CP372 - Programming Assignment

CP-372-B

Instructor: Lilatul Ferdouse

Kunal Gandhi, Muzzi Khan

169051546, 169073561

Due: October 20th, 2025

Introduction / Brief Description

First, our program opens a TCP server, which is multi-threaded and self-initializes on the address of localhost (127.0.0.1) and port 50000. Clients may be launched on the same machine, and each client attempts to connect to the server using the same localhost address. The server will only accept connections when there are fewer than three clients connected, and the hard-coded concurrency limit is checked.

Once a successful connection is made, the client is assigned a unique identifier by the server in the format Client##, with the beginning being client01. The client initiates a greeting message in order to complete the handshake and ensure successful registration. From that point onward the client is able to talk using a command-line interface (CLI) and sends messages to the server which is in turn echoed and responds with an "ACK" indicating that the server has received the message.

Specialized text inputs are considered commands and processed by a command handler on the server. As an example, typing status will ask the server to give its current list of connected and disconnected clients, with the connection and disconnection times of each client. The list command directs the server to give the names of all files that it has in its repository folder. In case the client issues the command get [file_name], the server transmits that file back and sends the client a hash of the file using SHA-256 to enable the client to certify the integrity of the file. Other utility commands are: help, about, who, ping, and uptime, which are mainly utilized in testing responsiveness as well as project metadata. Lastly, the exit command enables the client to close out the connection with the server in a proper manner and leave a slot open to receive any other incoming client.

Whenever a client is connected or disconnected, the server makes updates on an in-memory cache with the name of the client, the connection (ACCEPTED) time, and the disconnection (FINISHED) time. All major events, including connections, file transfers, and terminations are all logged to the console by the server, which includes timestamps.

Client-side information is contained in a simple Client class that is easier to maintain and reference information across threads. This architecture will enable the server to monitor, authenticate, and act upon every client effectively.

Walk Through

The server is launched by running the command python3 Server.py, which launches the program on the localhost address (127.0.0.1) and starts listening on port 50000. The console is used to ascertain that the server is active and is ready to accept the maximum number of client connections, which in our case would be 3.

Python3 Client.py is then used to launch clients and each client will automatically be named with a unique name starting with Client01. When three clients are already connected, further connections will lead to the message that the server is at full capacity.

After connection, the client is greeted by the server and is granted access to the command-line interface. Using this interface, the client is able to send messages or commands which are received by the server and processed and then recognized by the server resulting in an ACK. The status command is used to see the details of all the connections, list command is used to see the available files in the server repository and get sample.txt command is used to download a file to the local directory of downloads of the client with a SHA-256 check which confirms integrity.

Threading enables the server to serve a number of different clients at the same time, so that data can be exchanged without delays. Upon receiving a client entry of the word exit, the server responds with BYE, modifies its cache to make a finished timestamp, and frees the connection slot to accept new clients.

Difficulties Faced

The initial problem that was faced was the server would reprimand a port already in use error when trying to restart the program. This was the case since the port had been occupied by an earlier process. To resolve this the command lsof -iTCP:50000 was used to find out the process ID and SO REUSEADDR was added in the code to enable the port to reuse without waiting.

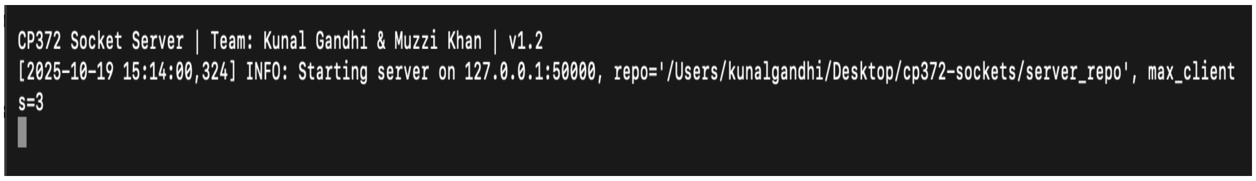
The other challenge was the thread synchronization since different clients using the same data at the same time resulted in variations in the cache. This was solved with the help of threading. Lock and Semaphore functions of Python to allow the access to common variables and restrict the number of active connections to its maximum value.

Cases were also witnessed where the server crashed when a client requested a file which did not exist or when the client stopped without any notice. To overcome this, input validation was included to check file paths by transferring them and other checks were put in place so as to make sure that only active sockets receive data sent by the server.

Lastly, the macOS testing also brought about small file path problems which were solved by utilizing the os.path.join() to properly manipulate both operating system directory paths.

Test Cases:

1. Server Startup

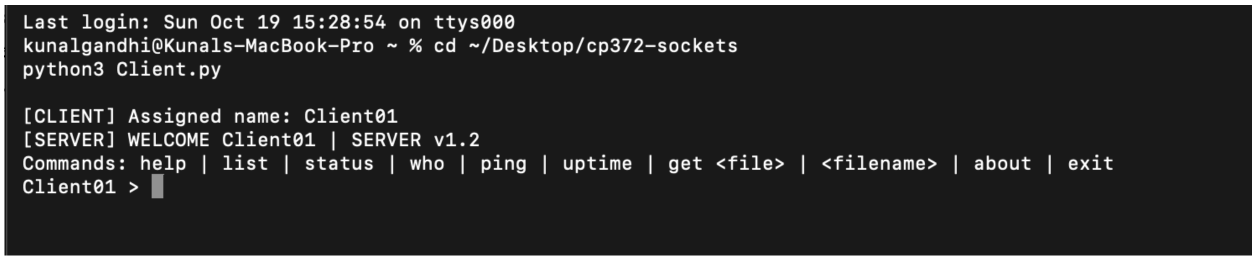


```
CP372 Socket Server | Team: Kunal Gandhi & Muzzi Khan | v1.2
[2025-10-19 15:14:00,324] INFO: Starting server on 127.0.0.1:50000, repo='/Users/kunalgandhi/Desktop/cp372-sockets/server_repo', max_client
s=3
```

Figure 1:

Server launched successfully on localhost (127.0.0.1) and began listening on port 50000 with a limit of three clients, satisfying the rubric requirement for program creation and server initialization.

2. Client Connection and Handshake



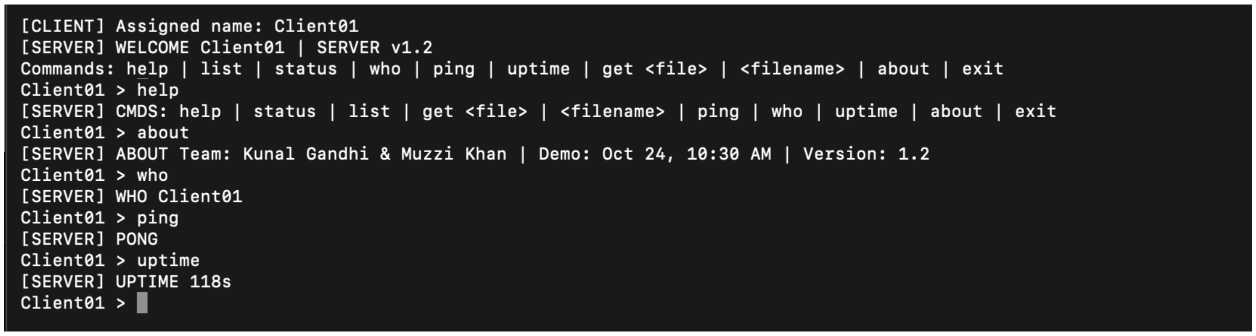
```
Last login: Sun Oct 19 15:28:54 on ttys000
kunalgandhi@Kunals-MacBook-Pro ~ % cd ~/Desktop/cp372-sockets
python3 Client.py

[CLIENT] Assigned name: Client01
[SERVER] WELCOME Client01 | SERVER v1.2
Commands: help | list | status | who | ping | uptime | get <file> | <filename> | about | exit
Client01 >
```

Figure 2:

Client01 connects to the server and receives an assigned name, fulfilling the requirement that each client is automatically named and acknowledged by the server upon connection.

3. Help, About, and Utility Commands

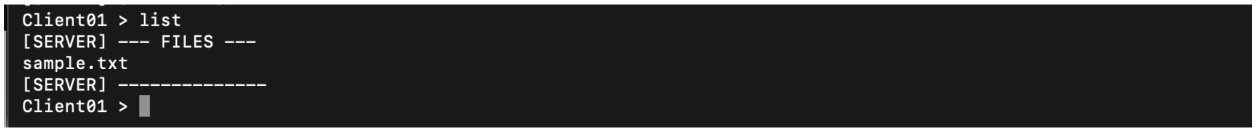


```
[CLIENT] Assigned name: Client01
[SERVER] WELCOME Client01 | SERVER v1.2
Commands: help | list | status | who | ping | uptime | get <file> | <filename> | about | exit
Client01 > help
[SERVER] CMDS: help | status | list | get <file> | <filename> | ping | who | uptime | about | exit
Client01 > about
[SERVER] ABOUT Team: Kunal Gandhi & Muzzi Khan | Demo: Oct 24, 10:30 AM | Version: 1.2
Client01 > who
[SERVER] WHO Client01
Client01 > ping
[SERVER] PONG
Client01 > uptime
[SERVER] UPTIME 118s
Client01 >
```

Figure 3:

Client01 executes several utility commands including help, about, who, ping, and uptime. The server responds correctly to each, demonstrating proper message exchange and responsiveness.

4. Listing Files in Server Repository

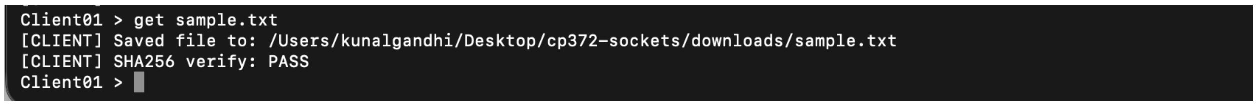


```
Client01 > list
[SERVER] --- FILES ---
sample.txt
[SERVER] --------------
Client01 >
```

Figure 4:

The client issues the “list” command, and the server returns the files stored in its repository. This confirms that the server correctly accesses and displays its local file repository.

5a. File Transfer and Verification

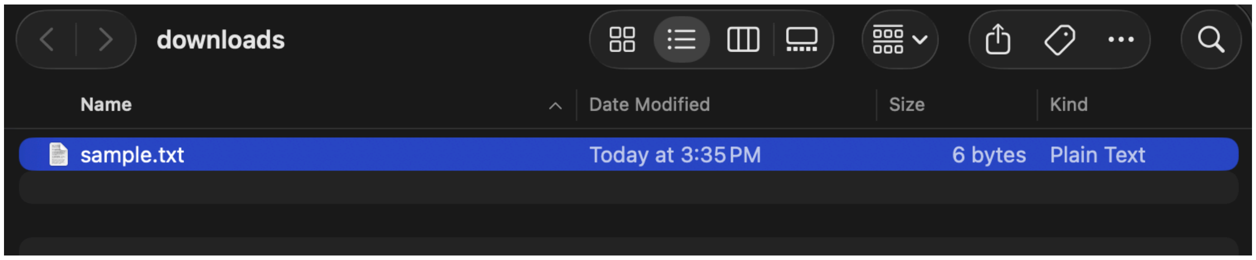


```
Client01 > get sample.txt
[CLIENT] Saved file to: /Users/kunalgandhi/Desktop/cp372-sockets/downloads/sample.txt
[CLIENT] SHA256 verify: PASS
Client01 >
```

Figure 5a:

The client requests sample.txt on the server repository and hashes it through a SHA-256 hash. This indicates that the file transfer and integrity checking functionality is functioning well.

5b. Downloaded File Confirmation

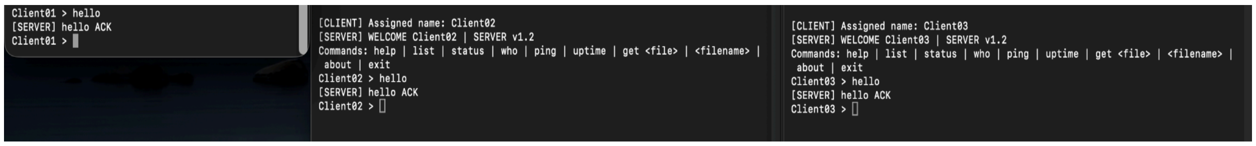


Name	Date Modified	Size	Kind
sample.txt	Today at 3:35 PM	6 bytes	Plain Text

Figure 5b:

The downloaded file sample.txt appears in the client's local "downloads" directory, confirming that the file was received and saved properly.

6. Multi-Client Concurrency



```
Client01 > hello
[SERVER] hello ACK
Client01 >
```

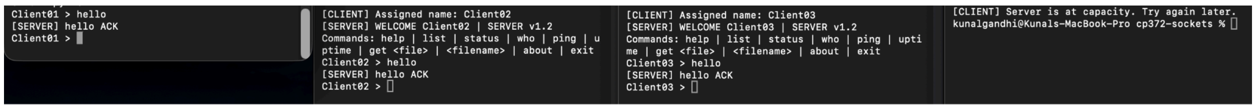
```
[CLIENT] Assigned name: Client02
[SERVER] WELCOME Client02 | SERVER v1.2
Commands: help | list | status | who | ping | uptime | get <file> | <filename> |
 about | exit
Client02 > hello
[SERVER] hello ACK
Client02 >
```

```
[CLIENT] Assigned name: Client03
[SERVER] WELCOME Client03 | SERVER v1.2
Commands: help | list | status | who | ping | uptime | get <file> | <filename> |
 about | exit
Client03 > hello
[SERVER] hello ACK
Client03 >
```

Figure 6:

Three clients connect and communicate simultaneously, each receiving an ACK response from the server. This demonstrates that the server supports concurrent connections using multithreading.

7. Enforced 3-Client Limit



```
Client01 > hello
[SERVER] hello ACK
Client01 >
```

```
[CLIENT] Assigned name: Client02
[SERVER] WELCOME Client02 | SERVER v1.2
Commands: help | list | status | who | ping | u
ptime | get <file> | <filename> | about | exit
Client02 > hello
[SERVER] hello ACK
Client02 >
```

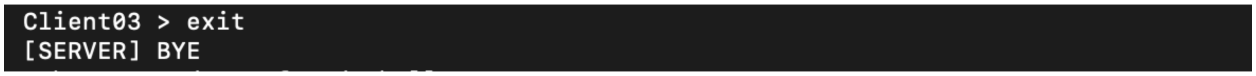
```
[CLIENT] Assigned name: Client03
[SERVER] WELCOME Client03 | SERVER v1.2
Commands: help | list | status | who | ping | upti
me | get <file> | <filename> | about | exit
Client03 > hello
[SERVER] hello ACK
Client03 >
```

```
[CLIENT] Server is at capacity. Try again later.
kunalgandhi@Kunals-MacBook-Pro cp372-sockets %
```

Figure 7:

When a fourth client attempts to connect, the server denies the request and displays “Server is at capacity.”

8a. Client Exit and Clean Disconnection

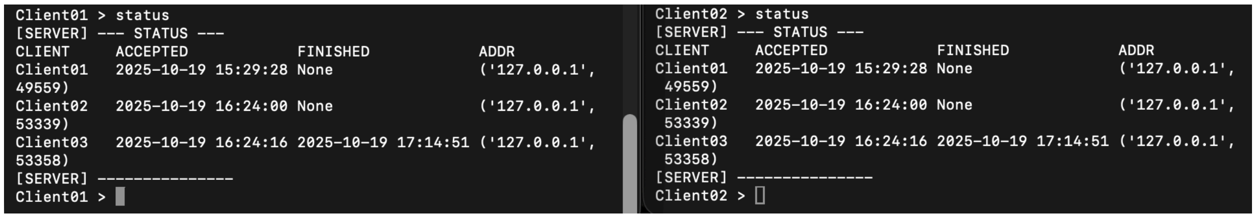


```
Client03 > exit
[SERVER] BYE
```

Figure 8a:

Client03 exits the session, and the server responds with “BYE,” confirming that the client connection is closed properly and resources are released.

8b. Updated Client Status Table



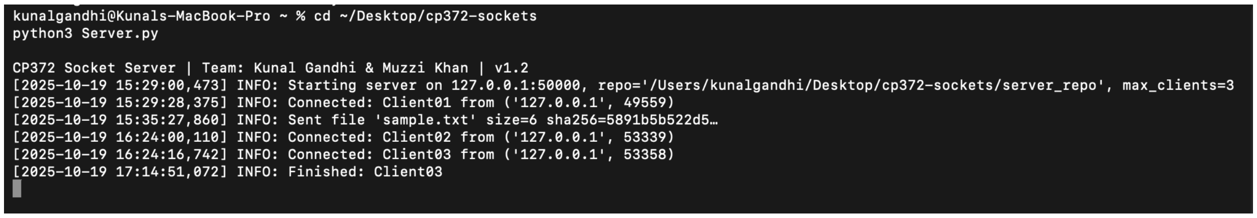
```
Client01 > status
[SERVER] --- STATUS ---
CLIENT     ACCEPTED            FINISHED            ADDR
Client01   2025-10-19 15:29:28 None                ('127.0.0.1',
49559)
Client02   2025-10-19 16:24:00 None                ('127.0.0.1',
53339)
Client03   2025-10-19 16:24:16 2025-10-19 17:14:51 ('127.0.0.1',
53358)
[SERVER] ---------------
Client01 >
```

```
Client02 > status
[SERVER] --- STATUS ---
CLIENT     ACCEPTED            FINISHED            ADDR
Client01   2025-10-19 15:29:28 None                ('127.0.0.1',
 49559)
Client02   2025-10-19 16:24:00 None                ('127.0.0.1',
 53339)
Client03   2025-10-19 16:24:16 2025-10-19 17:14:51 ('127.0.0.1',
 53358)
[SERVER] ---------------
Client02 >
```

Figure 8b:

After Client03 disconnects, the server's status table updates to include the client's finished timestamp, showing that it accurately tracks active and disconnected clients.

9. Server Log Output



```
kunalgandhi@Kunals-MacBook-Pro ~ % cd ~/Desktop/cp372-sockets
python3 Server.py

CP372 Socket Server | Team: Kunal Gandhi & Muzzi Khan | v1.2
[2025-10-19 15:29:00,473] INFO: Starting server on 127.0.0.1:50000, repo='/Users/kunalgandhi/Desktop/cp372-sockets/server_repo', max_clients=3
[2025-10-19 15:29:28,375] INFO: Connected: Client01 from ('127.0.0.1', 49559)
[2025-10-19 15:35:27,860] INFO: Sent file 'sample.txt' size=6 sha256=5891b5b522d5…
[2025-10-19 16:24:00,110] INFO: Connected: Client02 from ('127.0.0.1', 53339)
[2025-10-19 16:24:16,742] INFO: Connected: Client03 from ('127.0.0.1', 53358)
[2025-10-19 17:14:51,072] INFO: Finished: Client03
```

Figure 9:

The server log displays all major events, including startup, client connections, file transfer, and clean disconnections with timestamps. This shows the full operation and sequence of activities handled by the server.

Test Result Summary

All of the program features were tested for functionality correctly, and each one, in fact, came back correctly as expected. The server side managed to handle up to 3 concurrent clients, which also correctly handled all of the message exchanges and the connection limits. The commands, which were used, such as status, list, and get, were executed in a smoothly timed manner, and file transfers were also completed with verified checks through SHA-256. All of the clients were able to disconnect simply and securely, while the server recorded all of the exact timestamps for each session.

Possible Improvements

- For more of a improved performance, we could implement more general optimizations such as streamlining thread management, reducing the repetitions of certain tasks by combining them, and optimizing exception handling.
- By keeping the list of current clients, we could increase message delivery speed and make sure that the server doesn't waste time by looping through the disconnected clients.
- We can make better use of the statistical count of the messages by integrating through the client information object to identify the idle clients or check the communication frequency.

Rubric

#	Requirement	Figure
1	Program can create a Server and client+demo	1
2	Each client created is assigned a name with correct number+demo	2
3	Server can handle multiple clients (multi-threading)+demo	6
4	Server is able to force number of connected clients to 3 clients+demo	7
5	Server and client can exchange messages as described+demo	2-3
6	A client can send "exit" and server cleanly disconnects the client for new clients	8a
7	Server maintains clients' connections details+demo	8b
8	Server is able to list files in its repository+demo	4
9	Server streams files of choice to clients+demo	5a-5b