Concurrent Socket Server Java Edition

Abdelrahman Zahra

University of North Florida

**Introduction (Purpose of the Project)**

The Concurrent Socket Server project explores how a multi-threaded server performs when handling multiple simultaneous requests from a multi-threaded client. Built using Java, this client-server setup processes requests in parallel, leveraging a multi-threaded server model to handle concurrency efficiently. The main focus of the project was to evaluate how turn-around time and processing efficiency change as the number of concurrent client requests increases in this parallel environment.

This report covers the server's setup, the client's multi-threaded configuration, and the testing methods used. It provides a detailed analysis of the turn-around times recorded for each operation and includes charts that illustrate how the server's performance scales with increasing concurrent requests. The report also highlights the benefits of the concurrent model in high-demand scenarios.

**Client-Server Setup and Configuration**

**Design of the Client and Server Programs**

The server program was implemented using Java’s ServerSocket class and designed to handle client requests in parallel. Upon startup, the server binds to a specified port and listens for incoming client connections. For each connection, the server spawns a new thread to handle the client’s request. The thread reads the command request, executes the relevant system command (e.g., uptime or memory usage), and returns the result to the client. This concurrent operation model aligns with the project’s goal to assess the performance of a multi-threaded server.

The client program was developed as a multi-threaded Java application. By utilizing Java threads, the client program simulates multiple client connections, each generating a new request to the server. Each client thread records its individual turn-around time (the duration from when the request is sent to when the response is received) and calculates the average turn-around time for a set of requests. This configuration allowed us to simulate concurrent client requests and evaluate how well the server scales under high demand.

**Design Decisions**

1. Choice of Java: Java was chosen for its robust networking libraries and object-oriented capabilities, making it suitable for socket-based programming.
2. Multi-threaded server model: Implementing a multi-threaded server highlighted the advantages of parallel processing in network applications.
3. Multi-threaded client model: The multi-threaded client simulated concurrent client connections to generate load on the server, providing insights into the scalability of the concurrent model.

The server was set up on IP 139.62.210.155 and port 3000, which remained consistent throughout all test cases. Testing included multiple ports to ensure the program's robustness.

**Testing and Data Collection**

**Testing Process**

Testing focused on measuring turn-around times for six different server commands:

1. **Date and Time**: Returns the current server date and time.
2. **Uptime**: Provides the server’s uptime since the last boot-up.
3. **Memory Use**: Shows current memory usage.
4. **Netstat**: Lists all active network connections.
5. **Current Users**: Displays currently logged-in users.
6. **Running Processes**: Lists all active processes on the server.

Each command was tested with varying numbers of concurrent requests: 1, 5, 10, 15, 20, 25, and 100. This allowed us to observe changes in the average turn-around time as the server’s load increased. Each command was executed several times to verify consistency in turn-around times, ensuring that results accurately reflected the server’s behavior under varying load conditions.

**Data Collected**

Below are the average turn-around times collected for each command type with respective charts that illustrate how turn-around times scale with an increasing number of concurrent client requests.

1. **Date and Time**
   * Average Turn-around Times:
     + 1 request: 1 ms
     + 5 requests: 10.8 ms
     + 10 requests: 14.6 ms
     + 15 requests: 9.46 ms
     + 20 requests: 19.4 ms
     + 25 requests: 36.16 ms
     + 100 requests: 366.29 ms

A graph with a line

Description automatically generated

A graph with a line

Description automatically generated

***A graph with a line

Description automatically generated***

Figure Iterative Socket Server

***Analysis:***The **Date and Time** command demonstrates consistent performance for lower request counts, with slight variations that remain manageable. However, as the number of concurrent requests increases, there is a noticeable growth in turn-around time, especially beyond 20 requests. This indicates that while the command is lightweight, the iterative processing begins to strain resources under heavier loads, suggesting that the system may have capacity constraints when handling high-frequency executions of even simple operations.

1. **Uptime**
   * Average Turn-around Times:
     + 1 request: 10.0 ms
     + 5 requests: 12.6 ms
     + 10 requests: 30.5 ms
     + 15 requests: 21.27 ms
     + 20 requests: 23.55 ms
     + 25 requests: 33.72 ms
     + 100 requests: 232.19 ms

A green line graph with black text

Description automatically generated

A graph with a purple line

Description automatically generated

*A graph with a line

Description automatically generated*

Figure Iterative Socket Server

*Analysis*: The **Uptime** command exhibits steady performance for lower request counts, with only minor increases in turn-around time. However, there is some variability in performance as the number of requests rises, particularly at 10 and 25 requests, where the turn-around times spike. This indicates that the command's performance is relatively stable under moderate load but may experience inconsistencies as concurrent requests increase, possibly due to system overhead or resource allocation variations

1. **Memory Use**
   * Average Turn-around Times:
     + 1 request: 10.0 ms
     + 5 requests: 9.0 ms
     + 10 requests: 19.9 ms
     + 15 requests: 18.8 ms
     + 20 requests: 22.75 ms
     + 25 requests: 31.4 ms
     + 100 requests: 240.97 ms

A graph with a red line

Description automatically generated

A graph with a line going up

Description automatically generated

*A graph with blue lines

Description automatically generated*

Figure Iterative Socket Server

*Analysis*: The **Memory Use** command shows consistent performance for small to moderate request volumes, with minimal turn-around time increases up to 15 requests. However, as the number of requests scales beyond 20, there is a noticeable upward trend in average turn-around times, peaking significantly at 100 requests. This pattern suggests that while the command handles lower loads efficiently, higher concurrency begins to strain system resources, likely due to increased data processing or memory contention.

1. **Netstat**
   * Average Turn-around Times:
     + 1 request: 23.0 ms
     + 5 requests: 42.2 ms
     + 10 requests: 59.2 ms
     + 15 requests: 82.8 ms
     + 20 requests: 105.95 ms
     + 25 requests: 153.2 ms
     + 100 requests: 679.17 ms

A graph with a line

Description automatically generated

A graph with a line

Description automatically generated

*A graph with a line

Description automatically generated*

Figure Iterative Socket Server

*Analysis*: The **Netstat** command demonstrates a clear and steady increase in average turn-around times as the number of requests grows. Starting from a moderate 23 ms for a single request, the command's response time escalates significantly with higher loads, peaking at 679.17 ms for 100 requests. This behavior reflects the resource-intensive nature of the command, which involves network status queries and potentially large data handling. The steep rise at higher request volumes suggests that concurrent execution leads to bottlenecks, likely due to increased system I/O and processing overhead. This command is less suitable for high-frequency or bulk execution scenarios without performance optimization.

1. **Current Users**
   * Average Turn-around Times:
     + 1 request: 4.0 ms
     + 5 requests: 5.4 ms
     + 10 requests: 8.3 ms
     + 15 requests: 11.93 ms
     + 20 requests: 25.2 ms
     + 25 requests: 25.56 ms
     + 100 requests: 291.03 ms

A graph with a line

Description automatically generated

A graph with a line

Description automatically generated

*A graph with blue lines

Description automatically generated*

Figure Iterative Socket Server

*Analysis*: The **Current Users** command exhibits low initial turn-around times, starting at just 4.0 ms for a single request, indicating its lightweight and efficient nature. However, as the request volume increases, there is a gradual escalation in response times, with notable jumps beyond 15 requests. The climb to 291.03 ms for 100 requests suggests that the command’s performance is influenced by increased user-session data retrieval and system resource contention. Despite this, the command remains relatively efficient for small to moderate workloads, making it suitable for monitoring scenarios with limited concurrent executions. However, optimization may be needed for high-volume or real-time user status checks.

1. **Running Processes**
   * Average Turn-around Times:
     + 1 request: 27.0 ms
     + 5 requests: 81.4 ms
     + 10 requests: 142.0 ms
     + 15 requests: 168.67 ms
     + 20 requests: 222.0 ms
     + 25 requests: 264.6 ms
     + 100 requests: 867.51 ms

A graph with a line

Description automatically generated

A graph with orange line

Description automatically generated

*A graph with a line

Description automatically generated*

Figure Iterative Socket Server

*Analysis*: The **Running Processes** command demonstrates a clear trend of increasing average turn-around times as request volumes grow. Starting at 27.0 ms for a single request, the times increase significantly, reaching 867.51 ms for 100 requests. This pattern highlights the resource-intensive nature of fetching and listing process data, which likely involves more complex system calls and data aggregation as the load rises. The consistent growth in response time indicates that the command is sensitive to concurrency and system contention, making it less ideal for high-frequency or bulk monitoring scenarios without optimization. It performs adequately for low to moderate workloads but may require parallelization or caching mechanisms to handle larger volumes efficiently.

**Data Analysis**

**Impact of Increasing Client Requests on Turn-around Time**

The data collected showed that as the number of concurrent client requests increased, the turn-around times remained stable for lightweight commands (e.g., Date and Time, Current Users). Resource-intensive commands (e.g., Netstat, Running Processes) exhibited a proportional increase in turn-around time, reflecting the server's need to allocate more processing power for parallel execution. The concurrent model demonstrated significant improvements over the iterative approach, particularly under high load, by processing requests simultaneously instead of sequentially.

**Primary Cause of Impact**

The key improvement stems from the server’s ability to handle multiple requests simultaneously, reducing queueing delays and enhancing responsiveness. By spawning a dedicated thread for each client, the concurrent server mitigates bottlenecks observed in the iterative model.

**Conclusion**

The project demonstrated the clear advantages of a multi-threaded server model for handling high levels of concurrency. Unlike the iterative approach, the concurrent model scaled effectively as the number of client requests increased, maintaining responsiveness and minimizing delays. The results highlight the importance of selecting a concurrent architecture for applications with high concurrency demands.

**Lessons Learned**

1. **Client-server programming fundamentals**: Implementing a concurrent server reinforced key concepts in network communication and parallel programming.
2. **Benefits of parallel processing:** Observing the performance of a multi-threaded server provided insights into the advantages of concurrency in network applications.
3. **Data collection and analysis**: Measuring and analyzing turn-around times revealed how server architecture directly impacts scalability and responsiveness.
4. **Executing system commands in Java:** Gained valuable experience in interacting with OS-level processes from a Java application.

This project showed the importance of designing efficient server architectures to maintain performance under high load, also shows the practical advantages of concurrency for modern networked applications.