**Iterative Socket Server**

Michael Morrison, Jacob Patterson

CNT4504 - Computer Networks & Distributed Processing

Prof. John Kelly

**Introduction**

The purpose of this project is to create a single-threaded, or iterative, server program which interacts with a multi-threaded client program. The client connects to the server’s port, and the server listens for commands requested by the client before giving the appropriate response. The goal of the project is to help grasp a better understanding of how computer network programming functions by taking a more hands-on approach. This report delves into the intricacies of the project, shedding light on the configuration of the client-server system, testing procedures, data collection, analysis of results, and the lessons learned throughout the development process.

**Client-Server Setup and Configuration**

The client-server system consists of two separate programs. The server, implemented in the *IterativeServer.java* program, was intended to be an Iterative or Single-Threaded Server, capable of processing one client request at a time. Meanwhile, the client, realized in the *MultiThreadedClient.java* program, was designed as a Multi-Threaded Client to enable the creation of multiple client sessions.

The *IterativeServer.java* program begins by accepting a port number as a command-line argument and establishes a *ServerSocket* to wait for incoming connections. Upon accepting a client connection, the server prompts the client to choose a request from a predefined list (1 to 6) via the *PrintWriter* output stream. The client's choice is then processed using the *processRequest* method, which directs the flow to specific methods based on the chosen option.

Each option, from 1 to 6, corresponds to a system-related query: date and time, uptime, memory usage, network connections, current users, and running processes, respectively. The server executes shell commands using *ProcessBuilder* for options 2 to 6, captures the output, and sends it back to the client via the *PrintWriter* output stream.

The code utilizes a modular approach, encapsulating the logic for each type of request in separate methods *(getCurrentDateAndTime(), getUptime(), getMemoryUsage(), getNetworkConnections(), getCurrentUsers(), and getRunningProcesses())*. Error handling is implemented for potential I/O exceptions that might occur during the execution of shell commands or socket communication.

In the *MultiThreadedClient.java* program, the program takes command-line arguments for server address, server port, and the number of clients to be created. It initializes a loop based on the specified number of clients. Within the loop, a new *ClientThread* object is created and started. Each *ClientThread* represents an individual client connection.

The *ClientThread* class extends the *Thread* class, allowing each client to operate concurrently. Inside the *run* method, the client establishes a socket connection with the server using the provided server address and port. It sets up input and output streams to send and receive data. The client then prompts the user with a menu received from the server and waits for the user's input. The user's choice is sent to the server, and the client receives and prints the server's response.

The program’s multi-threaded approach enables multiple clients to interact with the server simultaneously. Each client operates independently in its thread, ensuring that one client's actions do not interfere with another's. This design facilitates efficient handling of multiple client connections, enhancing the overall responsiveness and performance of the application.

We made several key decisions when designing the Iterative Server:

1. Clients are given a menu of commands to choose from (e.g., Date and Time, Uptime, etc.). The server processes the selected command and responds accordingly.
2. The server program allows for the specification of the listening port as a command-line argument, providing flexibility in terms of port selection.

The following design decisions were made regarding the Multi-Threaded Client:

1. The client program presents users with a menu of available commands to choose from. Users can also specify the number of client requests they want to generate.
2. Both the server address and port are provided as command-line arguments, giving users the flexibility to connect to servers with different configurations.

The client-server system operated as follows:

1. The server listens on the specified port, awaiting incoming client connections.
2. After connecting, the client selects a command from the menu, which is processed by the server.
3. The server then sends the appropriate response back to the client, which displays the results on the client side.
4. The client program allows users to select the number of client requests they want to generate (1, 5, 10, 15, 20 or 25). For each request, a new client thread is created.

**Testing and Data Collection**

Our testing process for the Iterative Server involved the following steps:

1. We configured the server on a designated host machine, specifying the port to listen on.
2. We set up the client to connect to the server's host and port. The number of client requests to generate was also defined.
3. For each client request generated, a client thread was launched to establish a connection with the server.
4. During testing, we collected data regarding the total and average turnaround time for each of the six available operations: Date and Time, Uptime, Memory Use, Netstat, Current Users, and Running Processes.
5. The testing process was repeated with varying numbers of client requests to assess how the server's performance scales under different workloads.

**Data Collected**

We gathered the following data for each operation:

1. **Date and Time:** Includes day of week, month and day, current time in 00:00:00 format, time zone and year (Ex: Fri Nov 03 19:51:54 UTC 2023)
2. **Uptime:** Includes current uptime of machine with current time in 00:00:00 format and number of days (Ex: 19:54:40 up 302 days)
3. **Memory Usage:** Displays the amount of total, used and free memory and swap space in kilobytes, as well as the amount of shared, buff/cache and available memory space.
4. **Netstat:** Displays information on network connections including active internet connections and their addresses.
5. **Current Users:** Displays the list of all current users of the server as well as the date and time of connection and their IP addresses.
6. **Running Processes:** Displays all of the processes currently being run by the server.

We created a bar graph to compare the amount of total turn-around time related to each operation for each number of client requests as well as the average. This data can be seen below:

**Data Analysis**

Based on the graphs above, it appears that increasing the number of clients for each operation generally increased the total turn-around time, with slight deviation, but decreased the average, with the decrease from 1 client to 5 being especially drastic for all operations. Operation 6 also had a much higher turn-around time than the other operations in all instances, with Operation 1 being generally the lowest. It is likely that the increase in the number of clients led to a greater workload for the server and therefore it took longer to process the entire queue of operations, yet the average was lower since that total time was distributed across all the clients at once. For Operation 6, it is possible that since the server had to retrieve a considerably extensive amount of data compared to the other operations, the turn-around time was much greater.

**Conclusions**

From our Data Analysis, we can conclude that having an increased number of clients yielded a higher total, but a lower average turn-around time. We can also conclude that operations that rely on retrieving or computing large amounts of data will lead to a higher turn-around due to the greater demand from the server.

**Lessons Learned**

This project taught us a number of unique lessons. For one, it gave us the experience of creating working with a network-based server system in Java. It also taught us how to debug a small Java server when our output wasn’t what we desired. It also helped us gain greater experience in working as a team on a network-related application. We additionally discovered that a high number of clients within a server can correlate with longer total turn-around times but shorter averages. One obstacle we faced was that initially, most of our operations were not producing any output. After a few rewrites, however, we were soon back on track and able to record the data for the project.