CS202

Computer Networks Tutorial-1

Birudugadda Srivibhav 22110050

February 7, 2025

[GitHub link for code]

Problem 1: Client-Server Task Processing Program

Overview

This program consists of a client and three different server designs that process tasks sent by the client.

Client Side Program

a) IP Address Retrieval

This function uses the ***netifaces*** library to obtain the IP address of a specified network interface. It's crucial for dynamically determining the server's IP address.

```
def get_ip_address(interface):
    return ni.ifaddresses(interface)[ni.AF_INET][0]['addr']
```

b) Socket Creation and Connection

This code creates a TCP socket and establishes a connection to the server using the specified host and port.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((server_host, server_port))
```

c) User Interface Loop

This loop presents a menu to the user and captures their choice, forming the main interaction point of the client application.

```
while True:
    print("\n1. Change case\n2. Evaluate expression\n3. Reverse string\n4. Exit")
    choice = input("Enter choice (1-4): ")

    if choice == '4':
        break
```

d) Request Formatting and Sending

When a valid choice is made, the client formats the request by combining the choice and user input, then sends it to the server.

```
if choice in ('1', '2', '3'):
    data = input("Enter input: ")
    message = f"{choice}|{data}"
    s.sendall(message.encode())
```

e) Response Handling

After sending a request, the client waits for and receives the server's response, then displays it to the user.

```
response = s.recv(1024).decode()
print("Result:", response)
```

Server Side Program

a) Network Interface Configuration

The ***get_ip_address()*** function uses the ***netifaces*** library to dynamically retrieve the server's IP address from a specified network interface (eth0). This ensures the server binds to the correct interface without hardcoding IP addresses.

```
def get_ip_address(interface):
    return ni.ifaddresses(interface)[ni.AF_INET][0]['addr']
```

b) Request Processing Logic

```
def process_request(data):
    try:
        choice, payload = data.split('|', 1)
        if choice == '1':
            return payload.swapcase()
        elif choice == '2':
            return str(eval(payload))
        elif choice == '3':
            return payload[::-1]
        return "Invalid choice"
    except Exception as e:
        return f"Error: {str(e)}"
```

The ***process_request(data)*** function:

1. Splits incoming data into a choice (operation type) and payload (input data).
2. Executes one of three operations:
    - Case Conversion: Uses swapcase() to toggle letter cases.
    - Math Evaluation: Safely evaluates expressions using Python's eval().
    - String Reversal: Reverses strings via slicing ([::-1]).
3. Returns error messages for invalid requests or exceptions.

c) Client Connection Handler

The ***handle_client()*** function manages the entire lifecycle of a client connection:

- Logs connection timestamps and client addresses.
- Continuously receives requests (up to 1024 bytes) and sends back processed responses.
- Prints detailed logs for received requests and sent responses.
- Ensures connections are closed gracefully using a try-finally block.

```
def handle_client(conn, addr, server_type, worker_info=""):
    try:
        with conn:
            current_time = datetime.now().strftime("%H:%M:%S")
            print(f"\n[{current_time}] {server_type} - Connection from {addr}")
            print(f"{worker_info} - Handling client {addr}")

            while True:
                data = conn.recv(1024).decode()
                if not data:
                    break

                print(f"\n{worker_info} - Received request: {data}")
                response = process_request(data)
                print(f"{worker_info} - Sending response: {response}")
                conn.sendall(response.encode())

    finally:
        current_time = datetime.now().strftime("%H:%M:%S")
        print(f"\n[{current_time}] {worker_info} - Client {addr} disconnected")
```

d) Server Architectures

Single-Process Server

The single_process_server function implements a basic sequential server that handles one client at a time. It binds to a given network interface and port, listens for incoming connections, and processes client requests sequentially. When a client connects, the server accepts the connection and passes it to the handle_client function for processing. After a client connection is closed, the server waits for the next

client. Since it operates in a single process, it cannot handle multiple clients concurrently, making it suitable for simple applications but inefficient for high-traffic scenarios.

```python
# 1. Single-Process Server (Sequential)
def single_process_server(interface='eth0', port=65432):
    host = get_ip_address(interface)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        print(f"\nSingle-Process Server (PID: {os.getpid()})")
        print(f"Listening on {host}:{port}")
        print("-- Only handles one client at a time --\n")

        while True:
            print("[MAIN] Waiting for connections...")
            conn, addr = s.accept()
            print(f"New connection from {addr}")
            handle_client(conn, addr, "Single-Process", "[MAIN PROCESS]")
            print(f"Connection from {addr} closed. Ready for next client.")
```

Multi-Process Server

```python
# 2. Multi-Process Server
def multi_process_server(interface='eth0', port=65432):
    host = get_ip_address(interface)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        print(f"\nMulti-Process Server (Main PID: {os.getpid()})")
        print(f"Listening on {host}:{port}")
        print("-- New process for each client --\n")

        while True:
            conn, addr = s.accept()
            child_pid = os.fork()

            if child_pid == 0:  # Child process
                s.close()
                worker_info = f"[CHILD PID: {os.getpid()}]"
                handle_client(conn, addr, "Multi-Process", worker_info)
                os._exit(0)
            else:  # Parent process
                conn.close()
                print(f"\n[MAIN] Forked child process {child_pid} for {addr}")
```

The multi-process server creates a new OS process for each client using os.fork(), allowing multiple clients to be handled concurrently. The parent process listens for connections and forks a child process for each client, closing its copy of the client socket after forking. Each child process independently handles client requests in an isolated memory space, ensuring efficient parallel processing. The server also tracks process IDs (PIDs) for debugging, making it easier to monitor active child processes (e.g., [CHILD PID: 1234]).

Multi-Threaded Server

The multi_threaded_server function implements a multi-threaded TCP server that spawns a new thread for each client connection. It binds to a specified network interface and port, listens for incoming connections, and accepts clients in a loop. When a client connects, the server creates a new thread to handle the request using handle_client, passing the connection details and thread name for debugging. This approach allows multiple clients to be served concurrently while sharing memory space, making it more efficient than a single-process server for handling multiple requests simultaneously.

```python
# 3. Multi-Threaded Server
def multi_threaded_server(interface='eth0', port=65432):
    host = get_ip_address(interface)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        print(f"\nMulti-Threaded Server (Main PID: {os.getpid()})")
        print(f"Listening on {host}:{port}")
        print("-- New thread for each client --\n")

        while True:
            conn, addr = s.accept()
            thread = threading.Thread(target=handle_client, args=(
                conn,
                addr,
                "Multi-Threaded",
                f"[THREAD: {threading.current_thread().name}]"
            ))
            thread.start()
            print(f"\n[MAIN] Started thread {thread.name} for {addr}")
```

Working of Programs

To demonstrate the execution of both the programs, I am using **2 Ubuntu WSL Machines (Ubuntu 24.04 as the server and Ubuntu 20.04 as the client).** The server-side code can be executed using the command ***python server_side.py***

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side.py
Enter server type (1: Single-Process, 2: Multi-Process, 3: Multi-Threaded):
```

Upon execution of the above command, the program asks whether to run the server in Single-Process mode, Multi-Process mode, or Multi-Threaded mode.

a) Single-Process Server

Type 1 in the command line to choose the single-process server option.

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side.py
Enter server type (1: Single-Process, 2: Multi-Process, 3: Multi-Threaded): 1

Single-Process Server (PID: 19238)
Listening on 172.23.198.251:65432
-- Only handles one client at a time --

[MAIN] Waiting for connections...
```

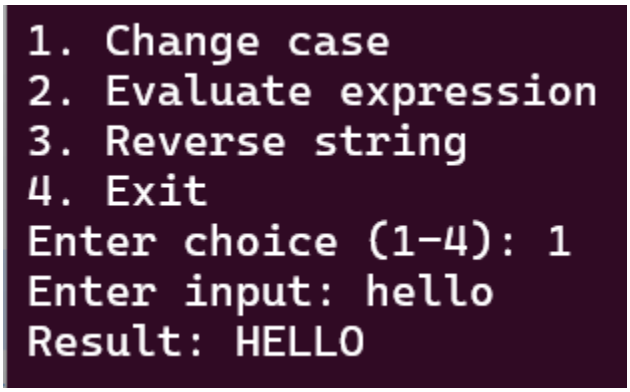
So now the server is waiting for a client to connect. We run the ***python client_side.py*** command on a second wsl machine to activate a client.

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$ python client_side.py
Connected to server at 172.23.198.251:65432

1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4):
```

As soon as ***python client_side.py*** command is executed on the client side, we can see that the server detects the connection:

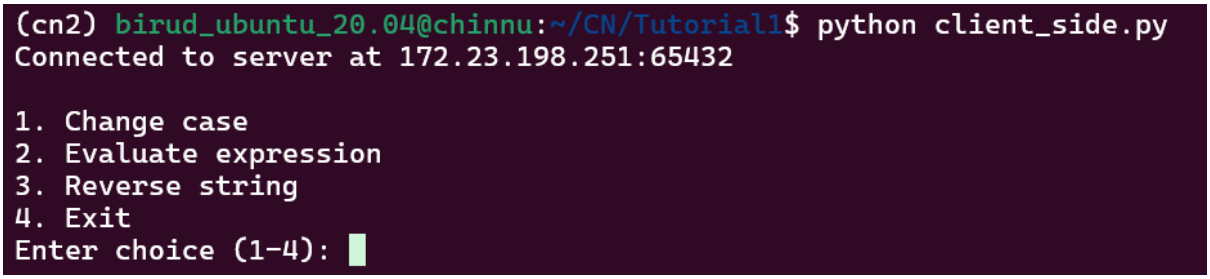
```
[MAIN] Waiting for connections...
New connection from ('172.23.198.251', 34416)

[12:26:25] Single-Process - Connection from ('172.23.198.251', 34416)
[MAIN PROCESS] - Handling client ('172.23.198.251', 34416)
```

Now let's make a request to change the case of the input string via the client machine.



We can also see that the same is updated near the server side too:

The same process repeats even when the client requests for evaluating an expression or reversing a string. Now let's try to connect a second client to the server, ideally the connection should fail since the server is already busy with our first client.



The above is the second client, we can see that the second client program has started but on the server side, the second client is not detected:

Even if the second client makes any request, it won't be processed:

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$ python client_side.py
Connected to server at 172.23.198.251:65432

1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 1
Enter input: world
```

We can see that the second client's program has been stuck there. But as soon as I kill the first client's program, the second client's request is processed.

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$ python client_side.py
Connected to server at 172.23.198.251:65432

1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 1
Enter input: hello
Result: HELLO

1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 4
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$
```

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$ python client_side.py
Connected to server at 172.23.198.251:65432

1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 1
Enter input: world
Result: WORLD
```

The same is also reflected on the server side.

```
[12:35:54] [MAIN PROCESS] - Client ('172.23.198.251', 34416) disconnected
Connection from ('172.23.198.251', 34416) closed. Ready for next client.
[MAIN] Waiting for connections...
New connection from ('172.23.198.251', 58990)

[12:35:54] Single-Process - Connection from ('172.23.198.251', 58990)
[MAIN PROCESS] - Handling client ('172.23.198.251', 58990)

[MAIN PROCESS] - Received request: 1|world
[MAIN PROCESS] - Sending response: WORLD
```

Hence, In the single-process mode, the server can only handle one client at a time.

b) Multi-Process Server

Let's rerun the server side program in multi-process mode:

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side.py
Enter server type (1: Single-Process, 2: Multi-Process, 3: Multi-Threaded): 2

Multi-Process Server (Main PID: 22677)
Listening on 172.23.198.251:65432
-- New process for each client --
```

In this mode, multiple clients can be connected simultaneously to the server. For instance, I ran the client side code twice and the below is the output on the server side:

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side.py
Enter server type (1: Single-Process, 2: Multi-Process, 3: Multi-Threaded): 2

Multi-Process Server (Main PID: 22677)
Listening on 172.23.198.251:65432
-- New process for each client --


[MAIN] Forked child process 22864 for ('172.23.198.251', 38792)

[12:40:33] Multi-Process - Connection from ('172.23.198.251', 38792)
[CHILD PID: 22864] - Handling client ('172.23.198.251', 38792)

[MAIN] Forked child process 22907 for ('172.23.198.251', 36084)

[12:40:48] Multi-Process - Connection from ('172.23.198.251', 36084)
[CHILD PID: 22907] - Handling client ('172.23.198.251', 36084)
```

We can clearly see that for each client, a new process is created using **os.fork()**

Let's make requests from both the client machines and see what happens on server side:

```
1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 1
Enter input: hello
Result: HELLO
```

```
1. Change case
2. Evaluate expression
3. Reverse string
4. Exit
Enter choice (1-4): 2
Enter input: 1+2+3
Result: 6
```

Server side:

```
[MAIN] Forked child process 22864 for ('172.23.198.251', 38792)

[12:40:33] Multi-Process - Connection from ('172.23.198.251', 38792)
[CHILD PID: 22864] - Handling client ('172.23.198.251', 38792)

[MAIN] Forked child process 22907 for ('172.23.198.251', 36084)

[12:40:48] Multi-Process - Connection from ('172.23.198.251', 36084)
[CHILD PID: 22907] - Handling client ('172.23.198.251', 36084)

[CHILD PID: 22864] - Received request: 1|hello
[CHILD PID: 22864] - Sending response: HELLO

[CHILD PID: 22907] - Received request: 2|1+2+3
[CHILD PID: 22907] - Sending response: 6
```

c) Multi-Threaded Server

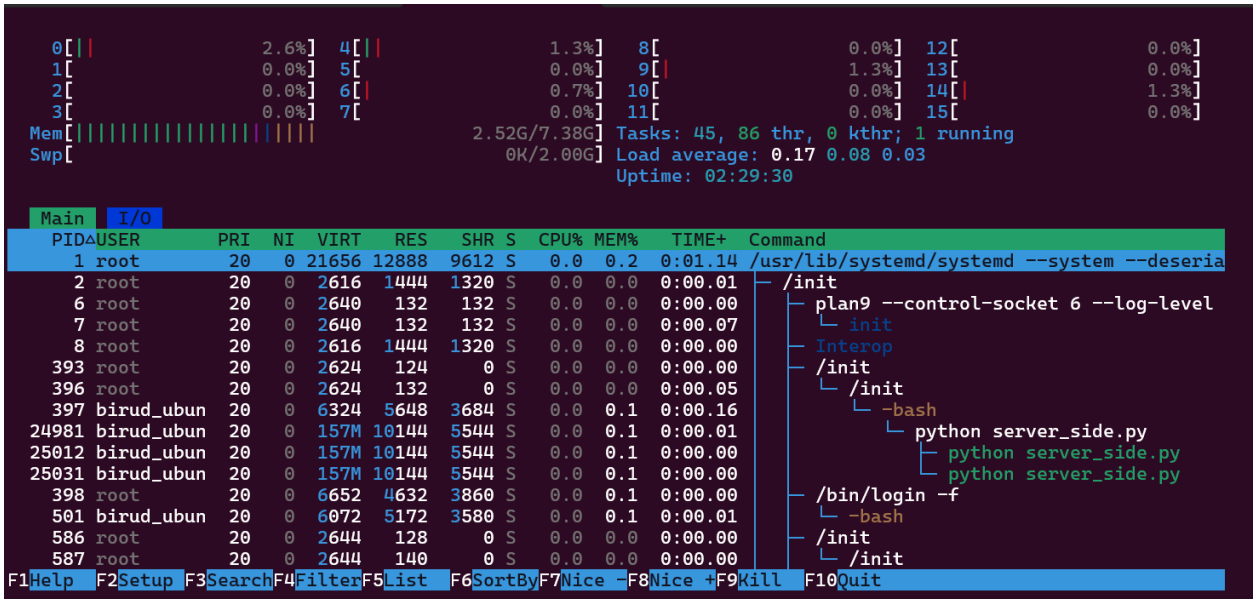
Now let's run the server in multi-threaded mode:

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side.py
Enter server type (1: Single-Process, 2: Multi-Process, 3: Multi-Threaded): 3

Multi-Threaded Server (Main PID: 24981)
Listening on 172.23.198.251:65432
-- New thread for each client --
```

Now when two clients are connected, we can see that two separate threads are created using **threading.Thread()**

```
[12:52:40] Multi-Threaded - Connection from ('172.23.198.251', 35598)

[MAIN] Started thread Thread-1 (handle_client) for ('172.23.198.251', 35598)
[THREAD: MainThread] - Handling client ('172.23.198.251', 35598)

[12:52:47] Multi-Threaded - Connection from ('172.23.198.251', 38152)
[THREAD: MainThread] - Handling client ('172.23.198.251', 38152)

[MAIN] Started thread Thread-2 (handle_client) for ('172.23.198.251', 38152)
```

We can also verify that threads are created using ***htop*** command in linux.



Problem 2: Client-Server Benchmarking Task

Overview

This task involves implementing a client-server architecture where the client sends a string to the server, and the server responds with the reversed string after a 3-second delay. The server will be implemented in three different ways, and we will benchmark its performance by running multiple concurrent clients and measuring execution time.

Client Side Program

For this question, I have modified the ***client_side.py*** to send a test string ("BenchmarkTestString123") and wait for a response from the server.

The below is the code for the client machine.

```python
import socket
import netifaces as ni
import sys

def get_ip_address(interface):
    return ni.ifaddresses(interface)[ni.AF_INET][0]['addr']

def run_client(server_interface='eth0', server_port=65432):
    try:
        # Get server's IP from its network interface
        server_host = get_ip_address(server_interface)

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((server_host, server_port))
            message = "BenchmarkTestString123"
            s.sendall(message.encode())
            response = s.recv(1024).decode()
            print(f"Received: {response}")

    except Exception as e:
        print(f"Connection error: {str(e)}")
        sys.exit(1)

if __name__ == "__main__":
    run_client(server_interface='eth0')
```

Server Side Program

Just like in the previous question, I have written three functions to handle Single-Process, Multi-Process, and Multi-Thread servers, respectively.

Upon executing the ***server_side.py*** code, we are again given three options, one for each type of server to choose from.

```python
# Single-Process Server
def single_process_server(interface='eth0', port=65432):
    host = get_ip_address(interface)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        while True:
            conn, addr = s.accept()
            handle_client(conn, addr, "Single-Process")

# Multi-Process Server
def multi_process_server(interface='eth0', port=65432):
    host = get_ip_address(interface)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        while True:
            conn, addr = s.accept()
            pid = os.fork()
            if pid == 0:
                s.close()
                handle_client(conn, addr, "Multi-Process")
                os._exit(0)
            else:
                conn.close()

# Multi-Threaded Server
def multi_threaded_server(interface='eth0', port=65432):
    host = get_ip_address(interface)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        while True:
            conn, addr = s.accept()
            thread = threading.Thread(target=handle_client, args=(conn, addr, "Multi-Threaded"))
            thread.start()
```

Benchmark Tests

We will be executing the ***start.sh*** given to us with varying clients and plot the execution times for each of the servers.

An example of running ***start.sh*** with 10 clients on a Single-Process server.

Client side:

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Tutorial1$ bash start.sh client_side2.py 10
Running 'client_side2.py' 10 times concurrently...
Starting instance #1...
Starting instance #2...
Starting instance #3...
Starting instance #4...
Starting instance #5...
Starting instance #6...
Starting instance #7...
Starting instance #8...
Starting instance #9...
Starting instance #10...
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Received: 321gnirtStseTkramhcneB
Total Execution Time: 30.062745717 seconds
Average Execution Time per script: 3.006274571 seconds
```

Server side:

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Tutorial1$ python server_side2.py
Enter server type (1/2/3): 1
 - Completed ('172.23.198.251', 60200)
 - Completed ('172.23.198.251', 60218)
 - Completed ('172.23.198.251', 60212)
 - Completed ('172.23.198.251', 60204)
 - Completed ('172.23.198.251', 60202)
 - Completed ('172.23.198.251', 60210)
 - Completed ('172.23.198.251', 60214)
 - Completed ('172.23.198.251', 60216)
 - Completed ('172.23.198.251', 60206)
 - Completed ('172.23.198.251', 60208)
```

We now continue testing with 10, 20, 30, … 100 clients for all three servers. Below is the output showing the number of concurrent clients and the execution time (latency) for each server.

Single-Process Server

Clients	Execution Time (s)
10	30.023681
20	60.036497
30	90.053178
40	120.071838
50	150.070198
60	180.100327
70	210.093805
80	240.123101
90	270.166767
100	300.163465

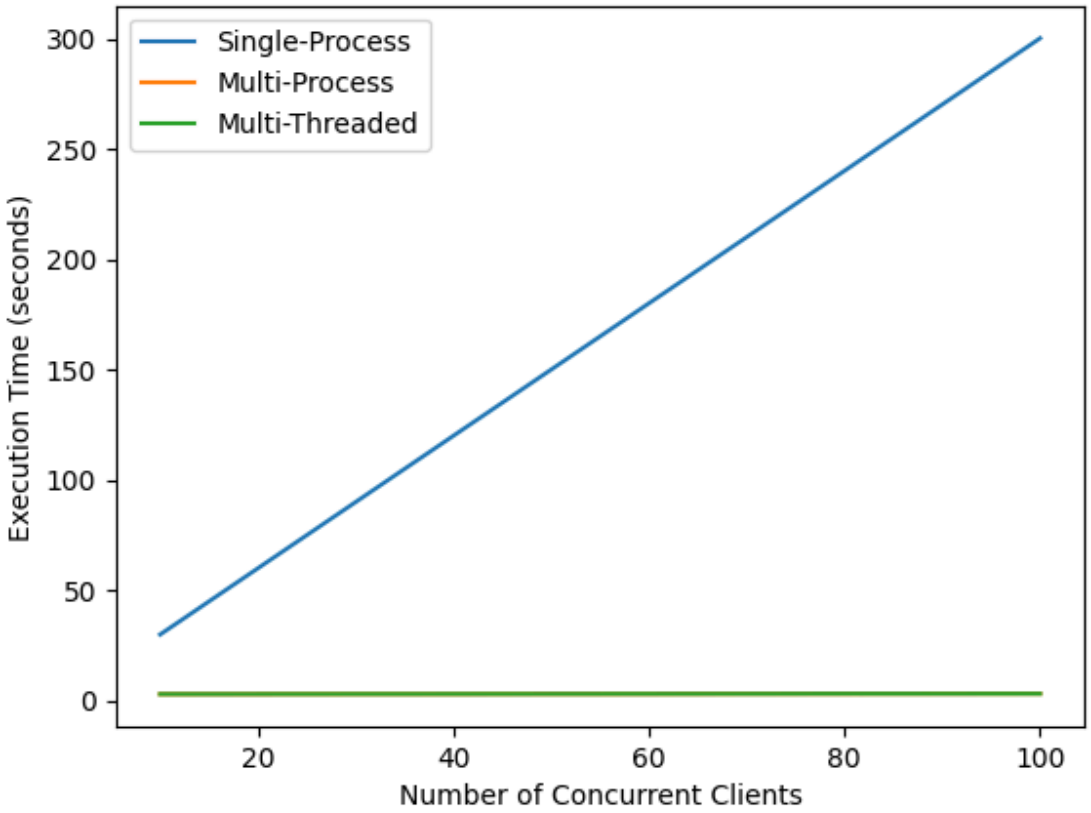
Multi-Process Server

Clients	Execution Time (s)
10	3.027398
20	3.055609
30	3.078539
40	3.078338
50	3.121362
60	3.106676
70	3.190848
80	3.164202
90	3.220507
100	3.229689

Multi-Threaded Server

Clients	Execution Time (s)
10	3.032718
20	3.044437
30	3.067425
40	3.088482
50	3.106144
60	3.142782
70	3.148112
80	3.212299
90	3.189932
100	3.202745

Now below is the plot showing the number of concurrent clients v/s execution time (latency) for each server.



We can clearly see that the Multi-Process and Multi-Threaded servers are much more efficient and fast at handling more concurrent clients than the Single-Process Server.