Rachelle Abdel Massih - 201800684

Sara Oud – 201702676

COE431 - Computer Networks

Project #1: TCP Sockets in Java

Spring 2021

a. "TCPClient" can behave badly in two different ways:

Attack 1:

- One possible way to disrupt the TCP server is to surround **outToServer.writeBytes(sentence + '\n');** (in TCPClient class) with a while loop that runs constantly. This way the client will keep on sending its message to the server, and the server will keep on accepting the message, without returning an output. And since the server is built to service a single client at a time, no other client will be able to contact that TCP server.

- Solution: a possible solution would be to allow multiple client connections to the TCP server, and this is done through threading [1].

```
while (true) {

    System.out.println("Client is running, enter some text:");
    // Read user's input and store it in a String
    sentence = inFromUser.readLine();
    // Send the user's input to the server
    outToServer.writeBytes( s: sentence + '\n');

    if (sentence.equals("Exit")) { // If user inputs "Exit", close the connection
        System.out.println("Closing this connection..." + clientSocket);
        clientSocket.close();
        System.out.println("Connection closed.");
        break;
    }
        // Read line from server
        modifiedSentence = inFromServer.readLine();
        // Display the server's output to the user
        System.out.println("FROM SERVER: " + modifiedSentence);

}
```

The above code fragment shows the modifications done to "TCPClient" class: The server always reads a new input from the client and outputs the modified sentence upon entering a carriage return. As an additional feature, we implemented multiple requests per client, where the latter is only able to close the connection/stop requesting a service whenever they input "Exit".

The below code fragment shows the modifications done to "TCPServer" class: The server is always on and waits for any requests. Once the server accepts a client's request, it will create a thread for that server (instantiates a ClientHandler object). ClientHandler takes as parameters the client socket, client input, and output to client, respectively. This ClientHandler object, then, begins executing.

```
ClientHandler thread = null;

try {
    while (true) {

        // Wait on welcoming socket for contact by client, creates a new socket
        connectionSocket = welcomeSocket.accept();
        System.out.println("Connected to client.");

        // Create input stream attached to socket
        BufferedReader inFromClient = new BufferedReader(
                new InputStreamReader(connectionSocket.getInputStream()));

        // Create output stream attached to socket
        DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

        System.out.println("Assigning new thread for this client");

        // Assign a thread for the new connected client
        thread = new ClientHandler(connectionSocket, inFromClient, outToClient);
        // Begin executing the thread associated with the connected client
        thread.start();

    }// Loop back and wait for another client connection

}catch (Exception e){
    // close the connection whenever the client requests
    connectionSocket.close();
}
```

The ClientHandler class (shown below), through its run() method , is responsible for reading the client's input and modifying it to upper case before sending it back to the client. This process can execute constantly (interactive requests) until the client inputs "Exit".

```
public void run(){
    String clientSentence; // Client input
    String capitalizedSentence; // Server output

        while (true) {
            try {
                clientSentence = inFromClient.readLine(); // Read line from input

                if (clientSentence.equals("Exit")) { // If user inputs "Exit", close this particular client's connection
                    System.out.println("Client " + this.connectionSocket + " sends exit...");
                    System.out.println("Closing this connection...");
                    this.connectionSocket.close();
                    System.out.println("Connection closed.");
                    break;
                }

                    // Capitalize client's message
                    capitalizedSentence = clientSentence.toUpperCase() + '\n';
                    outToClient.writeBytes(capitalizedSentence); // Send the modified sentence to the client
```

This approach allows the execution of multiple instances (infinite) of "TCPClient" class, further illustrating the concept of threading.

Attack 2:

Another possible attack a malicious client could do is to overflood the server buffer with an input message from its side only. Therefore, the TCPServer would not be available for other clients. One way to flood the server buffer is to keep on appending the user input string to itself until its size is greater than the 64 kilobytes buffer size. If this is not handled at the server side, the server will crash and will not be able to service other clients. The first image below shows the modifications that were carried out at the client side to carry out this attack. Therefore, a possible solution would be to close the connection of the Bad Client who is flooding the servers. This can be done in the ClientHandler Class by comparing the size of the client message that is read from the input stream ("**ReadFromClient**"), and the size of the server buffer ("**ByteBuffer**"). If the ByteBuffer has a

length less than or equal to that of ReadFromClient, then the client is trying to flood the server, and its connection will be closed. This is shown in the second image below. Note that the "Exit" client input still works if the client initially inputs it.

```
 //modified sentence will be null when connection is closed due to flooding the buffer.
while (true && modifiedSentence != null) {

     System.out.println("Client is running, enter some text:");

     StringBuilder sb = new StringBuilder();
     sentence = inFromUser.readLine();

     if (sentence.equals("Exit")) {
         System.out.println("Closing this connection..." + clientSocket);
         clientSocket.close();
         System.out.println("Connection closed.");
         break;
     }

     // Bad Client will flood the server's buffer by sending more Bytes than the buffer can carry.
     // Bad Client will keep on appending the same input until the buffer size is exceeded.

     while (sb.length() < 65536) {       //Buffer Size is 64KB.
         sb.append(sentence);
     }

     outToServer.writeBytes(sb.toString() + '\n');
     System.out.println("Length of bad client message: " + outToServer.size());
```

```
while (true) {
    try {
        clientSentence = inFromClient.readLine(); // Read line from input

        //byteBuffer array of chars to represent maximum server buffer capacity
        char[] byteBuffer = new char[(byte)64000];

        int ByteBufferLength = byteBuffer.length;
        int ReadFromClient = inFromClient.readLine().length();

       //Compare Buffer Length to Bad Client Input Length
       //If the Client flooded the Buffer, close client-specific connection socket.
        if (ByteBufferLength <= ReadFromClient) {
            System.out.println("Buffer is flooded. Connection Closed. ");
            this.connectionSocket.close();
            break;
        }
            // Capitalize client's message
            capitalizedSentence = clientSentence.toUpperCase() + '\n';
            outToClient.writeBytes(capitalizedSentence); // Send the modified sentence to the client

    } catch (IOException e) {
        e.printStackTrace();
    }
```

b.i. If we were to run "TCPClient" class before running "TCPServer" class, a *connection refused* error will result. This is because TCP is connection-oriented and therefore the client needs to contact the server, which will accept the connection, before allowing another byte-stream connection to be set between it and the client.

ii. If we were to run "UDPClient" class before running "UDPServer" class, no error will result and both classes will execute normally. This is because UDP requires no connection between server and client. Server IP address and port number are explicitly sent/received with the data to be transferred between client and server.

References:

1. Introducing threads in socket programming in Java. (2018, February 08). Retrieved March 26, 2021, from https://www.geeksforgeeks.org/introducing-threads-socket-programming-java/