Faculty of Computers & Informatics

Zagazig University

Topic No. 4:

A peer to peer application

A research project submitted
in partial fulfillment of the requirements for passing
the 2nd semester 2020 evaluation

In

Internet Programming

by

محمد السيد محمد حسن حسن

3277

Supervised by

Dr. Marwa Khashaba

June 2020

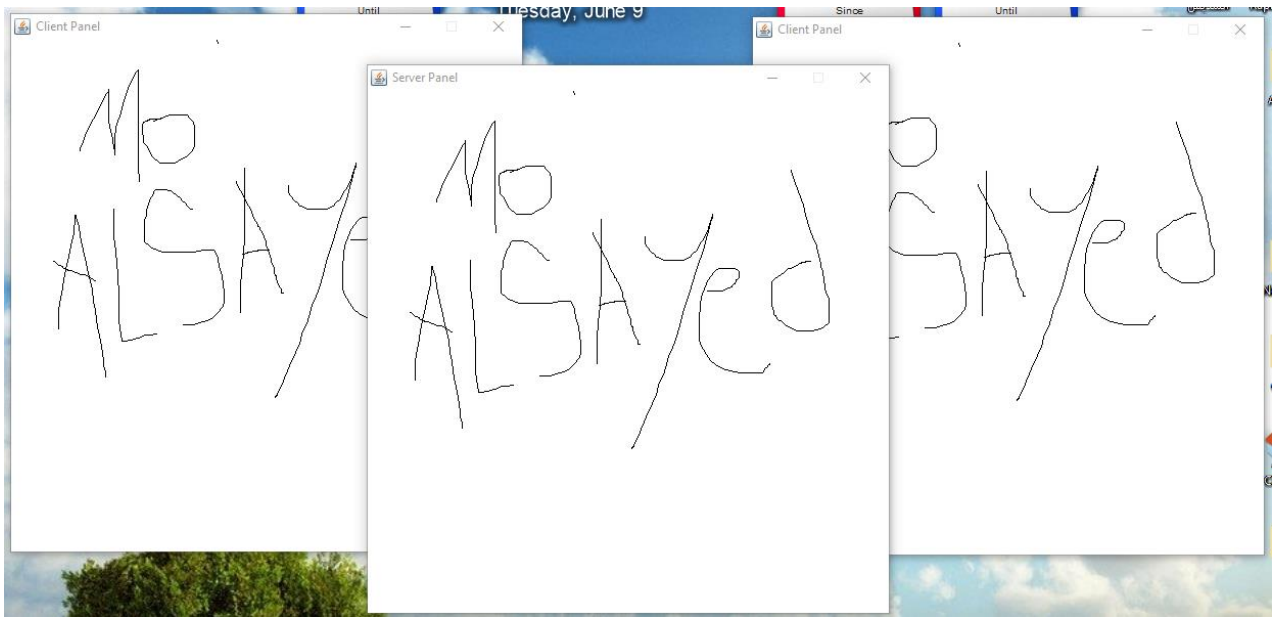
Contents

Introduction

In this research, I will demonstrate how I created a peer-to-peer whiteboard app using java.

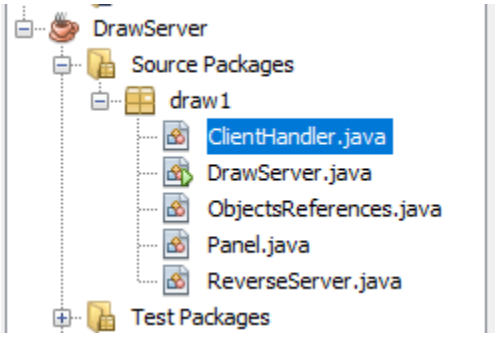
The Application is a White board that user can draw on, his drawing is later sent to a server which in turn send it to all other clients except the sender, so it appears as all client are drawing on the same board.

The main concepts used here are network programming and multithreading.



The server

The server application consists of five classes :



The DrawServer Class

It's the main class in the application, it initializes the panel and docks it into a JFrame, it also assigns it's a reference to the variable panel in the Objects references Class as well as initializing the state variable. We will explain everything in detail as we go through it.

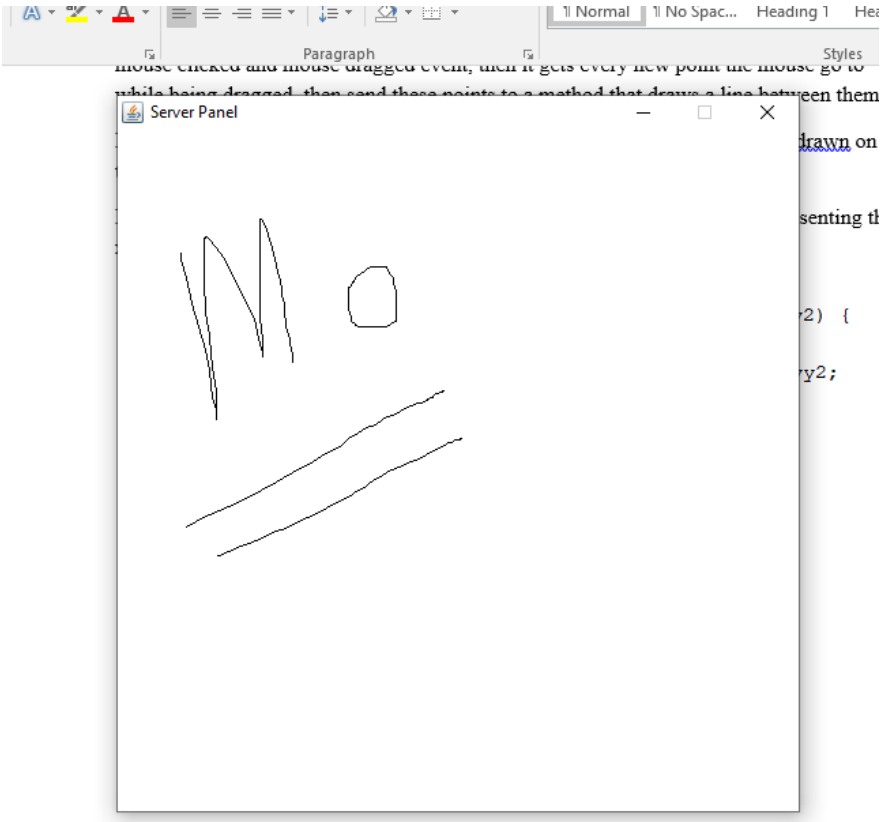
It then starts a reverse server to listen on a particular port.

```
public class DrawServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) throws Exception {

        JFrame frame = new JFrame("Server Panel");
        frame.setDefaultCloseOperation(3);
        frame.setResizable(false);

        Panel panel = new Panel();
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        ObjectsReferences.panel = panel;
        ObjectsReferences.state = new StringBuilder("");

        ReverseServer.startServer(4444);

    }
}
```

The panel Class

It extends JPanel, it's the class responsible for drawing, it works by hooking the mousePressed and mouseDragged events, then it gets every new point the mouse goes to while being dragged, then it sends these points to a method that draws a line between them.

In other words, it works by turning the mouse movement into tiny lines that are drawn on the main panel in a connected way that makes it appear like a single connected line.



drawLine(int,int,int,int);

It exposes the method drawLine(); publicly, which takes four parameters representing the x and y coordinates of the two points to draw a line between.

The state variable represents the state of the drawing panel, in other words, the drawings on it. It is a string builder that carries the coordinates of each tiny line drawn on the panel, this variable is sent to the client during the initial handshake, and the client uses its value to redraw the same drawing that is on the server's panel.

```
public void drawLine(int xx1, int yy1, int xx2, int yy2) {
    initializeGraphics();
    String msg = xx1 + ":" + yy1 + ":" + xx2 + ":" + yy2;
    ObjectsReferences.state.append(msg).append(",");
    g.drawLine(xx1, yy1, xx2, yy2);

}
```

Now there is a problem here regarding the state variable. Although this implementation works fine, with large drawings that have many lines, the state variable becomes too large, which makes it very costly to transfer and split it to get the coordinates.

There are multiple assumptions that can do to solve this problem, the first is instead of sending a list of points and redraw every single line of them, we can send an entire object representing the whole state of the panel of the server. In other words, we will use serialization to serialize and send the graphics object bound to the panel object. However, in java, there is no such thing, actually, every time that you call the paintComponent(); method of the JPanel, the entire panel is redrawn from scratch. The only way to keep the state is to save it in a variable, which we did here.

Another assumption is to capture the JPanel as an image and send it to the client, the client will display this image on its panel, which gives the illusion that we had transferred the state of the servers panel to the client.

The third assumption is to create an algorithm that reduces the redundancy (compress) the state variable. For example, if the mouse is on the position (0,0) and we pressed and dragged it to the position (5,5), there will be 5 points created, and the drawLine(); method will be called five times, instead, we can reduce them to two points, (0,0) and (5,5) which will give the same result.

We improve this solution by compensating for tiny lines that will not make a difference for the human eye by discarding them. There are implementations online for such an algorithm.

However, I decided to stick with the current implementation as the focus of this research is on the network part, and I do not want to waste time on premature optimization.

The objectsReferences Class

Our application needs a way for its classes to communicate with each-others, this is the role of the ObjectsReferences class, which holds references to all created instances in the app such as the panel class, a list of connected clients and the state of the drawing.

In other words, it serves as a channel of communication between the instance-classes of our application.

```
 */
public class ObjectsReferences {

    public static Panel panel;
    public static List<ClientHandler> clients;
    public static StringBuilder state;
}
```

The ReverseServer Class

We need to create a server, that multiple clients can connect to, we need this server to be able to get messages from these clients and send them to the panel object, this panel will only be a single instance in the memory.

We also need this server to be able to send messages back to a single client as well as all clients.

So we will create a ServerSocket, that accepts a socket then assign a new thread for that socket to handle it, and go back to waiting for new connections.

The difference between a normal server and a reverse server is that a reverse server is able to send messages to the client, while a normal server only receives messages.

It consists of two static methods, as there is no need to create multiple instances of this server.

startServer(int);

```
public class ReverseServer {

    public static void startServer(int port) {

        try {
            ServerSocket serverSocket = new ServerSocket(port);
            ObjectsReferences.clients = new ArrayList<>();

            int i = 0;
            while (true) {
                Socket socket = serverSocket.accept();
                ClientHandler client = new ClientHandler(socket, i);
                client.sendToClient(ObjectsReferences.state.toString());
                System.out.println("New client connected " + i);
                i++;
                ObjectsReferences.clients.add(client);
                client.startReceiving();
            }
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }
    }
```

This method creates a server socket that listens for connections coming at a certain port,

Then it initializes the clients list in the objectsReferences Class.

Then it goes in an infinite loop, waiting for new clients to connect.

When a client connects, we accept it's a socket and create a new client handler class, we pass that socket as well as an integer representing the client's id, we also add the new client to the list of clients in the Objects References, so we can communicate with it later.

We then send the state of the drawing to the client, so when the client opens his panel, he will see the drawing that has been drawing before on the server, not an empty panel.

sendToAllClients(String);

we use this method to send a message to all clients connected to this server.

It iterates over the list of the clients connected to the server and sends the message to every one of them.

```
public static void sendToAllClients(String message) {
    for (ClientHandler client : ObjectsReferences.clients) {
        try {
            client.sendToClient(message);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }

    }
}
```

The Client Handler Class

It has a constructor that accepts a socket and an integer representing the Id of the client,

It has three methods, startReceiving(); which receives data from the client, sendToClient(); which sends a message to the client, and sendToAllClientsExceptThis(); which broadcast a message to all clients on the server except the client that's being handled by this class.

```
 * @author MiDo
 */
public class ClientHandler {

    private Socket socket;
    public int ClientId;

    public ClientHandler(Socket socket, int ClientId) {...4 lines }

    public void startReceiving() {...32 lines }

    public void sendToClient(String message) {...16 lines }

    public void sendToAllClientsExceptThis(String message) {...8 lines }
}
```

startRecieving();

this method creates a new thread dedicated only for reading form the client, we get the input stream of the socket and start reading from it using a buffered reader, the client is expected to send us a string containing the four coordinates to draw a line between them, it's formatted like this X1:Y1:X2: Y2 so we split the message with the separator ":" to get each point alone, then we call the drawLine(); method of the panel class and pass these points to it, we use the reference we saved earlier in the Objects references class so that it's the same panel in memory that's appearing on the application.

In other words, the objectsReferences class serves as a channel that other classes use to communicate with each other.

Finally, we send this message to all clients except this client, although sending it to the client again won't make a difference as you will be redrawing on the lines that had already been drawn on the client's panel. but we avoid this for consistency and saving resources.

```
public void startReceiving() {
    new Thread() {
        @Override
        public void run() {
            try {

                while (true) {
                    try {
                        InputStream input = socket.getInputStream();
                        BufferedReader reader = new BufferedReader(new InputStreamReader(input));
                        String clientMessage = reader.readLine();
                        System.out.println("Message from client " + ClientId + " : " + clientMessage);
                        String[] coords = clientMessage.split(":");
                        int xx1 = Integer.parseInt(coords[0]);
                        int yy1 = Integer.parseInt(coords[1]);
                        int xx2 = Integer.parseInt(coords[2]);
                        int yy2 = Integer.parseInt(coords[3]);
                        ObjectsReferences.panel.drawLine(xx1, yy1, xx2, yy2);
                        sendToAllClientsExceptThis(clientMessage);

                    } catch (Exception ex) {
                        System.out.println(ex.getMessage());
                        ex.printStackTrace();
                    }
                }

            } catch (Exception ex) {

            }
        }
    }.start();
```

sendToClient(String);

this method works on its own separate thread, we get the output stream of the socket, assign it to a print writer, then write the message and flush the stream.

```
public void sendToClient(String message) {
    new Thread() {
        @Override
        public void run() {
            try {
                OutputStream output = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(output);
                writer.println(message);
                writer.flush();
            } catch (Exception ex) {
                System.out.println(ex.getMessage());
                ex.printStackTrace();
            }
        }
    }.start();
}
```

There's an important point here, why did we use two threads one dedicated for reading and others dedicated to writing?
our application needs to be able to read and write to the socket separately, now if we used a single thread for both reading and writing, the reading operation will block the execution until a message is received from the server, the same thing will happen on the server too, it will be waiting for a message from the client to continue execution, which results in a deadlock.

Also in our application, the reading and writing operations are completely separated, we don't know when we will be receiving a message or when we will be sending a message, so the reading and writing operation must not block each other, that's why we used these separated threads.

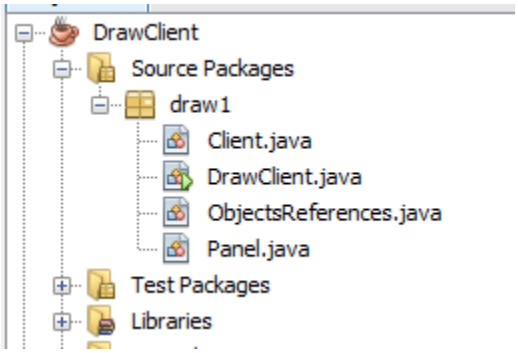
sendToAllClientsExceptThis(String);

This method works by iterating over all clients saved in the clients list in the objects references class, we then check the client id of each client, if it's not equal the client id of this clientHandler, we send the message to this client.

```
public void sendToAllClientsExceptThis(String message) {
    for (ClientHandler client : ObjectsReferences.clients) {
        if (client.ClientId != ClientId) {
            System.out.println("From client " + ClientId + " sending to " + client.ClientId);
            client.sendToClient(message);
        }
    }
}
```

The Client

The client app consists of four classes, which we will explain below.



The DrawClientClass

This is the main class of our application, it starts by creating a new panel and docking it into a JFrame as well as initializing its value on the objectsReferences repository.

Then it connects to a server specified by the IP and port number.

```
*/
public static void main(String args[]) {

    JFrame frame = new JFrame("Client Panel");
    frame.setDefaultCloseOperation(3);
    frame.setResizable(false);
    Panel panel = new Panel();
    frame.add(panel);
    ObjectsReferences.panel = panel;
    frame.pack();
    frame.setVisible(true);
    Client.startReceiving("127.0.0.1", 4444);

}
```

The objectsRefernces Class

This is the same as the class in the server, it serves as a communication channel between instances of classes of the project to communicate with each other.

However, here it only holds a reference to the panel class.

```
 * @author MIDO
 */
public  class ObjectsReferences {
    public static Panel panel;
}
```

The panel class

It the same as the panel class in the server, with one more method called

redrawFromState();

During the initial handshake with the server (our custom handshake not TCP handshake that happens before !), the servers send us a string representing the state of the panel on the server, it's a string consists of the coordinates of all tiny lines drawn on the server's panel, the client will break this string into a list of coordinates that it will subsequently call drawLine(); on all of them, so when the client first connects, it will see the drawing that's on the servers panel before it event connects not an empty panel.

```
public void redrawFromState( String stateStr) {

        if (stateStr == null || stateStr.isEmpty()) {
            return;
        }
        String[] lines = stateStr.split(",");
        for (String line : lines) {
            String[] coords = line.split(":");
            int xx1 = Integer.parseInt(coords[0]);
            int yy1 = Integer.parseInt(coords[1]);
            int xx2 = Integer.parseInt(coords[2]);
            int yy2 = Integer.parseInt(coords[3]);
            drawLine(xx1, yy1, xx2, yy2);
        }

    }
```

The Client Class

This is the class responsible for all the network logic in the application.

It's similar in architecture to the server, two threads, one for reading and one for writing, so no one can block another.

It consists of two static methods, startRecieving() and sendToServer();

```
 * @author MiDo
 */
public class Client {

    static Socket socket;

    public static void startReceiving(String host, int port)

    public static void sendToServer(String message) {...17 li
}
```

startRecieving(String,int);

this method takes in a host and port and creates a new socket instance that connects to the target server, we didn't set the socket initialization in the constructor because we are only going to have one instance of this class at a time, and the logic of receiving inputs from the server is coupled to the logic of connecting to the server, so it makes more sense to have them in a single method. We used a constructor in the server however because the server gets the socket from outside its scope, so we needed a method to assign this socket to a one in our scope, also there will be multiple instances of the client handler class in the memory, one for each client, but here We are the ones creating that socket and we only need a single instance.

After creating the socket, we assign it's a reference to the socket variable so that we can use it again in the sendToServer() method.

We get the input stream, assign it to a buffered reader, and start receiving from the server.

First, we read a message into the variable stateMessage, this is the initial handshake with the server where the server sends us the state of its board, so we can see what has been drawn on the server before we connect to it. Then we call the redrawFromState(); method in the panel object to draw on our panel.

Then we enter into an infinite loop where we keep getting messages from the server and pass them to the drawLine(); method of our panel.

This method works on a separate thread, so it doesn't block when we are trying to write to the server.

```
public static void startReceiving(String host, int port) {
    new Thread() {
        @Override
        public void run() {
            try {
                socket = new Socket(host, port);
                InputStream input = socket.getInputStream();
                BufferedReader reader = new BufferedReader(new InputStreamReader(input));
                String stateMessage = reader.readLine();
                ObjectsReferences.panel.redrawFromState(stateMessage);

                while (true) {
                    String serverMessage = reader.readLine();
                    //System.out.println("Message From server : " + serverMessage);
                    String[] coords = serverMessage.split(":");
                    int xx1 = Integer.parseInt(coords[0]);
                    int yy1 = Integer.parseInt(coords[1]);
                    int xx2 = Integer.parseInt(coords[2]);
                    int yy2 = Integer.parseInt(coords[3]);
                    ObjectsReferences.panel.drawLine(xx1, yy1, xx2, yy2);
                }
            } catch (Exception ex) {
                System.out.println(ex.getMessage());
                ex.printStackTrace();
            }
        }
    }.start();
```

sendToServer(String);

This method takes a string as an input, it then gets the output stream of the server, assigns it to a PrintWriter object then write it and close the stream.

It works on its separate thread, although it's not necessary, I want it to be consistent with the server logic.

```
public static void sendToServer(String message) {
    new Thread() {
        @Override
        public void run() {
            OutputStream output;
            try {
                output = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(output);
                writer.println(message);
                writer.flush();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
                ex.printStackTrace();
            }
        }
    }.start();
}
```

Conclusion

In this research, I explained how to create a peer-to-peer whiteboard application where multiple clients can draw on a shared whiteboard at different points of time.

The application consists of a server and a client, the server works by listing on a certain port, accept a client, save it on a list to be able to access it later, then assign two threads to handle this client, one for reading and one for writing.

All drawings on the server are saved in a variable called state, which is sent to every new client, the client will then uses it to redraw the same drawing on the server.

When a client sends a message, the server parses it, extract the coordinates, then draw a line based on these coordinates, it will then broadcast this message to all other connected clients so they can draw the same shape as well.

The panel class works by hocking the mouse pressed and the mouse dragged events, it will then get each point on the screen that the mouse passes on while being dragged and draw a line between them.

The client works by connecting to a server on a certain host and port, then get the state of the server panel and draw it on its panel. It

References

Internet & World Wide Web: How to Program

An Introduction to Network Programming with Java

Concurrency and multithreading in java:

https://docs.oracle.com/javase/tutorial/essential/concurrency/

All about sockets:

https://docs.oracle.com/javase/tutorial/networking/sockets/index.html

ServerSocket:

https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html

Socket:

https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html