Analysis

Our testing revealed consistent performance trends across all six operations. When we increased the number of client threads from 1 up to 50, we saw a proportional rise in total turnaround time. This made sense because the iterative server is single-threaded and handles one request at a time.

Interestingly, the **average turnaround time per client** remained very steady hovering around 0.52 to 0.53 seconds. This tells us that each individual request, once being served, took the same amount of time regardless of how many other clients were in line. It wasn't faster or slower based on the load; it just waited its turn.

The root cause of this behavior lies in the **server's iterative architecture**. Since it only handles one client at a time, incoming client threads are essentially placed in a queue. The system executes one request, sends back the result, closes that connection, and then starts handling the next one. There's no parallelism on the server side, so while it's reliable, it naturally becomes a bottleneck when client volume rises.

If this were a production-grade system or needed to scale up, switching to a multithreaded or asynchronous server model would be necessary to reduce client wait times.

Configuration

The client program was built to be user-friendly while allowing robust testing capabilities. Upon execution, it prompts the user to enter the IP address and port number of the server they wish to connect to. Then, the user selects one of the six supported operations (such as getting server time or listing users) and specifies how many requests to send. Once the parameters are set, the client spins up multiple threads—each one acting as an independent requester that connects to the server, sends the operation code, and receives the response. This multithreaded approach is essential to simulate multiple clients accessing the server at once. Each thread prints the result it gets back, providing a real-time view of the server's responsiveness. We used Python's `threading` module to implement this logic. The modular design allowed easy tweaking for different test scenarios, and it worked smoothly with the iterative server despite its single-threaded limitations.

Conclusion

In conclusion, the iterative server handled concurrent client requests with reliability and stability, even under increasing client loads. Each operation executed in a predictable amount of time, and the average per-client turnaround time was remarkably consistent. This confirms that our implementation is sound for its intended scope: a basic, serially processing socket server.

However, the testing also highlighted a key limitation—the lack of scalability. As the number of concurrent clients increased, the total time to complete all requests rose linearly. This would not be acceptable in a real-time or high-volume environment.

That said, the project effectively met all of its goals, and the client-server system performed exactly as expected for an iterative setup. The data analysis provides strong evidence supporting our architectural assumptions, and the assignment offered valuable insight into performance testing and socket-based network design.

Lessons Learned

1. **Socket Communication Basics**: This was our first time writing full socket-based programs, and we learned how TCP connections are established, managed, and torn down between clients and servers.
2. **Threading in Python**: We learned how to use Python's `threading` module to create multiple client threads. This was especially helpful in simulating a realistic load against the server.
3. **Executing Linux Commands**: Having the server execute shell commands and return their outputs helped us understand how to bridge system-level operations with networking code.
4. **Error Handling**: We encountered occasional errors like broken pipes or connection resets, especially under high thread counts. Learning to debug and handle these gracefully improved the resilience of our programs.
5. **Time Measurement and Logging**: Accurately capturing turnaround times required consistent use of timing functions, and we learned the importance of repeatable, logged data for good analysis.
6. **Limitations of Iterative Design**: Perhaps the most important lesson was recognizing the trade-offs involved in choosing an iterative architecture. While it was simple to implement and worked well at low volume, it doesn't scale—and we now understand why multithreaded or async servers are needed in many real-world applications.
7. **Team Collaboration**: Working together, dividing responsibilities between client and server components, and cross-testing code helped reinforce effective teamwork in software development.

This project was not just an exercise in coding—it taught us how to test, analyze, and reflect on design decisions, all of which are skills we'll carry forward.