

Concurrent socket server

Jacob Patterson, Michael Morrison



CNT4504 - Computer Networks & Distributed Processing

Prof. john kelly

**Introduction**

The purpose of this project is to create a multi-threaded, or concurrent, socket server, which will interact with a similarly multi-threaded client. This contrasts with the previous project, in which the socket server was designed to be single-threaded, or iterative. As before, the client will connect to the server’s port, after which the server will listen for commands requested by the client before giving an appropriate response. The goal of this project is to gain personal experience creating a multi-threaded server program, as well as to see how such a program operates and performs differently compared to a single-threaded one. This report provides details on the design and configuration of the client-server system, as well as testing procedures, data collection, analysis of results, and the lessons learned throughout the development process.

**Client-Server Setup and Configuration**

The client system remains essentially unchanged from the previous project. It is multi-threaded, meaning that it allows for multiple client sessions to occur at a time. Like before, it works by accepting command line arguments entered by the user for the server address, server port, and the number of clients to be created. Each client is represented by a *ClientThread* object, extending *Thread*, allowing concurrent operation.

In the run method, the client establishes a socket connection with the server, sets up input/output streams, and prompts the user with a menu received from the server. It waits for user input, sends it to the server, and prints the server's response. This multi-threaded approach ensures independent operation for each client, enabling efficient handling of multiple client connections without unwanted interference. In addition, it significantly enhances the application’s performance and responsiveness.

The server program, on the other hand, differs from the previous project in that it is also multi-threaded, meaning that it can handle multiple client requests simultaneously instead of carrying them out one at a time. Whereas the Iterative server allowed for handling 1, 5, 10, 15, 20, or 25 client requests at once, this one is also capable of carrying out 100 at the same time. The program begins by creating a *ServerSocket* to listen for incoming client connections. From there, it uses the *ExecutorService* function to handle the incoming connections concurrently using a cached thread pool, which dynamically manages the number of threads within the pull based on the workload given. The *ClientHandler* class then accepts the incoming client connections in an infinite loop and submits tasks to the thread pool for processing. Next, it implements the *Runnable* interface in order to handle the client requests. Once it receives a connection, the client is prompted to choose a request option represented by numbers 1-6 (representing, respectively, date and time, uptime, memory usage, network connections, current users, and running processes). Based on the option chosen, the request is processed, and the corresponding system information is sent back to the client. 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.

We made several key decisions when designing the Concurrent 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.

3. Implementation of an *ExecutorService* with a cached thread pool for handling client connections, ensuring efficient resource allocation.

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, 25 or 100). For every request, the client program initiates a new client thread, allowing multiple simultaneous interactions with the server.

**Testing and Data Collection**

Our testing process for the Concurrent 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 each operation. Operation 6 also had a much higher total and average turn-around time than the other operations in nearly all instances, with Operation 1 being generally the lowest. One thing that was observed during recording of the time amounts was that the average time seemed to generally equal the total time divided by the number of clients (for example, if 5 clients requesting operation 1 generated a total time of 15 ms, then the average time came out to 3 ms). It can be inferred that the multi-threaded server was “distributing” the total time across each thread. However, the particularly large amount of information involved with operation 6 led to large turn-around times regardless.

**Conclusions**

From our Data Analysis, we can conclude that having an increased number of clients yielded a higher total and a lower average turn-around time. We can also conclude that operations that rely on retrieving large amounts of data will lead to a higher turn-around due to the greater demand from the server. In addition, we learned how a multi-threaded server divides the average turn-around time dynamically depending on the number of requests received.

**Lessons Learned**

One of the biggest lessons we learned when doing this project was the difference in functionality between a multi-threaded and single-threaded server. It also taught us how to create such a server, which was overall similar to our previous but with several key differences and additions. Initially, we faced difficulty with figuring out how to modify our Iterative server to become Concurrent, but after understanding concepts such as cached thread pools, we were able to transition to a multi-threaded program with our newfound knowledge.