**PROJECT REPORT**

**(Project Name: ChatBoxx (A Messenger))**

Submitted By:

Aman Sharma(101412003)

Anshu Trehan(101412005)

Arushi Singla(101412009)

****

**Department of Computer Science and Engineering  
 THAPAR UNIVERSITY, PATIALA**

**CONTENTS**

1. COMPANY PROFILE.....................................................................6
2. INTRODUCTION…………………………………………………....7
3. BACKGROUND……………………………………………………..9
4. CHAPTER-1 (SOCKET PROGRAMMING)………………………10
5. CHAPTER-2 (CREATING GUI)…………………………………...16
6. CHAPTER-3 (EVENT HANDLING)……………………………….20
7. TESTING………………………………………………………….....23
8. LIMITATIONS……………………………………………………….24
9. CONCLUSION AND FUTURE WORK……………………………25
10. REFERENCES……….……………………………………………26

**List of Figures:**

1. TCP-UDP sockets ……………………………………………….…11
2. Screenshot of socket programming ………………………………...15
3. GUI of Project………………………………………………………19

**INTRODUCTION**

The Project (“ChatBoxx (A Messenger)”) is clear from the name itself that it is a messenger which is used to send and receive messages. Basically, it is a live chat messenger. Two or more persons can chat on this messenger. One will be connected as a Server and rest will be connected as Clients.

Both Client and Server are set up on a Port No. and whenever the Client needs to get connected, it will need the IP Address of the Server. (IP Address can be find out after the PC is connected to Internet Connection).  
Once they are connected to one another, they can have live chat.  
If more than one Client is connected to server then the feature of Group Chat has been also provided. After connection they can “Reset” their connections.

Both Server and Client have been provided with the feature of “Sending Files”. With this feature they can share files (like audio/video, images, ppts, office files, docx, PDFs, etc.) of size upto 100MB with each other in no time. Everyone using this messenger has the option of “Save”, by which they can save their chats in their PCs with suitable extension. Option “Clear chat” is provided to clear the chat history.

This software with these features is built in Java and is totally Java based. This runs everywhere where suitable Java libraries are found such as Windows, Linux.

Message and file transfer between the Server and Clients follow Socket Programming in Java.

Followed by the proper testing performed on multiple platforms, PCs, Laptops, different Windows, this Software is ready to perform Group Chat, sending image files, saving chats, clear text, all the controls set on buttons, etc. with respect to time as a constraint.

The main thing to be mentioned is that this software runs with the Internet Connection. If both the Server and the Client are under one router or the same Internet Connection then they will need dynamic IP that is usually provided to each machine having Internet Connection. But if Client is not under the same network and is in any other area having different network, then server needs “Static IP” to connect Clients with it.

Hence, this software works on both LAN and WAN.  
With time, there is a scope of improvement in the features of software.

**BACKGROUND**

The Project (“**ChatBoxx (A Messenger)”**) is basically an application or software that is built in Java on the purpose of having live chat with one or many persons easily just by filling IP Address to connect with one another in the presence of Internet Connection.

The primary focus of this project is to develop software that is easy to use and that too without many resources. Anyone using this software will just have the need to know the IP Address of Server and an Internet Connection and then they can communicate with one another in no time. At work or in company, this software can help to share conclusions, multiple files, documents etc.

This era is of social media. **ChatBoxx** combines instant messaging features with social networking, so one can stay in touch with the important people in their life. Additionally, with this messenger one can share images/audios/videos/PDFs/office files etc. of size upto 100MB. And they can save/clear their chats as a memory if they want to.

This Messenger is very easy to use. Just need to connect one person as a Server and rest will be the clients. They are already set with a definite Port No. (can change too but server and clients needs to be at the same port) and Clients just need to fill the IP Address of Server to get connected whenever demanded. A separate area of selecting files is provided along with the button to send them. Everything is just one click away (or you can press “Enter” to send messages) and will be able to contact each other regardless of the fact that where is the client located.

After creating the connections between Server and Clients with their desired names, messenger is ready to receive and send instant messages. You can reply your messages in no time. Whenever you want to share image files or any other files for any purpose you can! For any kind of emergency, memory, a specific feature of save/clear chat is provided to save your chat.

While having the group chat, if any of the persons has left the chat then others will be notified and will cause no harm to their chat and they can pursue it as they were doing already. The technical work undertaken this project is described in the next section of the report.

**CHAPTER -1**

**(Socket Programming)**

Java Socket Programming is used for communication between the applications running on differentJRE.  
Java Socket Programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know this information:

1. IP Address of server and,
2. Port Number.

**What is a Socket Class?**

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

**What Is a Socket?**

A socket is an abstraction through which an application may send and receive data, in much the same way as an open ﬁle handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a diﬀerent machine and vice versa. Diﬀerent types of sockets correspond to diﬀerent underlying protocol suites and diﬀerent stacks of protocols within a suite. The main types of sockets in TCP/IP today are stream sockets and datagram sockets. Stream sockets use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service. A TCP/IP stream socket represents one end of a TCP connection. Datagram sockets use UDP (again, with IP underneath) and thus provide a best-eﬀort datagram service that applications can use to send individual messages up to about 65,500 bytes in length. Stream and datagram sockets are also supported by other protocol suites. A TCP/IP socket is uniquely identiﬁed by an Internet address, an end-to-end protocol (TCP or UDP), and a port number.

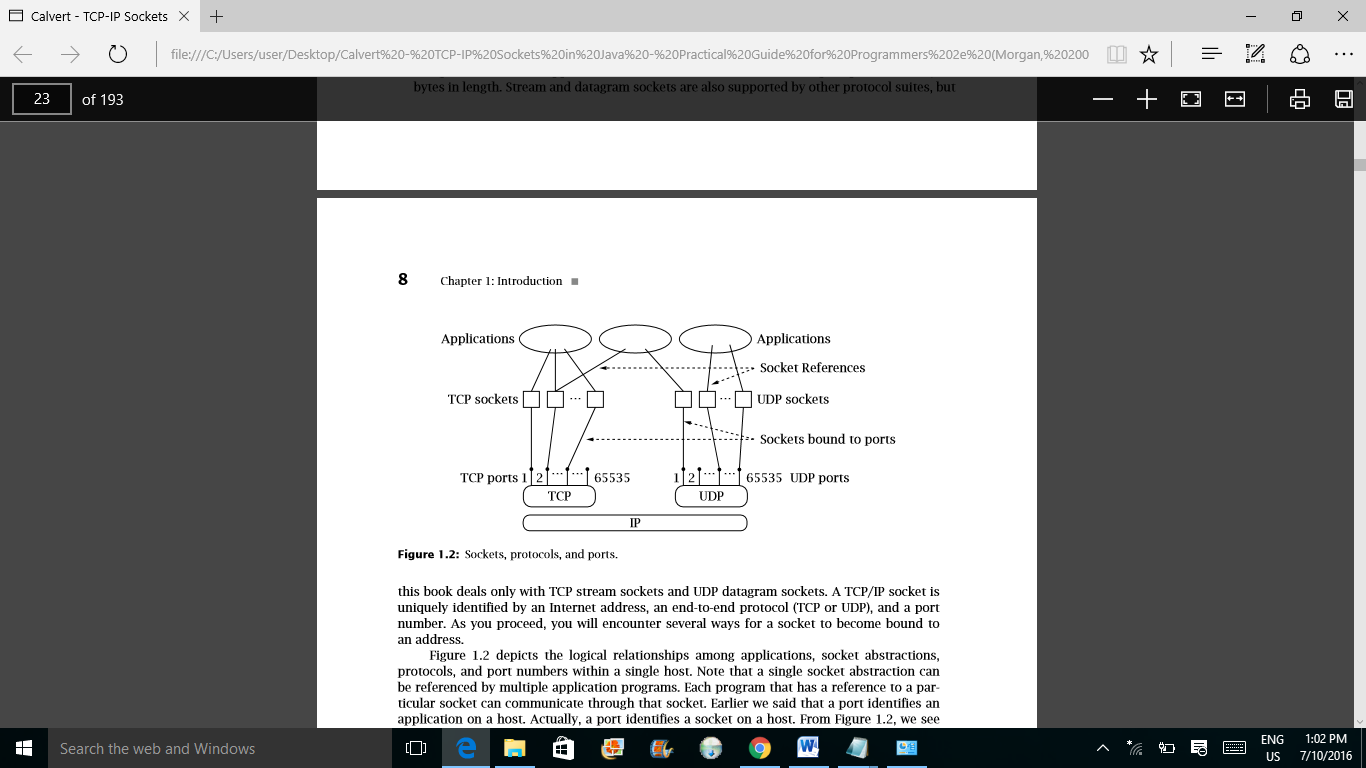
****

Figure-1

Figure-1 above depicts the logical relationships among applications, socket abstractions, protocols, and port numbers within a single host. Note that a single socket abstraction can be referenced by multiple application programs. Each program that has a reference to a particular socket can communicate through that socket. Earlier we said that a port identiﬁes an application on a host. Actually, a port identiﬁes a socket on a host. From Figure, we see that multiple programs on a host can access the same socket. In practice, separate programs that access the same socket would usually belong to the same application (e.g., multiple copies of a Web server program), although in principle they could belong to diﬀerent applications.

**What are TCP Sockets?**

Java provides two classes for TCP: Socket and ServerSocket. An instance of Socket represents one end of a TCP connection. A TCP connection is an abstract two-way channel whose ends are each identiﬁed by an IP address and port number. Before being used for communication, a TCP connection must go through a setup phase, which starts with the client’s TCP sending a connection request to the server’s TCP. An instance of ServerSocket listens for TCP connection requests and creates a new Socket instance to handle each incoming connection. Thus, servers handle both ServerSocket and Socket instances, while clients use only Socket. We begin by examining an example of a simple client.

**TCP Client**: The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:

1. Construct an instance of Socket: The constructor establishes a TCP connection to the speciﬁed remote host and port.

2. Communicate using the socket’s I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream.

3. Close the connection using the close() method of Socket.

**TCP Server**: We now turn our attention to constructing a server. The server’s job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server goes through two steps:

1. Construct a ServerSocket instance, specifying the local port. This socket listens for incoming connections to the speciﬁed port.

2. Repeatedly:

a. Call the accept() method of ServerSocket to get the next incoming client connection. Upon establishment of a new client connection, an instance of Socket for the new connection is created and returned by accept().

b. Communicate with the client using the returned Socket’s InputStream and OutputStream.

c. When ﬁnished, close the new client socket connection using the close() method of Socket.

**Input and Output Streams**: The basic I/O paradigm for TCP sockets in Java is the stream abstraction. A stream is simply an ordered sequence of bytes. Java input streams support reading bytes, and output streams support writing bytes. In our TCP client and server, each Socket instance holds an InputStream and an OutputStream instance. When we write to the output stream of a Socket, the bytes can (eventually) be read from the input stream of the Socket at the other end of the connection. OutputStream is the abstract superclass of all output streams in Java. Using an OutputStream, we can write bytes to, ﬂush, and close the output stream. InputStream is the abstract superclass of all input streams. Using an InputStream, we can read bytes from and close the input stream.

In case of my project the basic programs are:

**Server Side:**

import java.net.\*;

import java.io.\*;

class sr {

public static void main(String args[]) throws Exception {

ServerSocket ss = new ServerSocket(1043);

Socket s=ss.accept();

final DataInputStream din = new DataInputStream(s.getInputStream());

DataOutputStream dout = new DataOutputStream(s.getOutputStream());

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String str;

str="";

new Thread(){

public void run()

{ String str2="";

while(true)

{

try{str2=din.readUTF();}catch(IOException e){}

System.out.println("\nClient says:"+str2);

}}}.start();

while (!str.equals("stop")) {

str=br.readLine();

dout.writeUTF(str);

dout.flush();

}

dout.close();

s.close();ss.close();}}

**Client Side:**

import java.net.\*;

import java.io.\*;

class cl {

public static void main(String args[]) throws Exception {

Socket s = new Socket("localhost", 1043);

final DataInputStream din = new DataInputStream(s.getInputStream());

DataOutputStream dout = new DataOutputStream(s.getOutputStream());

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String str;

str="";

new Thread(){

public void run()

{ String str2="";

while(true)

{

try{str2=din.readUTF();}catch(IOException e){}

System.out.println("\nserver says:"+str2);

}}}.start();

while (!str.equals("stop")) {

str=br.readLine();

dout.writeUTF(str);

dout.flush();

}

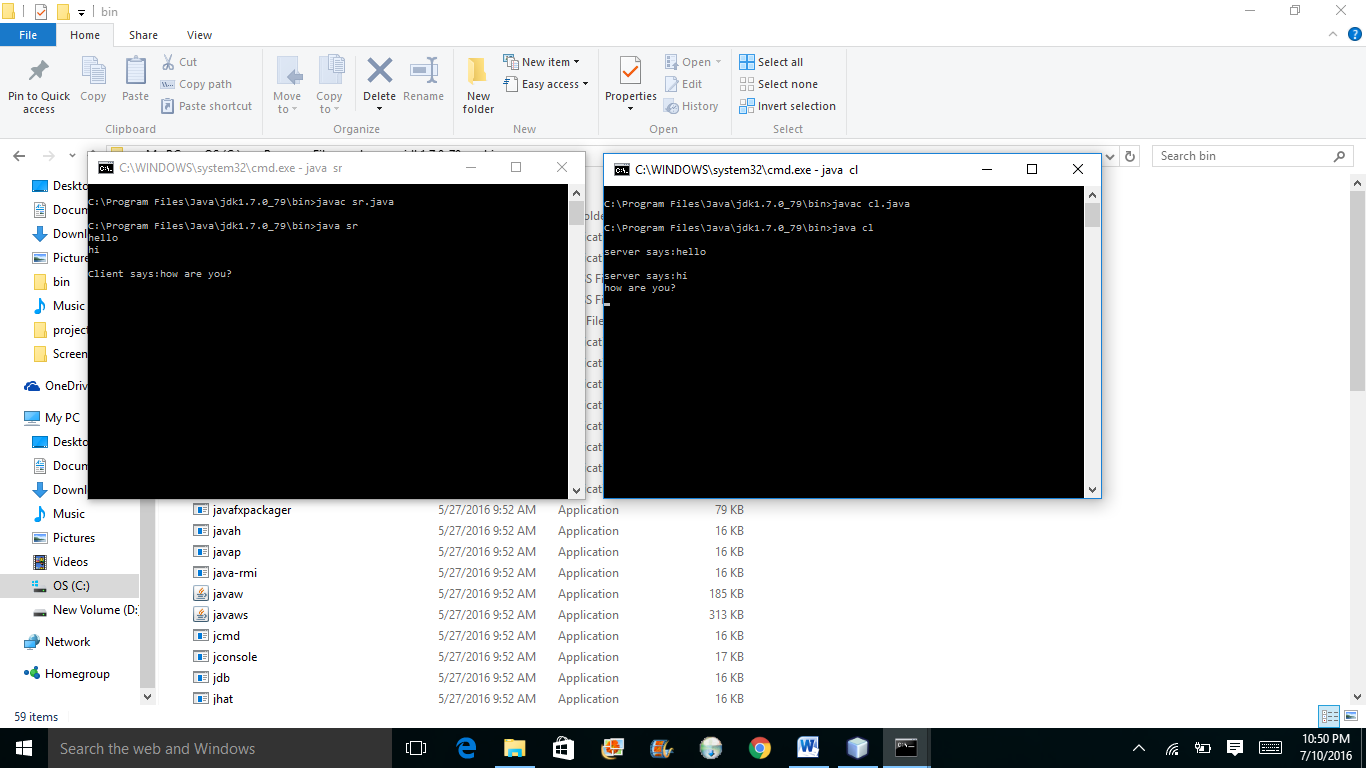
dout.close();

s.close();

}}

Figure-2 below is the screenshot of basic socket program.

Figure-2



**CHAPTER-2  
(Creating GUI)**

For GUI (Graphical User Interface) building in Netbeans, we need to follow some steps:

* [Step 1: Creating a Project](https://netbeans.org/kb/docs/java/gui-functionality.html#Exercise_1)
* [Step 2: Building the Front End](https://netbeans.org/kb/docs/java/gui-functionality.html#Exercise_2)
* [Step 3: Adding Functionality](https://netbeans.org/kb/docs/java/gui-functionality.html#Exercise_3)
* [Step 4: Running the Program](https://netbeans.org/kb/docs/java/gui-functionality.html#Exercise_4)

**Step-1:**

The first step is to create an IDE project for the application that we are going to develop. We will name our project.

1. Choose File > New Project. Alternatively, you can click the New Project icon in the IDE toolbar.
2. In the Categories pane, select the Java node. In the Projects pane, choose Java Application. Click Next.
3. Type “the name of your project=XYZ(or any)” in the Project Name field and specify a path, for example, in your home directory, as the project location.
4. (Optional) Select the Use Dedicated Folder for Storing Libraries checkbox and specify the location for the libraries folder. See [Sharing a Library with Other Users](http://www.oracle.com/pls/topic/lookup?ctx=nb8000&id=NBDAG455) in*Developing Applications with NetBeans IDE* for more information.
5. Deselect the Create Main Class checkbox if it is selected.
6. Click Finish.

**Step-2:**

To proceed with building our interface, we need to create a Java container within which we will place the other required GUI components. In this step we'll create a container using the JFrame component. We will place the container in a new package, which will appear within the Source Packages node.

We need to create a JFrame Container as follows:

1. In the Projects window, right-click the “XYZ” node and choose New > Other.
2. In the New File dialog box, choose the Swing GUI Forms category and the JFrame Form file type. Click Next.
3. Enter “XYZ” UI as the class name.
4. Enter my. “XYZ” as the package.
5. Click Finish.

### Adding Components: Making the Front End

Next we will use the Palette to populate our application's front end with a JPanel. Then we will add three JLabels, three JTextFields, and three JButtons.

If you do not see the Palette window in the upper right corner of the IDE, choose Window > Palette.

1. Start by selecting a Panel from the Swing Containers category on Palette and drop it onto the JFrame.
2. While the JPanel is highlighted, go to the Properties window and click the ellipsis (...) button next to Border to choose a border style.
3. In the Border dialog, select TitledBorder from the list, and type in XYZ in the Title field. Click OK to save the changes and exit the dialog.

**Step-3:**

The jTextField1 and jTextField2 boxes will be used for user input andjTextField3 for program output - what we are creating is a very simple calculator. Let's begin.

Making the Exit Button Work

In order to give function to the buttons, we have to assign an event handler to each to respond to events. In our case we want to know when the button is pressed, either by mouse click or via keyboard. So we will use ActionListener responding to ActionEvent.

1. Right click the Exit button. From the pop-up menu choose Events > Action > actionPerformed. Note that the menu contains many more events you can respond to! When you select the actionPerformed event, the IDE will automatically add an ActionListener to the Exit button and generate a handler method for handling the listener's actionPerformed method.
2. The IDE will open up the Source Code window and scroll to where you implement the action you want the button to do when the button is pressed (either by mouse click or via keyboard). Your Source Code window should contain the following lines:
3. private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
4. //TODO add your handling code here:

}

1. We are now going to add code for what we want the Exit Button to do. Replace the TODO line with System.exit(0);. Your finished Exit button code should look like this:
2. private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
3. System.exit(0);

}

**Step-4:**

**To run the program in the IDE:**

1. Choose Run > Run Project (XYZ) (alternatively, press F6).

**Note:** If you get a window informing you that Project XYZdoes not have a main class set, then you should selectmy.XYZ.XYZUI as the main class in the same window and click the OK button.

**To run the program outside of the IDE:**

1. Choose Run > Clean and Build Main Project (Shift-F11) to build the application JAR file.
2. Using your system's file explorer or file manager, navigate to the XYZ/dist directory.

**Note:** The location of the XYZ project directory depends on the path you specified. Double-click the XYZ.jar file.

After a few seconds, the application should start.

**To launch the application from the command line:**

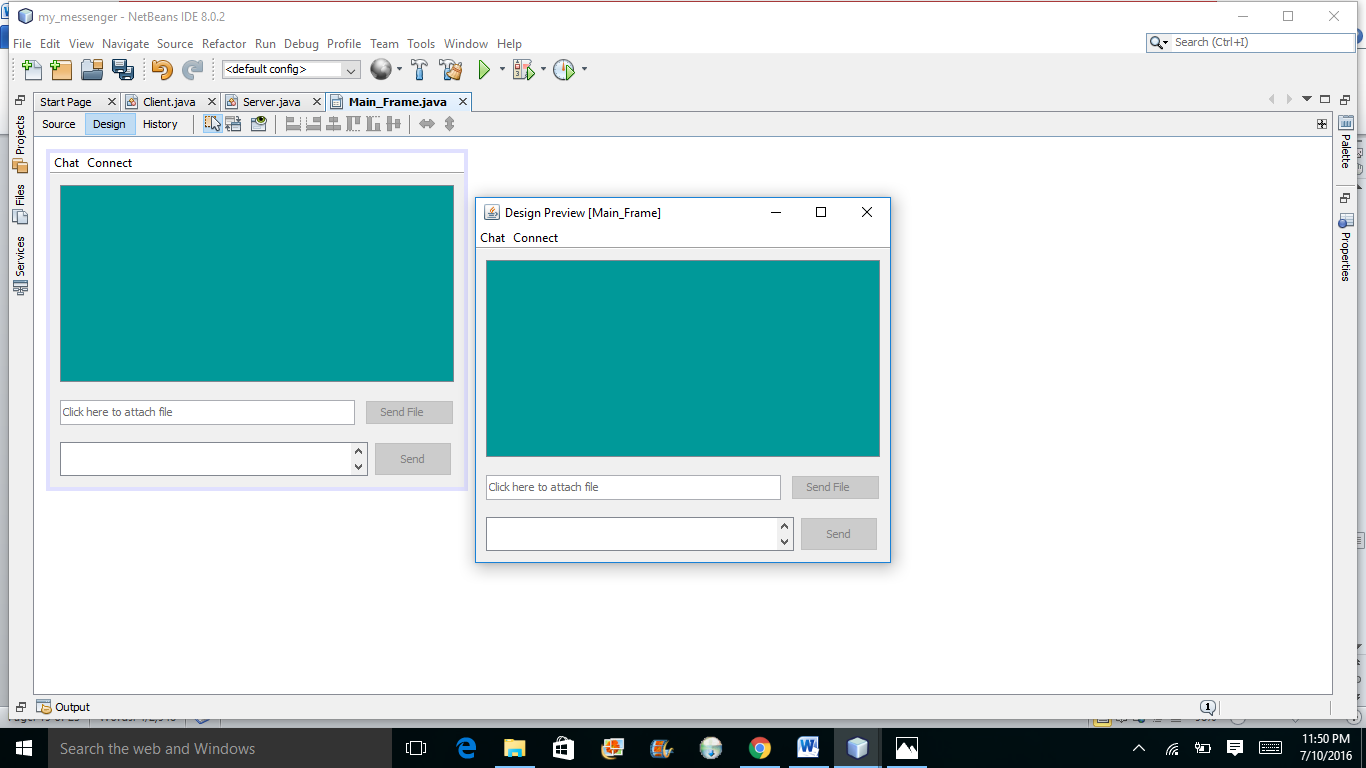
1. On your system, open up a command prompt or terminal window.
2. In the command prompt, change directories to the XYZ/dist directory.
3. At the command line, type the following statement:

java -jar XYZ.jar

**GUI of this Project:**

Figure-3 below shows the GUI of the project.

Figure-3



**Chapter-3**

**(Event Handling)**

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

**Steps to perform Event Handling**

Following steps are required to perform event handling:

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods. For example:

* **Button**
  + public void addActionListener(ActionListener a){}
* **MenuItem**
  + public void addActionListener(ActionListener a){}
* **TextField**
  + public void addActionListener(ActionListener a){}
  + public void addTextListener(TextListener a){}
* **TextArea**
  + public void addTextListener(TextListener a){}
* **Checkbox**
  + public void addItemListener(ItemListener a){}
* **Choice**
  + public void addItemListener(ItemListener a){}
* **List**
  + public void addActionListener(ActionListener a){}
  + public void addItemListener(ItemListener a){}

**Event Handling Codes:**

|  |
| --- |
| We can put the event handling code into one of the following places:   1. Same class 2. Other class 3. Anonymous class |

**How Event Handling Works?**

The IDE can help you find the list of available events your GUI components can handle:

1. Go back to the file XYZUI.java in the Editor. Click the Design tab to see the GUI's layout in the GUI Builder.
2. Right-click any GUI component, and select Events from the pop-up menu. For now, just browse the menu to see what's there, you don't need to select anything.
3. Alternatively, you can select Properties from the Window menu. In the Properties window, click the Events tab. In the Events tab, you can view and edit events handlers associated with the currently active GUI component.
4. You can have your application respond to key presses, single, double and triple mouse clicks, mouse motion, window size and focus changes. You can generate event handlers for all of them from the Events menu. The most common event you will use is an Action event.
5. How does event handling work? Every time you select an event from the Event menu, the IDE automatically creates a so-called event listener for you, and hooks it up to your component.

*Go through the following steps to see how event handling works.*

1. Go back to the file XYZUI.java in the Editor. Click the Source tab to see the GUI's source.
2. Scroll down and note the methods jButton1ActionPerformed (), jButton2ActionPerformed (), and jButton3ActionPerformed () that you just implemented. These methods are called event handlers.
3. Now scroll to a method called initComponents(). If you do not see this method, look for a line that says Generated Code; click the + sign next to it to expand the collapsed initComponents() method.
4. First, note the blue block around the initComponents() method. This code was auto-generated by the IDE and you cannot edit it.
5. Now, browse through the initComponents() method. Among other things, it contains the code that initializes and places your GUI components on the form. This code is generated and updated automatically while you place and edit components in the Design view.
6. In initComponents(), scroll down to where it reads
7. jButton3.setText("Exit");
8. jButton3.addActionListener(new java.awt.event.ActionListener() {
9. public void actionPerformed(java.awt.event.ActionEvent evt) {
10. jButton3ActionPerformed(evt);
11. }

});

This is the spot where an event listener object is added to the GUI component; in this case, you register an ActionListener to the jButton3. The ActionListener interface has an actionPerformed method taking ActionEvent object which is implemented simply by calling your jButton3ActionPerformed event handler. The button is now listening to action events. Everytime it is pressed an ActionEvent is generated and passed to the listener's actionPerformed method which in turn executes code that you provided in the event handler for this event.

**TESTING**

This software is properly tested on many platforms with all the features. Initially it was tested on one PC with IP Address “localhost” to check whether messages are sent and received. Then it was tested on two to check with IP Address of server. After its success, the feature of image file was added and then again it was checked on localhost. Following the above process, it was checked on two.

After finding the errors and removing them, some more features of sending more files, save and clear chat were added and then again the software was tested.

**LIMITATIONS**

With the development of software, limitations also get developed with the development of benefits.  
Since there are many benefits of this software, there are some limitations too.  
Some of these are:

* It doesn’t work on Bluetooth as it was planned to do.
* This can send files upto size 100MB.
* Works only on windows/Linux.
* Once Clients are connected to Server, Server cannot remove them.
* Files sent other than images or text will not open in the chat area, they need to be opened in another window.
* While saving the chat, only text messages will be saved.
* Emoticons are yet not added as a feature.
* It doesn’t support offline messaging. Only Live chat is possible.

**CONCLUSIONS AND FUTURE WORK**

With the working of this Project many concepts were cleared. Many implementations were made successful like:

* Advanced Java is more clearly understandable now.
* Socket Programming in Java is fully implemented and is well known and will help in further semesters.
* Concepts of Event Handling, Swings, GUI building are totally implemented.

**Future Work:**

All the limitations that are being observed in the Project are considered as the future work. Some of these are:-

* It doesn’t work on Bluetooth as it was planned to do.
* This can send files upto size 100MB.
* Works only on windows/Linux.
* Once Clients are connected to Server, Server cannot remove them.
* Files sent other than images or text will not open in the chat area, they need to be opened in another window.
* While saving the chat, only text messages will be saved.
* Emoticons are yet not added as a feature.
* It doesn’t support offline messaging. Only Live chat is possible.

**YouTube Video Link:**

https://www.youtube.com/watch?v=N-9ysV3x5lM

**REFERENCES**

* Calvert-TCP-IP sockets in java.
* <http://www.javatpoint.com/event-handling-in-java>
* <https://netbeans.org/kb/docs/java/gui-functionality.html>
* Java The Complete Reference, 8th Edition by Herbert Schildt.
* <http://www.javatpoint.com/java-swing>
* [www.stackoverflow.com](http://www.stackoverflow.com)