INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

Tutorial Report
Computer Networks

Submitted By

Bhavik Patel - 22110047

Github Repository Link

Problem 1

Client-Server Task Processing Program Using Socket Programming

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

Client Program

- The client presents a menu with four options:
  1. Change the case of a string (convert uppercase to lowercase and vice versa).
  2. Evaluate a mathematical expression (e.g., "2+3*5/2").
  3. Find the reverse of a given string.
  4. Exit the program.
- The client sends the selected task to the server and waits for the response.
- The menu repeats until the client chooses to exit or presses Ctrl+C.

Server Implementations

The server will be implemented in three different ways:

1) Single-Process Server
   - The server can only handle one client at a time.
   - If a client is connected, other clients must wait until the current client disconnects.
2) Multi-Process Server
   - The server creates a new process for each connected client.
   - Multiple clients can be handled at the same time, each running in a separate process.
3) Multi-Threaded Server
   - The server creates a new thread for each connected client.
   - Multiple clients can be handled simultaneously but with shared memory resources.

Tools and Technologies

- Programming Language: Python 3
- Socket Programming: Python's built-in socket module
- Concurrency:
  - Multi-Process Server: Utilizes the os.fork() method
  - Multi-Threaded Server: Utilizes the threading module

Methodology and Steps

Client Implementation:

1. Set up a TCP socket connection to the server.
2. Display a user-friendly menu.
3. Read the user's selection and input data.
4. Send the formatted request to the server and wait for a response.
5. Display the server's response.

Single-Process Server Implementation:

1. Create a TCP server socket and bind it to a port.
2. Listen for a client connection.

3. Accept the connection, receive the request, process the request (change case, evaluate an expression, or reverse string), and send the response back and wail for the next request.
4. If the client closes the connection, then wait for the next client.

Multi-Process Server Implementation:

1. Create and bind the server socket.
2. Upon accepting a connection, use `os.fork()` to create a new process.
3. In the child process, handle the client's request and then exit if the connection is closed.
4. The parent process continues accepting new connections.
5. The parent process doesn't serve any client. It just accepts a new client connection and gives it to a child process.

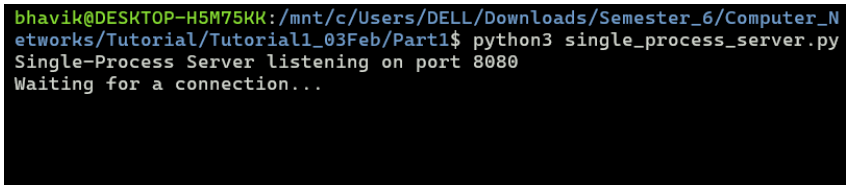
Multi-Threaded Server Implementation:

1. Create and bind the server socket.
2. Upon accepting a connection, start a new thread using Python's `threading.Thread` to handle the client's request.
3. Threads run concurrently, allowing multiple clients to be served simultaneously.

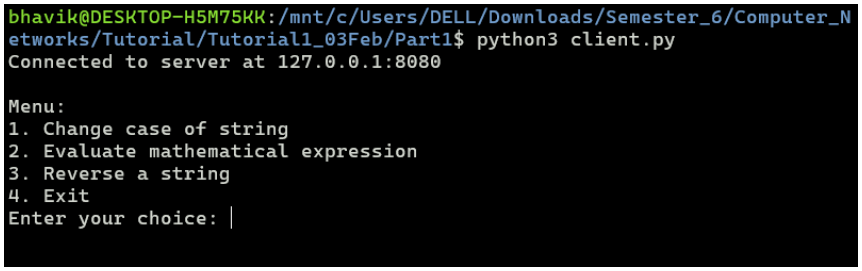
Code Implementation

The complete Python source code for the client and the three types of servers is provided below.
(Refer to the attached code files: `client.py`, `single_process_server.py`, `multi_process_server.py`, and `multi_threaded_server.py`.)

Some screenshots of Client and Server.
Single-Process Server running.



The client is running and connected to the server hosted on localhost.



The client-side output of some request-response

```
bhavik@DESKTOP-H5M75KK:/mnt/c/Users/DELL/Downloads/Semester_6/Computer_N
etworks/Tutorial/Tutorial1_03Feb/Part1$ python3 client.py
Connected to server at 127.0.0.1:8080

Menu:
1. Change case of string
2. Evaluate mathematical expression
3. Reverse a string
4. Exit
Enter your choice: 1
Enter input: bhavik
Server Response: BHAVIK

Menu:
1. Change case of string
2. Evaluate mathematical expression
3. Reverse a string
4. Exit
Enter your choice: 2
Enter input: 2*9
Server Response: 18

Menu:
1. Change case of string
2. Evaluate mathematical expression
3. Reverse a string
4. Exit
Enter your choice: 3
Enter input: bhavik
Server Response: kivahb

Menu:
1. Change case of string
2. Evaluate mathematical expression
3. Reverse a string
4. Exit
Enter your choice: 4
Goodbye! received. Closing connection
```

The corresponding logs on the server side of the same requests are shown in the above screenshot.

```
bhavik@DESKTOP-H5M75KK:/mnt/c/Users/DELL/Downloads/Semester_6/Computer_N
etworks/Tutorial/Tutorial1_03Feb/Part1$ python3 single_process_server.py

Single-Process Server listening on port 8080
Waiting for a connection...
Connection from ('127.0.0.1', 48118)
Request: 1:bhavik
Request: 2:2*9
Request: 3:bhavik
Request: 4:exit
Waiting for a connection...
```

Problem 2

Client-Server Benchmarking Task

We use same programs that we wrote in problem with some updates

Updated Client and Server Behavior

Client Program:

The client (updated from Problem 1) now performs a single task: connecting to the server, sending a string, waiting for the server's response (the reversed string), and then closing the connection.

Server Implementations:

Each server variant was modified to introduce a fixed 3-second delay before reversing and returning the string.

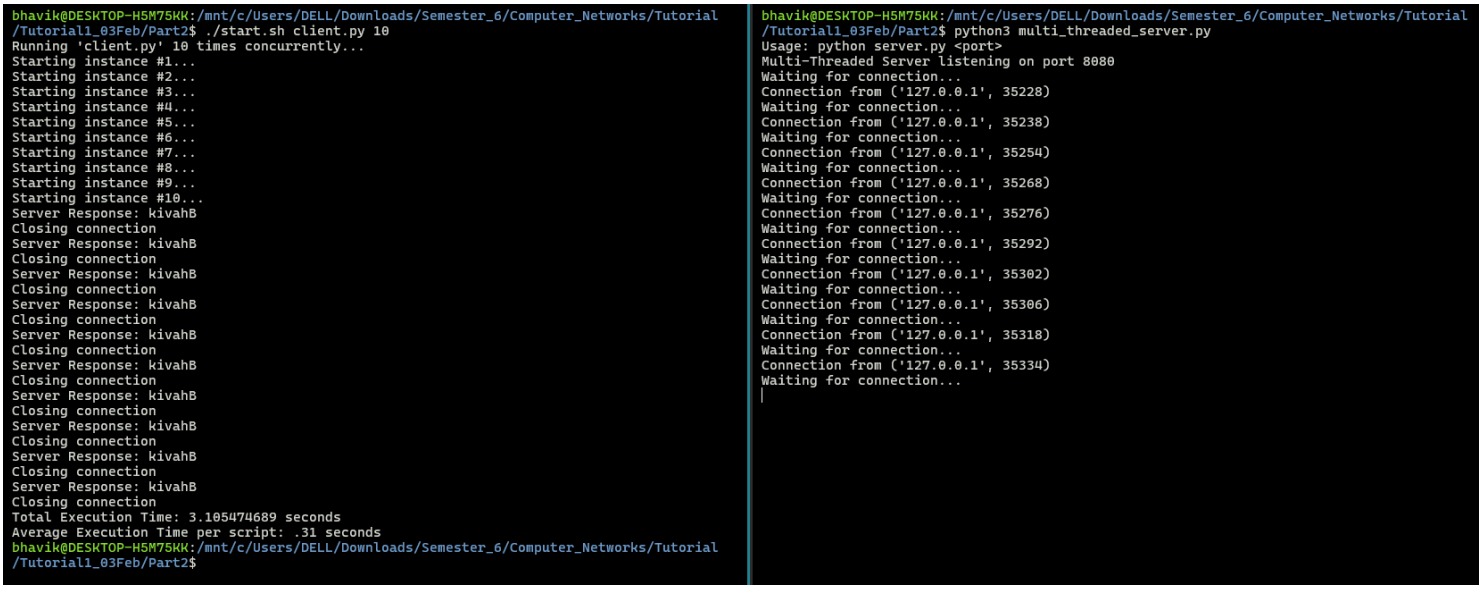
Benchmarking Scripts

To measure performance under varying loads, two approaches were developed:

`start.sh` **Script:** This script is used to launch a specified client program a given number of times in parallel. For example:

`bash start.sh client.py 5`

This command runs 5 concurrent instances of the client program. You can use this script to manually test each server implementation with different client counts (e.g., 10, 20, …, 100), then record the execution times manually.



1. First, start the server.
2. Then run the `start.sh` script and store the execution time in some text file.
3. Repeat the above steps for different numbers of concurrent clients and for each server.
4. At last run `plot_results.py` script to generate the plot.

`finalbenchmark.sh` Script:

To automate the benchmarking process completely, I have written a comprehensive script named `finalbenchmark.sh`. This script iterates over all three server implementations and runs the tests with different numbers of concurrent clients (e.g., from 10 up to 100). It captures and stores the execution times in separate text files for each server type and generates plots from these text files to visualize performance (latency vs. number of

concurrent clients). The script will help to generate all benchmark results automatically instead of using start.sh where we need to run the script multiple times manually.

```
bhavik@DESKTOP-H5M75KK:/mnt/c/Users/DELL/Downloads/Semester_6/Computer_Networks/Tutorial/Tutorial1_03
Feb/Part2$ ./finalBenchmark.sh
Starting server: single_process_server.py...
Single-Process Server listening on port 8020
Waiting for connection...
Running 10 concurrent clients...
Connection from ('127.0.0.1', 53518)
Waiting for connection...
Server Response: kivahB
Closing connection
Connection from ('127.0.0.1', 53528)
Waiting for connection...
Server Response: kivahB
Closing connection
Connection from ('127.0.0.1', 53536)
Waiting for connection...
Server Response: kivahB
Closing connection
Connection from ('127.0.0.1', 53542)
Waiting for connection...
Connection from ('127.0.0.1', 53544)
Server Response: kivahB
Closing connection
Waiting for connection...
Connection from ('127.0.0.1', 53554)
Server Response: kivahB
Closing connection
Waiting for connection...
Connection from ('127.0.0.1', 53556)
Server Response: kivahB
Closing connection
Waiting for connection...
Connection from ('127.0.0.1', 53562)
Server Response: kivahB
Closing connection
```

Results and Observations

Single-Process Server: Latency increases linearly with the number of clients since each client must wait its turn through the 3-second delay.

Multi-Process Server: The use of separate processes improves performance by handling multiple clients concurrently. However, the overhead associated with process creation is noticeable as the number of concurrent clients grows.

Multi-Threaded Server: Exhibiting the lowest overall latency, the thread-based approach scales best in this benchmark. Thread creation overhead is minimal compared to processes, leading to more efficient concurrency for the given workload.

NOTE: *The below result is generated after running the benchmark 3 times and then taking the average of the output generated in each of the benchmarks.*

The results are stored inside a results folder in the main directory.

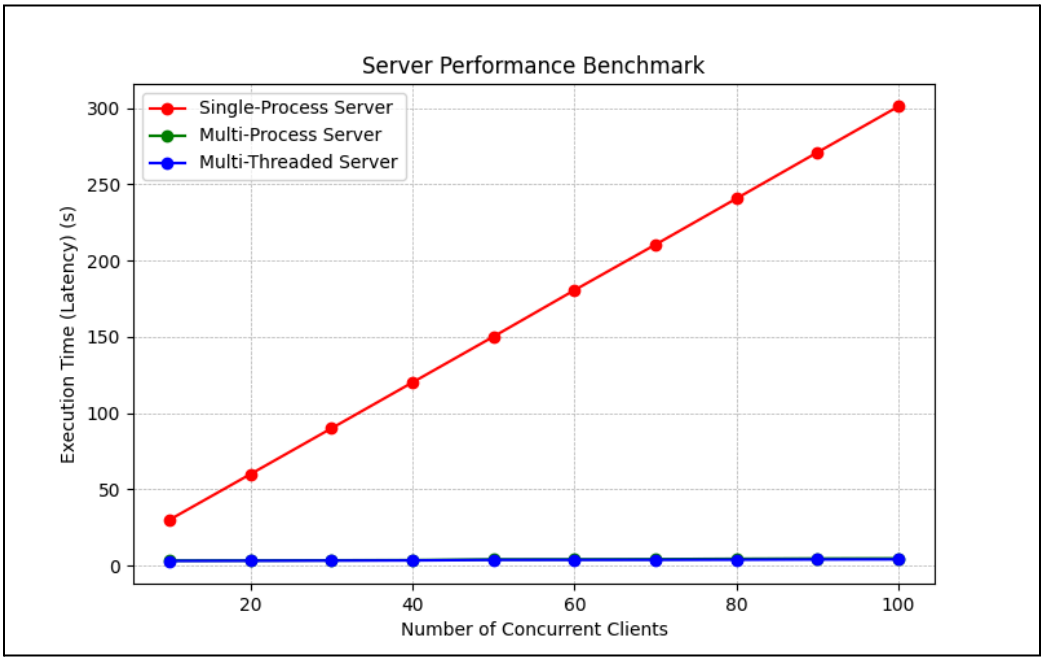


Fig: Plot of Execution Time(sec) Vs Concurrent Users

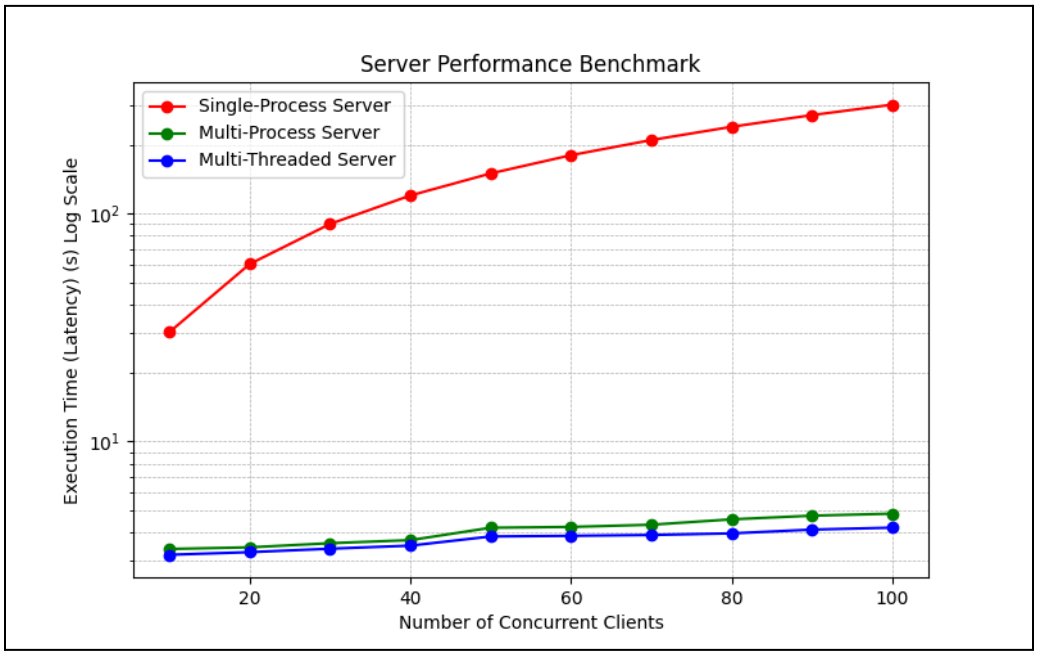


Fig: Plot of Execution Time(**Log Scale**) Vs Concurrent Users

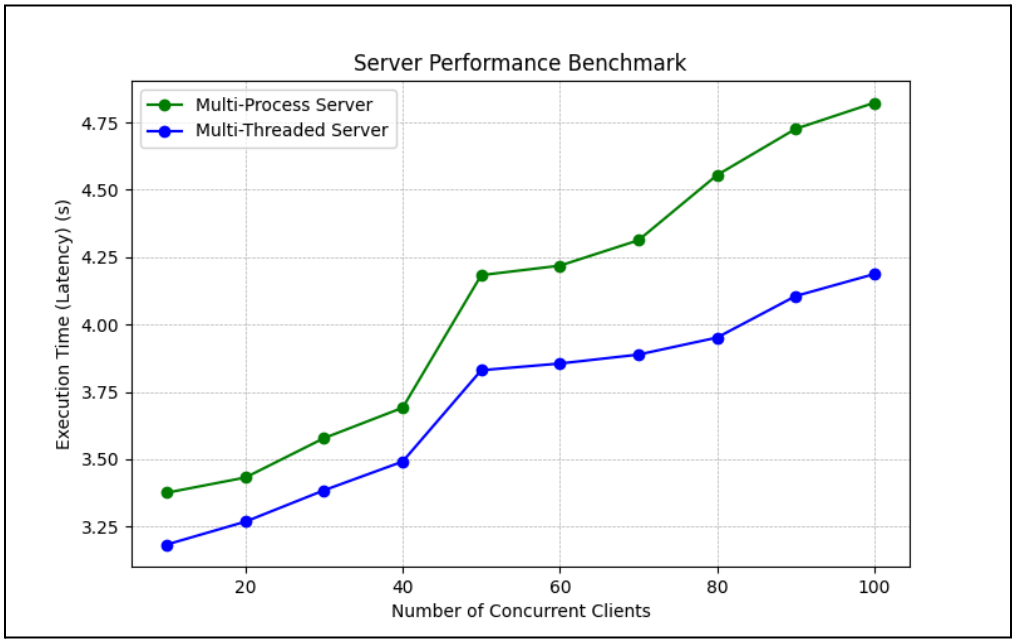


Fig: Multi-Process Vs Multi_THreaded Server

Conclusion

The benchmarking exercise clearly confirms that while all three server implementations function correctly, their scalability under concurrent load varies considerably:

- Single-Process Server is simple but not suitable for high concurrency.
- Multi-Process Server offers improved parallelism at the cost of increased overhead due to new process creation.
- Multi-Threaded Server provides the best scalability for this particular task.