5700 Assignment 1 Report: Socket Programming

Tony

September 30, 2025

1 Program Description

This project is a Java client-server application demonstrating socket programming, multi-threading, and basic protocol design. It includes a multi-threaded server for concurrent clients, a single-threaded version for sequential handling, and a command-line client.

The communication protocol is text-based, using comma-separated strings. The server validates client input and shuts down if it receives a number outside the predefined range [1, 100].

2 Choice of Programming Language

Java was chosen for this project due to its:

- **Rich Networking API:** Java's standard library (`java.net`) provides a powerful and straightforward API for socket programming. Classes like `Socket`, `ServerSocket`, and `InetAddress` abstract away many of the low-level complexities of network communication, making it easier to build robust client-server applications.
- **Built-in Multi-threading:** Concurrency is a core requirement for the server to handle multiple clients. Java has excellent built-in support for multi-threading through the `Thread` class and the `Runnable` interface, simplifying the development of a concurrent server.
- **Platform Independence:** The Java Virtual Machine (JVM) allows the compiled Java bytecode to run on any platform where a JVM is available (e.g., Windows, macOS, Linux). This "write once, run anywhere" capability is highly beneficial for network applications that may be deployed in diverse environments.
- **Strong Typing and Object-Oriented:** Java's static typing helps catch errors at compile time, and its object-oriented nature allows for a modular and maintainable program design. This is evident in the use of a dedicated `ClientHandler` class to encapsulate the logic for handling a single client.

3 Program Design

The application design separates client and server components.

3.1 Communication Protocol

The client and server communicate over a TCP socket connection. The messages exchanged are simple comma-separated strings.

- **Client to Server:** The client sends a message in the format: `CLIENT_NAME,CLIENT_NUMBER`. For example: `"Client of Tony,50"`.
- **Server to Client:** The server responds with a message in the format: `SERVER_NAME,SERVER_NUMBER`. For example: `"Server of Tony,42"`.
- **Error/Shutdown Condition:** If the client sends a number outside the valid range of 1-100, the server sends a shutdown message and terminates.

3.2 Multi-threaded Server (`SocketServer.java`)

The main server is designed to be concurrent, allowing it to handle multiple client connections at the same time. This is achieved using Java's threading capabilities.

- **Main Thread:** The main server thread initializes a `ServerSocket` and enters an infinite loop, waiting for client connections by calling `serverSocket.accept()`.
- **Client Handler Thread:** For each incoming connection, a new `Socket` object is created. This socket is then passed to a new `ClientHandler` runnable object, which is executed in a separate thread. This design allows the main thread to immediately return to waiting for new connections, ensuring the server remains responsive.

The core logic for processing a client's request is encapsulated in the `handleClientRequest` method within the `ClientHandler` class.

```
// ...
while (!serverSocket.isClosed()) {
    try {
        Socket clientSocket = serverSocket.accept();
        System.out.println("[Server] Accepted connection from " + clientSocket.
    getInetAddress());
        // Create a new thread to handle the client connection
        new Thread(new ClientHandler(clientSocket, serverSocket)).start();
    } catch (IOException e) {
        // ...
    }
}
// ...
```

Listing 1: SocketServer main loop for accepting connections

3.3 Single-threaded Server (`SocketServerSingleThread.java`)

For comparison and simplicity, a single-threaded version of the server is also provided. This server handles only one client at a time. After accepting a connection, it processes the entire request and sends a response before it can accept a new connection. This version reuses the same `handleClientRequest` logic from the multi-threaded server's `ClientHandler` to maintain consistency.

```
// ...
while (!serverSocket.isClosed()) {
    try {
        Socket clientSocket = serverSocket.accept();
        System.out.println("[SingleThreadServer] Accepted connection from " +
    clientSocket.getInetAddress());
        handleClient(clientSocket, serverSocket);
    } catch (IOException e) {
        // ...
    }
}
// ...
```

Listing 2: SocketServerSingleThread main loop

3.4 Client (`SocketClient.java`)

The client is a simple command-line application that performs the following steps:

1. Prompts the user to enter an integer between 1 and 100.
2. Establishes a connection to the server at `localhost` on port `5701`.
3. Sends its name and the user-provided number to the server.
4. Waits for and receives the server's response.
5. Parses the server's name and number from the response.

6. Calculates and prints the sum of the client's and server's numbers.
7. Closes the connection.

```
// ...
try (Socket socket = new Socket(SERVER_HOST, SERVER_PORT);
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
     BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream
    ())) {

    System.out.println("[Client] Connected to server.");

    String message = CLIENT_NAME + "," + clientNumber;
    out.println(message);
    System.out.println("[Client] Sent: " + message);

    String response = in.readLine();
    System.out.println("[Client] Received: " + response);
    // ...
}
// ...
```

Listing 3: SocketClient communication logic

4 Features

- **Concurrency:** The primary server (`SocketServer.java`) can serve multiple clients simultaneously, making it scalable.
- **Input Validation:** The server validates the number received from the client. If the number is not within the range [1, 100], the server initiates a shutdown procedure. This provides a mechanism for controlled termination.
- **Modularity:** The client handling logic is encapsulated within the `ClientHandler` class, separating it from the main server connection-listening loop. This logic is reused by the single-threaded server.

5 User Guide

This guide provides instructions on how to compile, run, and test the application.

5.1 Prerequisites

A Java Development Kit (JDK) version 8 or higher must be installed on your system.

5.2 How to Compile

Open a terminal or command prompt, navigate to the project's `src` directory, and execute the following commands to compile all Java source files:

```
javac SocketServer.java
javac SocketServerSingleThread.java
javac SocketClient.java
```

5.3 How to Run

1. **Start the Server:** You can run either the multi-threaded or single-threaded server.

   To run the **multi-threaded server**, execute:

   ```
   java SocketServer
   ```

   To run the **single-threaded server**, execute:

```
java SocketServerSingleThread
```

The server will start and print a message indicating it is waiting for client connections.

2. **Run the Client:** Open a **new** terminal window, navigate to the `src` directory, and run:

```
java SocketClient
```

The client will prompt you to enter a number. You can run multiple instances of the client to test the multi-threaded server's concurrency.

6 Test Plan

The test plan ensures application correctness and includes both automated unit tests and manual system tests.

6.1 Unit Tests

Unit tests were written using the JUnit 5 framework to validate the core business logic of the server in isolation.

- **Test Environment:** JUnit 5.
- **Test File:** `SocketServerTest.java`.
- **Target Method:** `handleClientRequest(String clientName, int clientNumber)`.

6.1.1 Test Cases

1. **Test Case 1: Valid Input**
   - **Objective:** Verify the server's response for a valid input number within the range [1, 100].
   - **Input:** `clientName = "TestClient"`, `clientNumber = 50`.
   - **Expected Output:** The server returns a string containing its name and number, e.g., `"Server of Tony,42"`.
   - **Result:** Passed.
2. **Test Case 2: Boundary Input (Low)**
   - **Objective:** Verify the server's shutdown response for an out-of-range number just below the valid minimum.
   - **Input:** `clientName = "TestClient"`, `clientNumber = 0`.
   - **Expected Output:** The server returns the shutdown message: `"Server is shutting down due to out-of-range input."`.
   - **Result:** Passed.
3. **Test Case 3: Boundary Input (High)**
   - **Objective:** Verify the server's shutdown response for an out-of-range number just above the valid maximum.
   - **Input:** `clientName = "TestClient"`, `clientNumber = 101`.
   - **Expected Output:** The server returns the shutdown message: `"Server is shutting down due to out-of-range input."`.
   - **Result:** Passed.

6.2 Manual System Testing

Manual tests were conducted to verify the end-to-end functionality of the client-server interaction.

6.2.1 Test Cases

1. **Test Case 1: Single Client Connection**
   - **Objective:** Ensure a single client can connect, communicate, and receive the correct sum.
   - **Steps:**
     (a) Start the server.
     (b) Start one client.
     (c) Enter a valid number (e.g., 60).
   - **Expected Output:** The client and server consoles should display the names, numbers, and the correct sum (60 + 42 = 102). The connection should terminate gracefully.
   - **Result:** Passed.
2. **Test Case 2: Concurrent Client Connections (Multi-threaded Server)**
   - **Objective:** Verify the multi-threaded server can handle multiple clients simultaneously.
   - **Steps:**
     (a) Start the multi-threaded server.
     (b) Start three clients in quick succession.
     (c) Enter different valid numbers for each client.
   - **Expected Output:** The server should handle all three clients concurrently without blocking. Each client should receive its correct, distinct response from the server.
   - **Result:** Passed.
3. **Test Case 3: Server Shutdown**
   - **Objective:** Confirm that the server shuts down when a client sends an out-of-range number.
   - **Steps:**
     (a) Start the server.
     (b) Start a client and enter an invalid number (e.g., 150).
     (c) Attempt to start another client.
   - **Expected Output:** The first client receives a shutdown message. The server terminal indicates it is shutting down. The second client fails to connect.
   - **Result:** Passed.

7 Screenshots

This section is reserved for screenshots showing the output from the collaborative portion and testing.

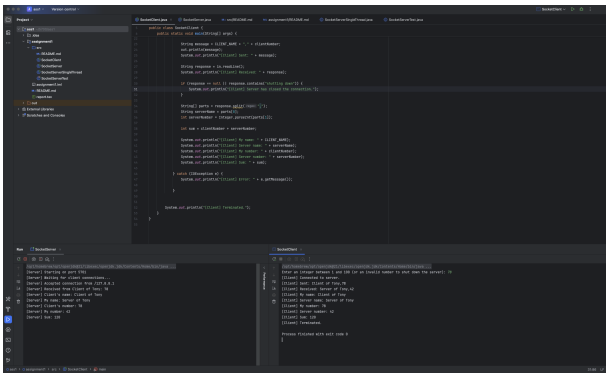

Figure 1: Normal operation with a single client.

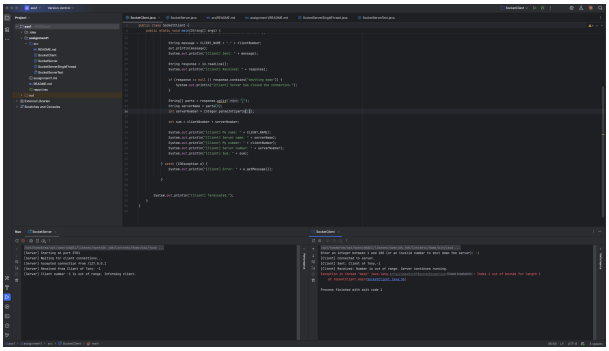

Figure 2: Server shutdown test.

A Source Code

A.1 SocketServer.java

```java
import java.io.*;
import java.net.*;

public class SocketServer {
    public static final int SERVER_PORT = 5701;
    public static final String SERVER_NAME = "Server of Tony";
    public static final int SERVER_NUMBER = 42;

    public static void main(String[] args) {
        System.out.println("[Server] Starting on port " + SERVER_PORT);
        try (ServerSocket serverSocket = new ServerSocket(SERVER_PORT)) {
            System.out.println("[Server] Waiting for client connections...");
            while (!serverSocket.isClosed()) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    System.out.println("[Server] Accepted connection from " +
    clientSocket.getInetAddress());
                    // Create a new thread to handle the client connection
                    new Thread(new ClientHandler(clientSocket, serverSocket)).start();
                } catch (IOException e) {
                    if (serverSocket.isClosed()) {
                        System.out.println("[Server] Server socket closed, shutting down
    .");
                        break;
                    }
                    System.out.println("[Server] Error accepting client connection: " +
    e.getMessage());
                }
            }
        } catch (IOException e) {
            System.out.println("[Server] Error starting server: " + e.getMessage());
        }
        System.out.println("[Server] Terminated.");
    }

    static class ClientHandler implements Runnable {
        private final Socket clientSocket;
        private final ServerSocket serverSocket;

        public ClientHandler(Socket socket, ServerSocket serverSocket) {
            this.clientSocket = socket;
            this.serverSocket = serverSocket;
        }

        /**
         * Processes the client request and returns a response string.
         * This method contains the core logic for handling client input.
         * @param clientName The name of the client.
         * @param clientNumber The number sent by the client.
         * @return A string response to be sent back to the client.
```

```
48          */
49         public String handleClientRequest(String clientName, int clientNumber) {
50             System.out.println("[Server] Received from " + clientName + ": " +
    clientNumber);
51 
52             if (clientNumber < 1 || clientNumber > 100) {
53                 System.out.println("[Server] Client number " + clientNumber + " is out
    of range. Shutting down server.");
54                 try {
55                     if (serverSocket != null && !serverSocket.isClosed()) {
56                         serverSocket.close();
57                     }
58                 } catch (IOException e) {
59                     System.out.println("[Server] Error closing server socket: " + e.
    getMessage());
60                 }
61                 return "Server is shutting down due to out-of-range input.";
62             }
63 
64             System.out.println("[Server] Client's name: " + clientName);
65             System.out.println("[Server] My name: " + SocketServer.SERVER_NAME);
66             System.out.println("[Server] Client's number: " + clientNumber);
67             System.out.println("[Server] My number: " + SocketServer.SERVER_NUMBER);
68             int sum = clientNumber + SocketServer.SERVER_NUMBER;
69             System.out.println("[Server] Sum: " + sum);
70 
71             // Send server name and number back to the client
72             return SocketServer.SERVER_NAME + "," + SocketServer.SERVER_NUMBER;
73         }
74 
75         public void run() {
76             try (BufferedReader in = new BufferedReader(new InputStreamReader(
    clientSocket.getInputStream()));
77                  PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)
    ) {
78 
79                 String message = in.readLine();
80                 if (message == null) {
81                     System.out.println("[Server] Client disconnected before sending a
    message.");
82                     return;
83                 }
84 
85                 String[] parts = message.split(",");
86                 if (parts.length != 2) {
87                     System.out.println("[Server] Invalid message format from client.");
88                     return;
89                 }
90 
91                 String clientName = parts[0];
92                 int clientNumber = Integer.parseInt(parts[1]);
93 
94                 String response = handleClientRequest(clientName, clientNumber);
95                 out.println(response);
96 
97             } catch (IOException e) {
98                 System.out.println("[Server] Error handling client: " + e.getMessage());
99             } catch (NumberFormatException e) {
100                 System.out.println("[Server] Invalid number format from client.");
101             } finally {
102                 try {
103                     clientSocket.close();
104                 } catch (IOException e) {
105                     System.out.println("[Server] Error closing client socket: " + e.
    getMessage());
106                 }
107             }
108         }
109     }
110 }
```

Listing 4: SocketServer.java - Multi-threaded Server

A.2 SocketServerSingleThread.java

```
import java.io.*;
import java.net.*;

public class SocketServerSingleThread {
    public static void main(String[] args) {
        System.out.println("[SingleThreadServer] Starting on port " + SocketServer.SERVER_PORT);
        try (ServerSocket serverSocket = new ServerSocket(SocketServer.SERVER_PORT)) {
            System.out.println("[SingleThreadServer] Waiting for client connections...");
            while (!serverSocket.isClosed()) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    System.out.println("[SingleThreadServer] Accepted connection from " + clientSocket.getInetAddress());
                    handleClient(clientSocket, serverSocket);
                } catch (IOException e) {
                    if (serverSocket.isClosed()) {
                        System.out.println("[SingleThreadServer] Server socket closed, shutting down.");
                        break;
                    }
                    System.out.println("[SingleThreadServer] Error accepting client connection: " + e.getMessage());
                }
            }
        } catch (IOException e) {
            System.out.println("[SingleThreadServer] Error starting server: " + e.getMessage());
        }
        System.out.println("[SingleThreadServer] Terminated.");
    }

    private static void handleClient(Socket clientSocket, ServerSocket serverSocket) {
        try (BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)) {

            String message = in.readLine();
            if (message == null) {
                System.out.println("[SingleThreadServer] Client disconnected before sending a message.");
                return;
            }

            String[] parts = message.split(",");
            if (parts.length != 2) {
                System.out.println("[SingleThreadServer] Invalid message format from client.");
                return;
            }

            String clientName = parts[0];
            int clientNumber = Integer.parseInt(parts[1]);

            // Reuse SocketServer.ClientHandler's handleClientRequest method
            SocketServer.ClientHandler handler = new SocketServer.ClientHandler(clientSocket, serverSocket);
            String response = handler.handleClientRequest(clientName, clientNumber);
            out.println(response);

        } catch (IOException e) {
            System.out.println("[SingleThreadServer] Error handling client: " + e.getMessage());
        } catch (NumberFormatException e) {
            System.out.println("[SingleThreadServer] Invalid number format from client.");
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
```

```
                System.out.println("[SingleThreadServer] Error closing client socket: "
    + e.getMessage());
            }
        }
    }
}
```

Listing 5: SocketServerSingleThread.java - Single-threaded Server

A.3 SocketClient.java

```
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class SocketClient {
    public static void main(String[] args) {
        final String SERVER_HOST = "localhost";
        final int SERVER_PORT = 5701;
        final String CLIENT_NAME = "Client of Tony";

        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an integer between 1 and 100 (or an invalid number to
    shut down the server): ");
        int clientNumber = scanner.nextInt();


        if (clientNumber < 1 || clientNumber > 100) {
            System.out.println("You entered a number out of range. The server should
    shut down.");
        }

        try (Socket socket = new Socket(SERVER_HOST, SERVER_PORT);
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.
    getInputStream()))) {

            System.out.println("[Client] Connected to server.");

            String message = CLIENT_NAME + "," + clientNumber;
            out.println(message);
            System.out.println("[Client] Sent: " + message);

            String response = in.readLine();
            System.out.println("[Client] Received: " + response);

            if (response == null || response.contains("shutting down")) {
                System.out.println("[Client] Server has closed the connection.");
                return;
            }

            String[] parts = response.split(",");
            String serverName = parts[0];
            int serverNumber = Integer.parseInt(parts[1]);

            int sum = clientNumber + serverNumber;

            System.out.println("[Client] My name: " + CLIENT_NAME);
            System.out.println("[Client] Server name: " + serverName);
            System.out.println("[Client] My number: " + clientNumber);
            System.out.println("[Client] Server number: " + serverNumber);
            System.out.println("[Client] Sum: " + sum);

        } catch (IOException e) {
            System.out.println("[Client] Error: " + e.getMessage());
        }
        System.out.println("[Client] Terminated.");
    }
}
```

Listing 6: SocketClient.java - Client Application

A.4 SocketServerTest.java

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class SocketServerTest {

    private SocketServer.ClientHandler clientHandler;

    @BeforeEach
    void setUp() {
        // For testing the handleClientRequest method, we don't need a real socket.
        // We can pass null for the Socket and ServerSocket objects.
        clientHandler = new SocketServer.ClientHandler(null, null);
    }

    @Test
    void testHandleClientRequest_ValidNumber() {
        String clientName = "TestClient";
        int clientNumber = 50;
        String expectedResponse = SocketServer.SERVER_NAME + "," + SocketServer.SERVER_NUMBER;

        String actualResponse = clientHandler.handleClientRequest(clientName, clientNumber);

        assertEquals(expectedResponse, actualResponse, "Response for a valid number should contain server name and number.");
    }

    @Test
    void testHandleClientRequest_NumberTooLow() {
        String clientName = "TestClient";
        int clientNumber = 0; // Out of range (less than 1)
        String expectedResponse = "Server is shutting down due to out-of-range input.";

        String actualResponse = clientHandler.handleClientRequest(clientName, clientNumber);

        assertEquals(expectedResponse, actualResponse, "Response for a number less than 1 should be a shutdown message.");
    }

    @Test
    void testHandleClientRequest_NumberTooHigh() {
        String clientName = "TestClient";
        int clientNumber = 101; // Out of range (greater than 100)
        String expectedResponse = "Server is shutting down due to out-of-range input.";

        String actualResponse = clientHandler.handleClientRequest(clientName, clientNumber);

        assertEquals(expectedResponse, actualResponse, "Response for a number greater than 100 should be a shutdown message.");
    }
}
```

Listing 7: SocketServerTest.java - JUnit Tests