the number of processes and the duration of file downloads, which is further amplified by larger file sizes. The clustering of data points suggests a low variability in the download speeds across the different processes, indicating consistent download times across the varying number of processes. It is worth noting that the results could be influenced by the limited range of processes tested, highlighting the need for more sophisticated testing with a larger number of processes and dedicated hardware to confirm the hypothesis.

**Conclusion**

Essentially, my code worked after much difficulty. It works as intended