Department of
Computer Science and Engineering

University of Dhaka

Title: Implementation of Client-Server Communication using Socket Programming

---

CSE 3111: Computer Networking Lab

Batch: 28/3rd Year 1st Semester 2024

Course Instructors

Dr. Ismat Rahman (IsR)
Mr. Jargis Ahmed (JA)
Mr. Palash Roy (PR)

1 Objective(s)

- To gather basic knowledge of socket programming using threading
- To learn about step-by-step implementation where a single client communicates entirely with the server as a fully functional chat application.

2 Problem analysis

A socket is mainly the door between the application process and the end-to-end transport protocol. Java Socket programming can be connection-oriented or connectionless. The main problem of the simple two-way socket programming is that it can not handle multiple client requests at the same time. A server can only provide service to the client that comes first to connect with the server. The other clients can not connect with that server. To resolve this problem, the server opens different threads for each client, and every client communicates with the server using that thread.

Another problem of simple two-way socket programming is that if a client can send more than one message one after another without reading any message from the server side, then the server receives only the first message from the client side. However, the other messages that are sent with the first message can not be received. After sending one message from the server side to client, then server receive the messages those are previously sent by the client.

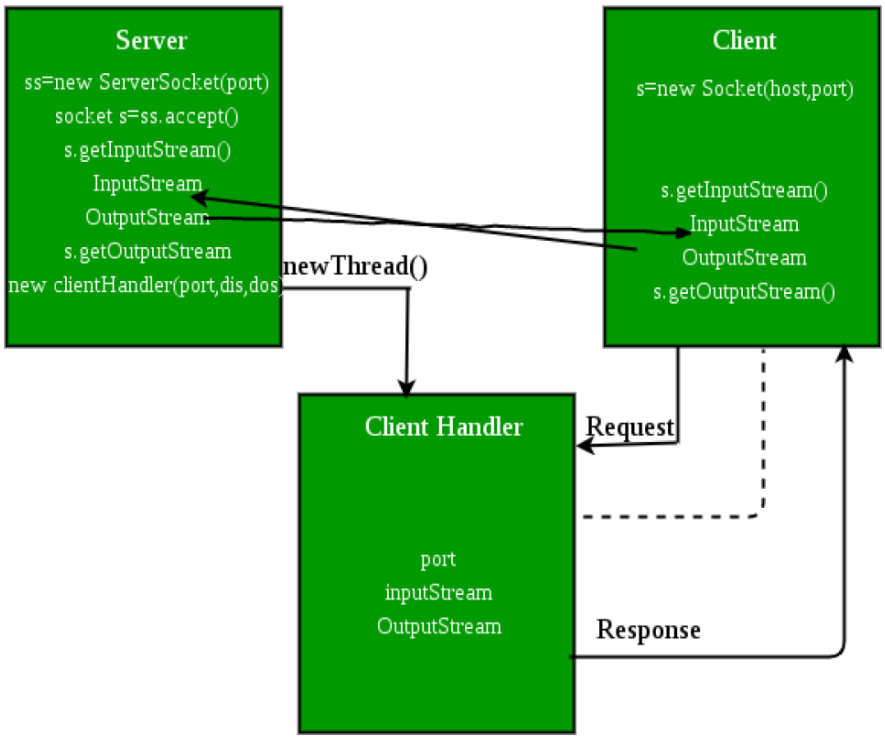

Figure 1: Flow chart

3 Procedure

In the multi-threaded socket programming, at first client creates a connection with server through serversocket, which is depicted in Fig. 1. Then the Server creates a different thread, namely ClientHandler, for each client. In the ClientHandler class, the server passes communication port, the inputStream, and the outputStream as parameters. Then the Client conducts all types of communications with the server through the ClientHandler class. The detailed step-by-step procedures are discussed as follows.

3.1 Server Side Programming

On the server side, we need to create two classes. One is the server class, and another is the clientHandler class.

3.1.1 Server class

The steps involved on the server side are similar to the last lab, with a slight change to create the thread object after obtaining the streams and port number.

- **Establishing the Connection:** Server socket object is initialized, and inside a loop, a socket object continuously accepts incoming connections.
- **Obtaining the Streams:** The inputstream object and outputstream object are extracted from the current requests' socket object.
- **Creating a handler object:** After obtaining the streams and port number, a new clientHandler object (the above class) is created with these parameters.
- **Invoking the start() method:** The start() method is invoked on this newly created thread object.

3.1.2 ClientHandler class:

As we will be using separate threads for each request, let's understand the working and implementation of the ClientHandler class, extending Thread. An object of this class will be instantiated each time a request comes.

- First of all, this class extends Thread so that its objects assume all properties of Threads.
- Secondly, the constructor of this class takes three parameters, which can uniquely identify any incoming request, i.e., a **Socket**, a **DataInputStream** to read from, and a **DataOutputStream** to write to. Whenever we receive any request from the client, the server extracts its port number, the DataInputStream object, and the DataOutputStream object, and creates a new thread object of this class and invokes the start() method on it.
  Note: Every request will always have a triplet of the socket, input stream, and output stream. This ensures that each object of this class writes on one specific stream rather than on multiple streams.
- Inside the **run()** method of this class, it performs necessary operations.

3.2 Client Side Programming

Client-side programming is similar to general socket programming with the following steps-

- **Establish a Socket Connection**
- **Communication**
- **Closing the Connection**

The detailed algorithms of the server-side and client-side connections are given in Algorithm 1 and 2.

Algorithm 1: Algorithm of Server Side Socket Connection

1: Create a ServerSocket object, namely handshaking socket, which takes the port number as input
2: Create a plain Socket, namely a communication socket object that accepts client requests
3: Create two objects of the DataOutputStream and DataInputStream classes, which are used for sending and reading data, respectively.
4: Create an object for ClientHandler Thread class and pass the communication socket, the object of DataOutputStream and DataInputStream classes as the parameters
5: Start the thread class by calling the start() method.
6: In the ClientHandler Thread class, create a constructor of that class.
7: Do the necessary communication until the client sends "Exit"
8: Close the connection

Algorithm 2: Algorithm of Client Side Socket Connection

1: Create a Socket object which takes IP address and port number as input.
2: Create two objects of the DataOutputStream and DataInputStream classes, which are used for sending and reading data, respectively.
3: Client can send the data to the server side using the writeUTF() function, and Client can read any data using the readUTF() function
4: Client can continue its communication with the server until the client sends "Exit"
5: Close the connection

4 Discussion & Conclusion

Based on the focused objective(s) to understand multi-threaded socket programming, this task helped us to learn about about step by step implementation where a single client communicates entirely with the server as a fully functional chat application. The additional lab exercise of multi-threaded socket programming will help us to be confident in the fulfillment of the objective(s) and guide us to implement some real-life problems using multi-threaded socket programming, where multiple clients can communicate with the servers at the same time.

5 Lab Task (Please implement yourself and show the output to the instructor)

Using the above concept, design and implement a non-idempotent operation using exactly-once semantics that can handle the failure of request messages, failure of response messages, and process execution failures.

Design and describe an application-level protocol to be used between an automatic teller machine (ATM) and a bank's centralized server. Your protocol should allow the verification of the user's card and password, the verification of the account balance (which is maintained on the central computer), and the withdrawal of the account (that is, the money paid to the user).

Your protocol entities should be able to handle the all-too-common cases in which there is not enough money in the account to cover the withdrawal. Specify your protocol by listing the messages exchanged and the action taken by the ATM or the centralized bank computer on the transmission and receipt of the messages. Sketch the operation of your protocol for the case of a simple withdrawal with no errors.

What is an exactly-once semantics?
As its name suggests, exactly-once semantics means that each message is delivered precisely once. The message can neither be lost nor delivered twice (or more times). Exactly-once is by far the most dependable message delivery guarantee.

What is an idempotent operation?
In computing, an idempotent operation is one that has no additional effect if it is called more than once with the same input parameters. For example, removing an item from a set can be considered an idempotent operation on the set.

5.1 Problem analysis

Some message types that can be considered between the ATM Client and the Bank are as follows.

Table 1: ATM Server and Bank Communication Message Types

Message Type	Sender	Receiver	Description
`AUTH`	ATM	Bank	Sends card number and PIN
`AUTH_OK` / `AUTH_FAIL`	Bank	ATM	Authentication result
`BALANCE_REQ`	ATM	Bank	Request account balance
`BALANCE_RES:<amount>`	Bank	ATM	Response with current balance
`WITHDRAW:<amount>`	ATM	Bank	Request money withdrawal
`WITHDRAW_OK` / `INSUFFICIENT_FUNDS`	Bank	ATM	Withdrawal success or failure
`ACK`	Either	Other	Acknowledge receipt of message

You can assume following **Protocol Flow (Normal Withdrawal):**

1. ATM sends AUTH:<card_no>:<pin>
2. Bank replies with AUTH_OK
3. ATM sends WITHDRAW:<amount>
4. Bank verifies balance: a) If sufficient, deducts and replies WITHDRAW_OK b) Else, replies INSUFFICIENT_FUNDS
5. ATM sends ACK after receiving response

To ensure non-idempotent operation (e.g., withdrawal) is executed once only, even if messages are lost or retried, you can use the following techniques:

- Use unique transaction IDs for each operation
- Server keeps a log of completed transaction IDs
- On receiving a duplicate request, respond with the cached response
- Require ACK from ATM after response is received
- Server retries sending response until ACK is received

6 Lab Exercise (Submit as a report)

In this lab report, you have to create a fully functional chat application where multiple clients can be connected with a single server. The client will send one sentence or multiple sentences to the server at the same time, and the server will reply to each client separately. The client sends messages as many times as he wishes. However, your program should have the termination conditions for both the client and server sides. You can assume the termination condition as your own choice. However, mention it properly in the lab report.

7 Policy

Copying from the Internet, classmates, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.