Iterative Socket Server Java Edition

Abdelrahman Zahra

University of North Florida

**Introduction (Purpose of the Project)**

The Iterative Socket Server project explored how a single-threaded server performs when handling multiple requests from a multi-threaded client. Built using Java, this client-server setup processes requests one at a time in a single-threaded way. The main focus of the project was to see how turn-around time and processing efficiency change as the number of client requests increases in this single-threaded 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 is affected as concurrent requests increase. The report also highlights how single-threaded servers handle scaling under higher demand.

**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 one at a time. Upon startup, the server binds to a specified port and listens for incoming client connections. When a client connects, the server reads the command request and executes the relevant system command, such as uptime or memory usage. It then returns the result to the client and closes the connection before accepting the next request. This sequential operation model aligns with the project’s goal to assess the performance of a single-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 examine how well the server manages an increasing number of requests.

**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. Alse we are both very used to working with Java because of our past classes and projects
2. **Single-threaded server model**: Implementing a single-threaded server highlighted the effects of iterative 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 iterative model.

The server was set up on IP 139.62.210.155 and port 3000, which remained consistent throughout all test cases. We also did a lot of our testing with multiple ports to ensure the program would run smoothly.

**Testing and Data Collection**

**Testing Process**

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

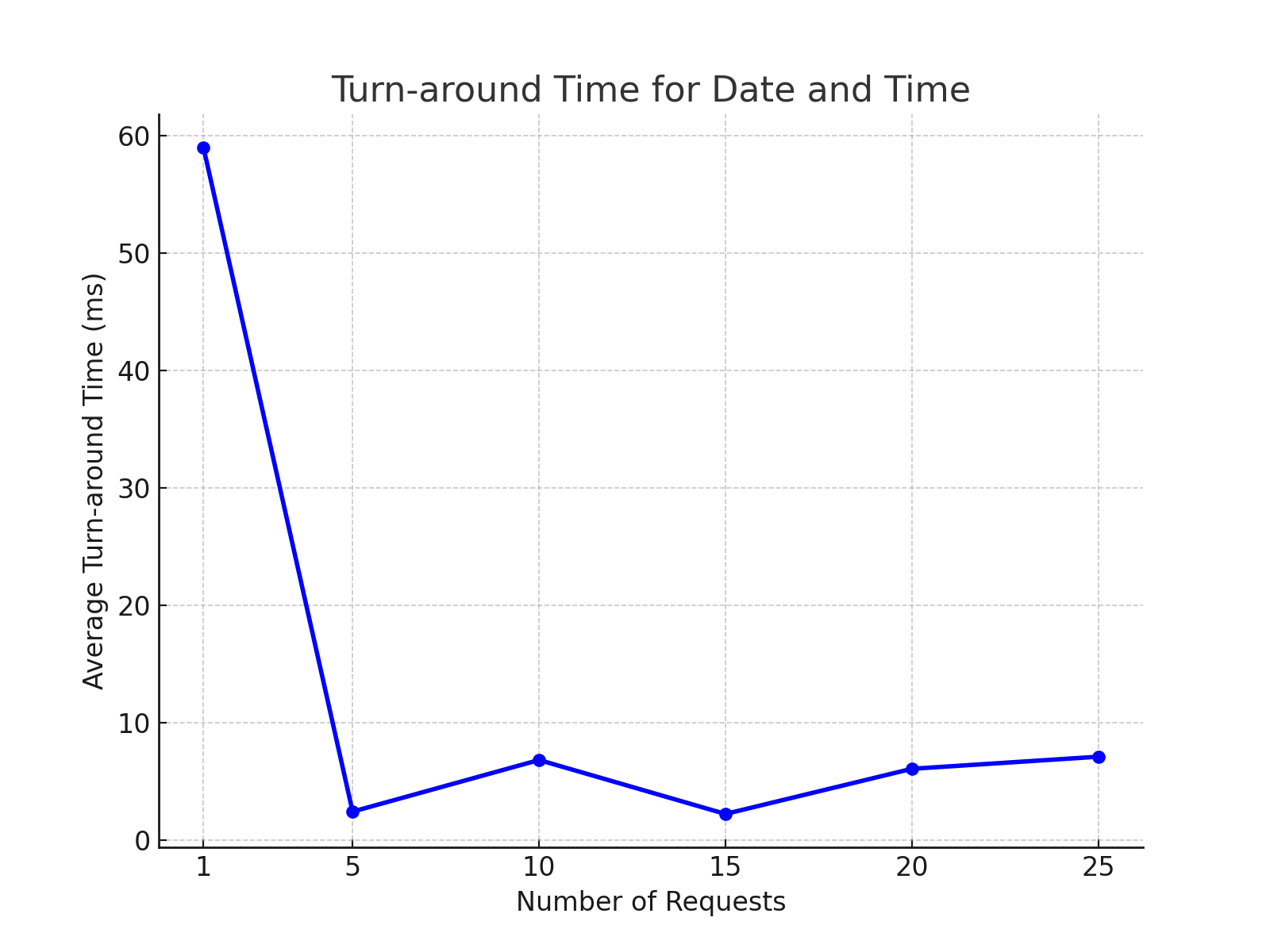
1. **Date and Time**: Returns the current server date and time.
2. **Uptime**: Provides the server’s uptime since the last boot-up.
3. **Memory Use**: Shows current memory usage.
4. **Netstat**: Lists all active network connections.
5. **Current Users**: Displays currently logged-in users.
6. **Running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

Each command was tested with varying numbers of concurrent requests: 1, 5, 10, 15, 20, and 25. 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**

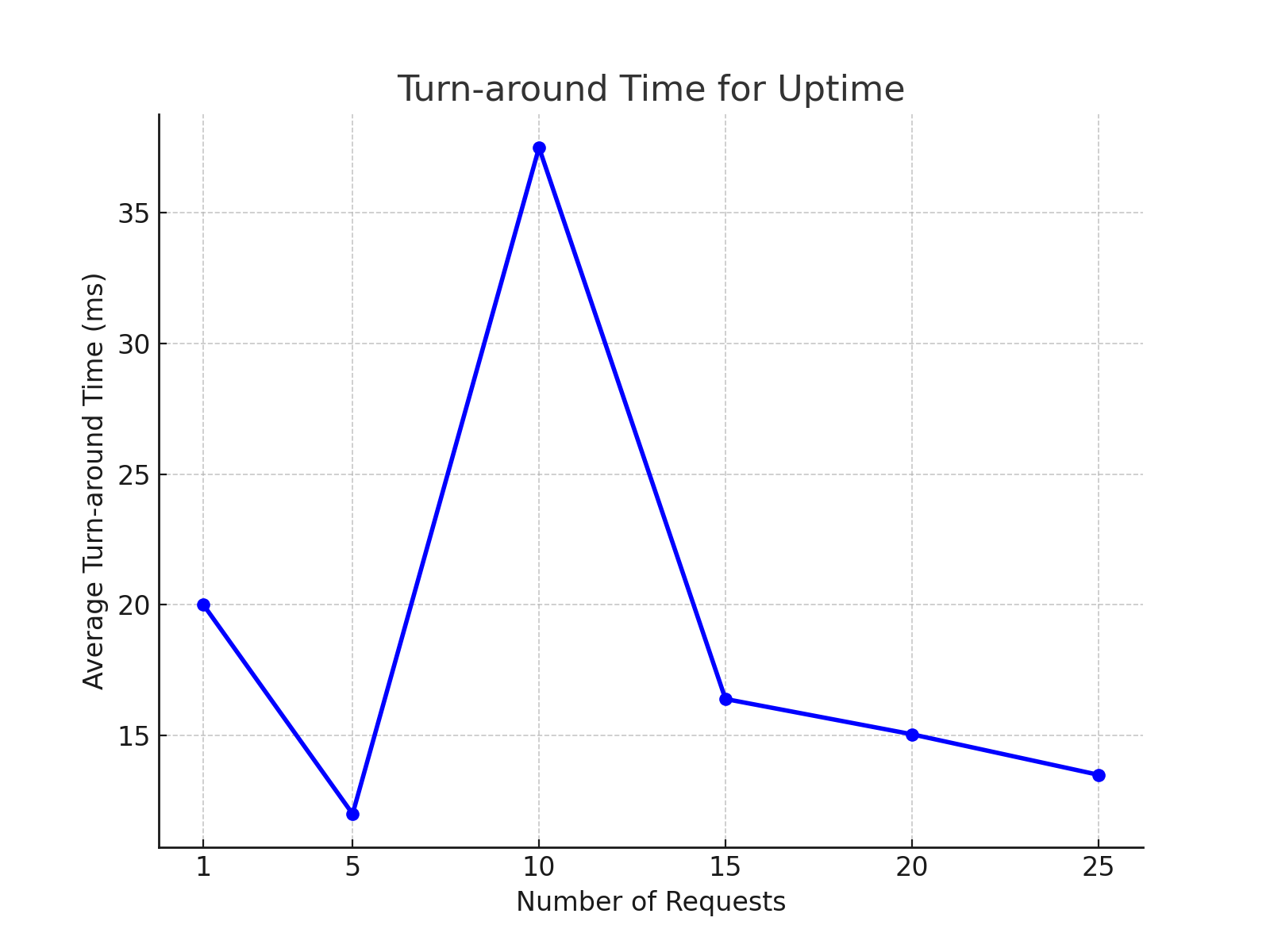
Below are the average turn-around times collected for each command type with respective charts that illustrate how turn-around times scale with an increasing number of concurrent client requests.

1. **Date and Time**
   * Average Turn-around Times:
     + 1 request: 59 ms
     + 5 requests: 2.4 ms
     + 10 requests: 6.8 ms
     + 15 requests: 2.2 ms
     + 20 requests: 6.05 ms
     + 25 requests: 7.08 ms



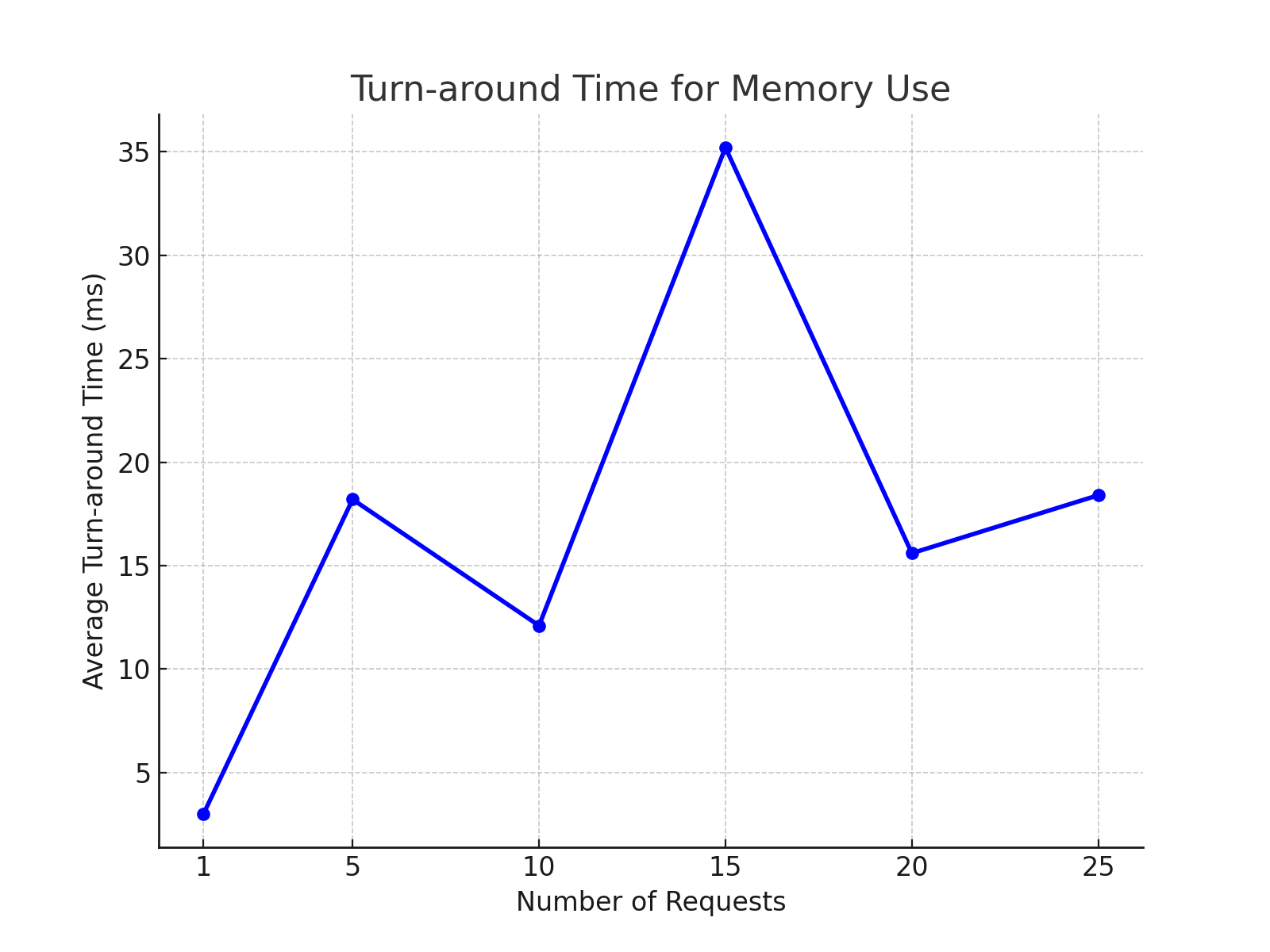
*Analysis*: The **Date and Time** command, being lightweight, maintained stable turn-around times with minimal increases as concurrent requests grew. This indicates that the iterative model can handle simple operations with relatively low impact on performance.

1. **Uptime**
   * Average Turn-around Times:
     + 1 request: 20 ms
     + 5 requests: 12.0 ms
     + 10 requests: 37.5 ms
     + 15 requests: 16.4 ms
     + 20 requests: 15.05 ms
     + 25 requests: 13.5 ms



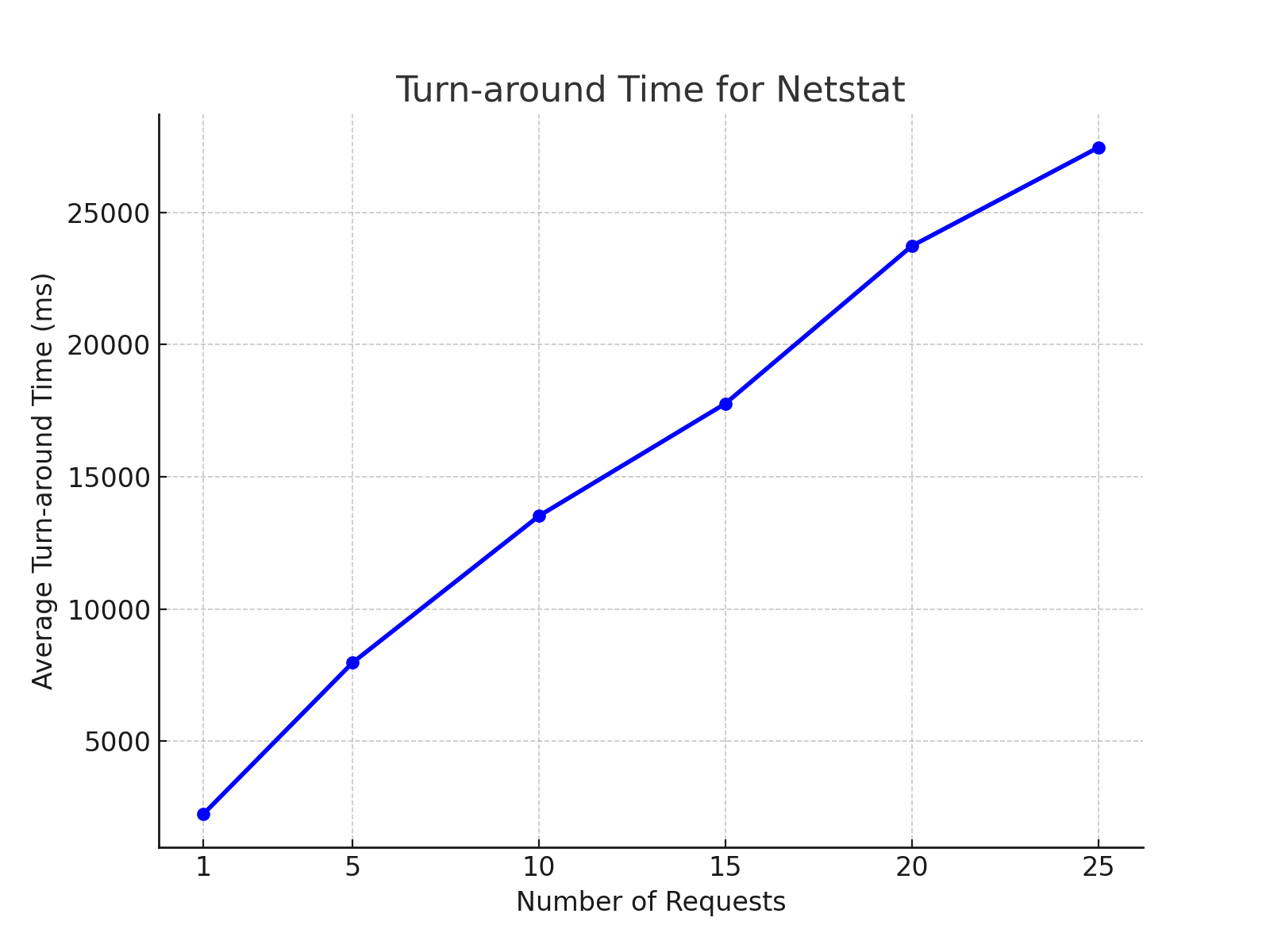
*Analysis*: The **Uptime** command showed a slight increase in turn-around time as requests increased. However, the difference remained moderate, indicating that this command does not impose a heavy load on the server.

1. **Memory Use**
   * Average Turn-around Times:
     + 1 request: 3 ms
     + 5 requests: 18.2 ms
     + 10 requests: 12.1 ms
     + 15 requests: 35.2 ms
     + 20 requests: 15.6 ms
     + 25 requests: 18.4 ms



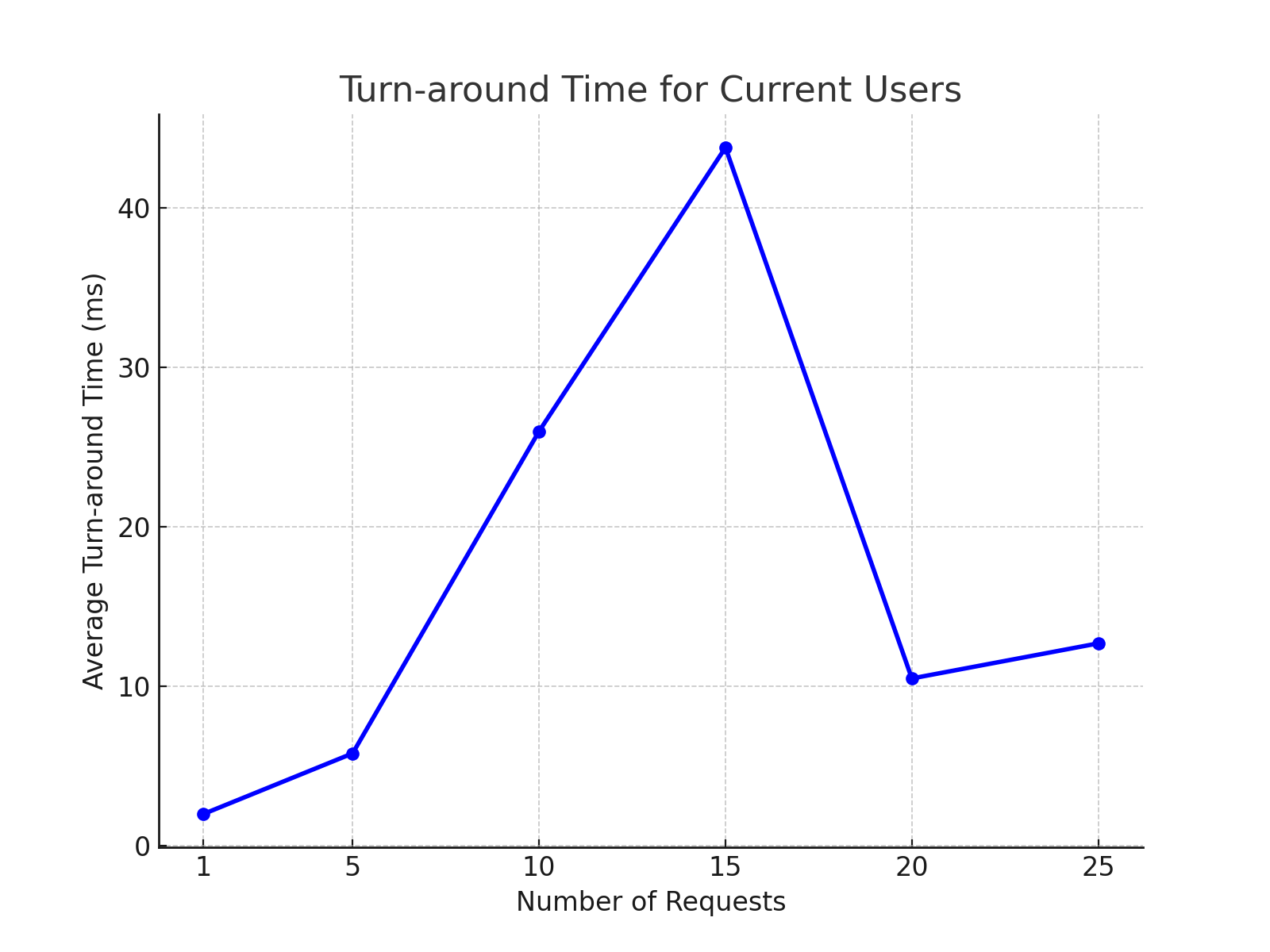
*Analysis*: For **Memory Use**, turn-around times varied more significantly with increased requests. This suggests that while this operation is not overly intensive, it still experienced moderate performance impacts under higher load.

1. **Netstat**
   * Average Turn-around Times:
     + 1 request: 2246 ms
     + 5 requests: 7969.4 ms
     + 10 requests: 13520.3 ms
     + 15 requests: 17771.4 ms
     + 20 requests: 23732.25 ms
     + 25 requests: 27454.2 ms



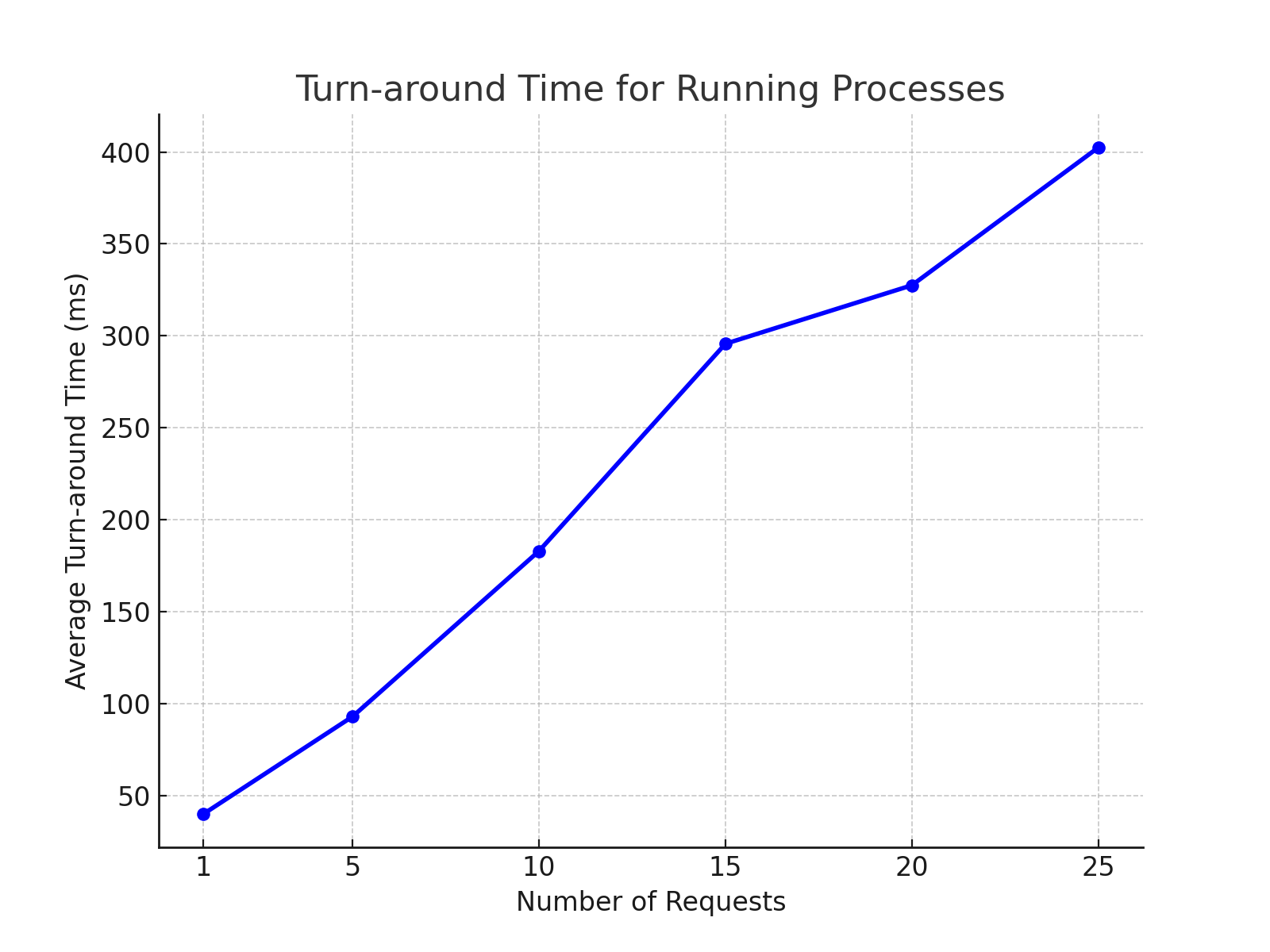
*Analysis*: The **Netstat** command showed a significant increase in turn-around time with each additional request. This resource-intensive command revealed the server’s limitations in handling high-load operations, with delays compounding substantially as the client request count grew.

1. **Current Users**
   * Average Turn-around Times:
     + 1 request: 2 ms
     + 5 requests: 5.8 ms
     + 10 requests: 26.0 ms
     + 15 requests: 43.8 ms
     + 20 requests: 10.5 ms
     + 25 requests: 12.7 ms



*Analysis*: The **Current Users** command displayed relatively low and stable turn-around times across different request counts. The results suggest this command is lightweight and does not place significant demand on the server, even under concurrent requests.

1. **Running Processes**
   * Average Turn-around Times:
     + 1 request: 40 ms
     + 5 requests: 93.0 ms
     + 10 requests: 183.0 ms
     + 15 requests: 295.7 ms
     + 20 requests: 327.5 ms
     + 25 requests: 402.4 ms



*Analysis*: **Running Processes** is another resource-heavy command, and as the data shows, it incurred increasing turn-around times as concurrent requests grew. This operation’s delays reflect the server’s limitations in managing multiple CPU-intensive tasks in an iterative setup.

**Data Analysis**

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

The data and charts show that as the number of concurrent client requests goes up, turn-around times generally increase. This effect was most noticeable with commands like Netstat and Running Processes, which need more processing power. On the other hand, simpler commands like Date and Time and Current Users only had small increases in turn-around times, suggesting that they aren’t as impacted by the server's iterative way of handling requests.

**Primary Cause of Impact**

The main reason for the increased turn-around times is the server’s single-threaded, sequential processing model. Because the server handles requests one by one, any extra requests have to wait in line, leading to delays—especially for operations that use a lot of CPU power. This waiting, or queueing effect, is what causes the rise in turn-around times we observed.

**Conclusion**

The project showed that a single-threaded server model isn’t ideal for handling high levels of concurrency, especially with resource-intensive operations. As the number of client requests went up, the server’s turn-around got worse, highlighting the performance limits of an iterative approach. The data makes it clear that for applications needing high concurrency, a multi-threaded or parallel-processing model would deliver much better performance.

**Lessons Learned**

1. **Client-server programming fundamentals**: Using Java for socket programming reinforced key concepts in network communication.
2. **Iterative processing limitations**: Observing the performance of a single-threaded server under load provided insight into the constraints of sequential processing.
3. **Data collection and analysis**: Measuring and analyzing turn-around times highlighted how server architecture affects responsiveness.
4. **Running system commands in Java**: Gained experience in executing system commands on a server, which proved useful for interacting with OS-level processes.

This project underscored the importance of selecting the right server model for high-demand applications, emphasizing the need for efficient server architecture to maintain responsiveness under load.