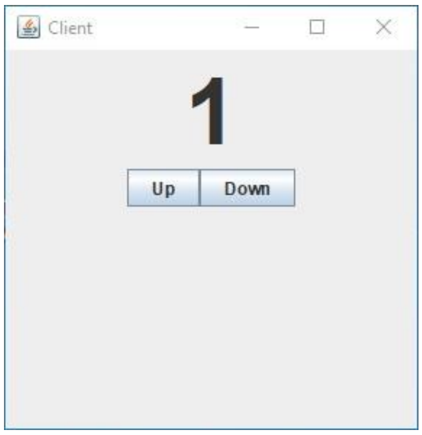
COMP2396B   Tutorial 9

Java Web Socket server/clients communication and Multithreading.

In this tutorial, we are going to create a system that supports one server and multiple clients. This system contains an integer counter in the server and allows clients to increment and decrement the counter. When the counter is updated by one client, the value displayed on all client programs will be updated accordingly.

User-Interface

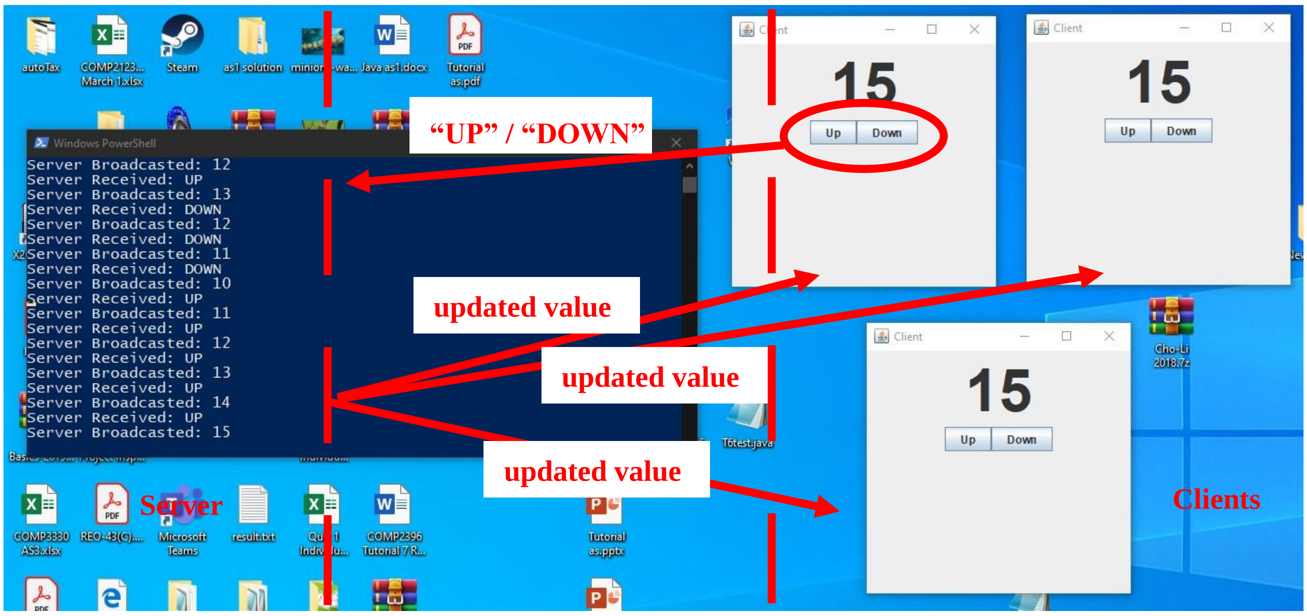
This is how the client program looks like. A label displays the value of the counter, which is received from the server. The “Up”/ “Down” buttons are used to send the “UP” command and the “DOWN” command to the server.



Sever/clients communication protocol

When the server received the “UP”/”DOWN” command. The server will increment/decrement the value accordingly and send the updated value to all connected clients.

1. One client sends the “UP” or “DOWN” command to the server.
2. The server updates the counter accordingly.
3. The server sends the updated value to all connected clients.



```
Server Broadcasted: 12
Server Received: UP
Server Broadcasted: 13
Server Received: DOWN
Server Broadcasted: 12
Server Received: DOWN
Server Broadcasted: 11
Server Received: DOWN
Server Broadcasted: 10
Server Received: UP
Server Broadcasted: 11
Server Received: UP
Server Broadcasted: 12
Server Received: UP
Server Broadcasted: 13
Server Received: UP
Server Broadcasted: 14
Server Received: UP
Server Broadcasted: 15
```

Solution

How to Run the sample solution:

1. Compile and Run the server in Terminal. You will see the server's console logs in the Terminal
   - $ javac *.java
   - $ java T9Server
2. Run multiple client programs in Eclipse / Terminal. You will see the client's console logs in the Eclipse console window / Terminal.

Explanation of the server program:

The counter is implemented by a class object. Increment, decrement and get methods are protected by synchronized function to avoid more than one thread from executing the functions simultaneously.

```java
public class SharedNumber {
    private int n;

    public SharedNumber() {
        this.n = 0;
    }

    public synchronized void up() {
        n++;
    }

    public synchronized void down() {
        n--;
    }

     public synchronized int getNumber() {
        return n;
    }
}
```

In the main program, an object of ServerSocket is created and passed to a Server object.

```
    try (var listener = new ServerSocket(58901)) {
        Server myServer = new Server(listener);
        myServer.start();
    }
```

In the constructor of Server, an object of SharedNumber is created and it will be used for the whole life of the server program.

```
    public Server(ServerSocket serverSocket) {
        this.serverSocket = serverSocket;
        this.number = new SharedNumber();
    }
```

After the construction of the server, the start() is called from the main thread. This function keeps looking for new client connections. serverSocket.accept() is a blocking function and will wait for a client connection. When a connection comes, the socket object, associate with that connection, is passed to a new Handler.

```
    while (true) {
        try {
            Socket socket = serverSocket.accept();
            pool.execute(new Handler(socket));
            System.out.println("Connected to client " + clientCount++);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

As the while loop blocks the thread for awaiting new client connections, the handling of a client is implemented in a separated thread. One thread handles one client.

```
    public class Handler implements Runnable {
    }
```

The code for handling a client is simple. It keeps reading from the socket and matches the incoming command to call the up() or down() of the shared number. Lastly, it broadcasts the updated value to all clients.

```java
    while (input.hasNextLine()) {
        var command = input.nextLine();

        System.out.println("Server Received: " + command);

        if (command.startsWith("UP")) {
            number.up();
        } else if (command.startsWith("DOWN")) {
            number.down();
        }

        // broadcast updated number to all clients
        for (PrintWriter writer : writers) {
            writer.println(number.getNumber());
        }

        System.out.println("Server Broadcasted: " + number.getNumber());
    }
```

Explanation of the client program:

The main thread calls the start() of the controller object. This method setup socket connection to the server, setup AcionListeners for the buttons and create a new thread for handling incoming message from the server.

Setups socket:

```
    try {
        this.socket = new Socket("127.0.0.1", 58901);
        this.in = new Scanner(socket.getInputStream());
        this.out = new PrintWriter(socket.getOutputStream(), true);
    }
```

Setups AcionListeners for the “Up” button:

When the button is clicked, PrintWriter will send “UP” command to the server via socket. For the “Down” button, “DOWN” command is sent.

```
    upButtonListener = new ActionListener() {
        public void actionPerformed(ActionEvent actionEvent) {
            out.println("UP");
            System.out.println("Client Sent: UP");
        }
    };
    view.getUpButton().addActionListener(upButtonListener);
```

Creates a thread for handling incoming messages from server.

```
    Thread handler = new ClinetHandler(socket);
    handler.start();
```

The handling of incoming messages is simple. A while loop keeps monitoring the message from socket input stream buffer. Once a message is received, which is the latest value of the number, it is directly set to the label.

The while loop blocks the thread from doing other things. As the client program still need to deal with GUI updates and button actions, as well as sending commands to the server. This while loop cannot be in main thread. That is why the handler is implemented in a separated thread.

```
    while (in.hasNextLine()) {
        var command = in.nextLine();
        System.out.println("Client Received: " + command);
        out.flush();
        view.getResultLabel().setText(command.trim());
    }
```