Report

Hubert Bao – 169077248 Adnan Awad - 169028425

# Brief Description

## Code

This code allows up to 3 clients to communicate simultaneously with a TCP server. The server handles client requests through this connection and logs the client activities.

## Steps

Initialization:

* Sets up a TCP “serverSocket”, binds it to a specified port, and listens for incoming connections (maximum of 3 clients).
* Initializes a list “avaliableClients” to track available client slots.
* Initializes a dictionary “clientLogs” to track client logs.

Connection Loop:

* Continuously waits for client connections, timeout every 5 seconds.
* Upon client connection request, initialize a “connectionSocket” with client, assigns a client number, and starts a new thread under “handle\_client()” function to handle client messages.

Client Handle:

* Client slots, client log, and directory are stored globally and can be modified by all connection threads.
* Create or update client log for every new connection, notifies clients of their client number, and enters message loop.
* Message loop responds continuously to client commands until client exits.

Client Side:

* Input loop continues to receive and deliver commands to server.
* Gracefully terminate on user exits or exception.

## Operation

Client communicates with the server via 4 commands:

1. “exit”: Disconnects the clients and logs the disconnection.
2. “status”: Returns a JSON string with all cached and current client connections.
3. “list”: Returns a JSON string of accessible text files in the server directory.
4. “get <filename.txt>”: Returns the content of the specified text file or an error message if the file does not exist.

## Special consideration

1. Server timeouts every 5 seconds to check for ctrl-c keystroke termination.
2. Server can only read file encoded in “utf-8”.
3. Server automatically closes all thread on sys.exit().
4. Client receives up to 2MB.
5. Server multi-threads each client connection.
6. Server and client handles exceptions and gracefully shuts down.
7. Uses JSON for data serialization to send structured responses.

# Difficulties

1. When client rejoins, it changes existing record rather than appending a new one

The “dictionary[“key”] = value” would assign a new key-value pair if it doesn’t already exist or assign a new value for an existing key. So, if a client rejoins, then this command won’t append a new log into the list but change the existing log. However, if only “list.append” is used, then it cannot append to a non-existent list if client log wasn’t initialized.

Solution: Create a new dictionary key if it doesn't already exist, and only use append to modify values. If a new client is added, then run the create new dictionary before appending. Otherwise, ignore the if statement and only append.

1. Race condition on the log when clients are multi-threaded

If race condition exist, use “lock = Lock()” and add a lock to some lines. Although no race condition exists because of two reasons. First, every single client log is unique to each client, so no two clients modify the same data at once. Second, no two clients share the same client number, so they cannot append to the same list in the same time.

1. Client with lower client number exits before higher ones

When using a “currentClient = 0” to track available client number where it increments when a new client joins and decrements when it leaves, it does not work. Say “currentClient = 3” when there are 3 clients, and client number 1 leaves; then, the new client who joins should be client number 1, not 3 – 1 = 2.

Solution: Use an ordered list to store client spots where 1 represents full and 0 represents empty. In which case list tracks exact client spots that are available. Use “list.index(0)” to find the next available spot.

1. “serverSocket.accpet()” blocks call; program cannot terminate using Ctrl-C

When “serverSocket” is waiting for a new client connection, it blocks all console commands. In other words, user cannot run any console commands on the terminal.

Solution: Set “serverSocket.settimeout(5)” that timeouts “.accept()” every 5 seconds. During which all console command inputted will be responded to after the timeout. The “TimeoutError” itself does not terminate the waiting for connection loop, it just cycles the connection to be more flexible.

1. “sys.exit()” does not terminate program and waits for non-daemon threads to finish

In general, system can only exit after all the thread process are completed. This way, the program can perform all intended clean ups before exiting. In this case, however, client connection is continuous and does not complete.

Solution: whereas non-daemon threads does not allow forced termination, when thread is set as “Thread(daemon=True)” however, it allows “sys.exit()” to automatically terminate all threads. This way, server can shut down without waiting. The impact is minimal as clients simply loses connection.

1. \*All other difficulties not mentioned here are not fixed and will be listed in “Possible Improvements” section

# Test Result

## Test steps

* Initiate Server
* Initiate Client01
  + Run “message” with ACK return
  + Run “a#.%” with ACK return
* Initiate Client02
* Initiate Client03
* Initiate Client04 – failure on timeout
* Client02 run “status”
  + Run “exit” – graceful shut down
  + Rejoin
  + Run “status” again – 2 clients should be logged for Client02
* Client03 run “list”
  + Run “get asdf” – generate error
  + Run “get <filename.txt>” – display file content
* Ctrl-C on Server – gracefully shut down

## Console messages

### Server

PS C:\Users\huber\Projects\CP372\a1> cd src

PS C:\Users\huber\Projects\CP372\a1\src> python Server.py

Server listening on 0.0.0.0:6789

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Client01 joined!

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Client02 joined!

Waiting for connection...

Waiting for connection...

Waiting for connection...

Client03 joined!

Waiting for connection...

Waiting for connection...

Waiting for connection...

Server full: Client join failed

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Client02 left!

Waiting for connection...

Client02 joined!

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Client03 accessed short\_test\_file.txt

Waiting for connection...

Waiting for connection...

Waiting for connection...

Waiting for connection...

Server shut down

PS C:\Users\huber\Projects\CP372\a1\src>

### Client01

PS C:\Users\huber\Projects\CP372\a1> cd src

PS C:\Users\huber\Projects\CP372\a1\src> python Client.py

You are Client01

Enter message: message

Server response: message ACK

Enter message: a#.%

Server response: a#.% ACK

### Client02

Enter message:

PS C:\Users\huber\Projects\CP372\a1> cd src

PS C:\Users\huber\Projects\CP372\a1\src> python Client.py

You are Client02

Enter message: status

Server response: {

"Client01": [

{

"address": [

"127.0.0.1",

51126

],

"connected\_at": "2024-10-13 10:54:38.043892",

"disconnected\_at": "Currently Active"

}

],

"Client02": [

{

"address": [

"127.0.0.1",

51130

],

"connected\_at": "2024-10-13 10:55:02.187017",

"disconnected\_at": "Currently Active"

}

],

"Client03": [

{

"address": [

"127.0.0.1",

51131

],

"connected\_at": "2024-10-13 10:55:15.799576",

"disconnected\_at": "Currently Active"

}

]

}

Enter message: exit

PS C:\Users\huber\Projects\CP372\a1\src> python Client.py

You are Client02

Enter message: status

Server response: {

"Client01": [

{

"address": [

"127.0.0.1",

51126

],

"connected\_at": "2024-10-13 10:54:38.043892",

"disconnected\_at": "Currently Active"

}

],

"Client02": [

{

"address": [

"127.0.0.1",

51130

],

"connected\_at": "2024-10-13 10:55:02.187017",

"disconnected\_at": "2024-10-13 10:55:57.111386"

},

{

"address": [

"127.0.0.1",

51135

],

"connected\_at": "2024-10-13 10:56:02.688362",

"disconnected\_at": "Currently Active"

}

],

"Client03": [

{

"address": [

"127.0.0.1",

51131

],

"connected\_at": "2024-10-13 10:55:15.799576",

"disconnected\_at": "Currently Active"

}

]

}

Enter message:

### Client03

PS C:\Users\huber\Projects\CP372\a1> cd src

PS C:\Users\huber\Projects\CP372\a1\src> python Client.py

You are Client03

Enter message: list

Server response: [

"alice.txt",

"greatexpectations.txt",

"mobydick.txt",

"prideandprejudice.txt",

"short\_test\_file.txt"

]

Enter message: get asdf

Server response: Invalid filename, please try again

Enter message: get short\_test\_file.txt

Server response: Txt file imported successfully.

Enter message:

### Client04 – fail

PS C:\Users\huber\Projects\CP372\a1> cd src

PS C:\Users\huber\Projects\CP372\a1\src> python Client.py

Timeout disconnect

PS C:\Users\huber\Projects\CP372\a1\src>

# Possible Improvements

Possible improvements: If you had given more time, what would you add or do differently?

1. Using Ctrl-C to terminate server is not standard procedure, as Ctrl-C is intended for forced termination.

Multi-threading Server console input and Server connection waiting loop; thus, it allows user to enter user input on Server console to quit program.

1. Client input still waits for user input even when server connection is already lost.

Multi-threading Client console input and Client connection waiting loop; thus, even when user input holds the main thread, the connection thread continues to check for connection. This way, connection thread could notify main thread of disconnection.

1. Enhance code organisation
   1. Seperation of concern using classes and methods
   2. Add documentations to said classes and methods
2. Transfer large files in multiple transfers instead of one
3. Storing log cache in a file so it can be re-read in a new Server