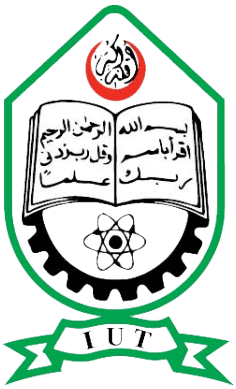
Islamic University of Technology



Visual Programming Lab

CSE 4402

Final Project Report: XChange

Author:
MD AKIB HAIDER
SAMEEN YEASER
SAKEEF HOSSAIN

July 4, 2024

Contents

1 Introduction

1.1 Statement and Purpose

XChange is a Java application that enables secure, fast and efficient file transfer between multiple devices connected on the same local network, eliminating the need for an internet connection. It aims for:

- **Internet Independency**: Plethora of file-sharing systems relying on internet connection out there, which can be a limitation in areas with poor connectivity specially in some remote places. Students and other working professionals often need to share resources without internet connection. XChange primarily deals with that issue.
- **Fast and Efficient**: Internet based file transfer process can be slow and inefficient, when it comes dealing with large files and poor internet. It aims to provide a faster and more reliable alternative by leveraging the local network.
- **Generic and Accessible**: Easy to use, having a modern UI design makes it suitable for users with various levels of technical proficiency. All it requires is a Java compiler on local computer without any complex software requirements.

1.2 Key Functionalities and Features

Key functionalities and features making it an effective and efficient file-sharing system are:

- **File Transfer Over Local Network**: Facilitates file transferring between devices on the same local network, overcoming the need for an internet connection. Thus, it makes the process more secure.
- **File Transfer Of Any Size**: Supports file transfer of mostly any size be it large or small files at a handsome speed.
- **High Speed File Transfer with Speed Checker**: By using the local network, it achieves faster file transfer speed compared to internet-based systems. It also has an interactive speed checker that will give continuous feedback of the transfer process including current progress of time of the sent files.
- **Multi-File Support**: Users can select and send multiple files at one go when connected, enhancing the overall user satisfaction.
- **Friendly User Interface and Usability**: The system includes mechanisms to handle errors and ensures that the whole process is completed reliably with necessary warning warning, error and information messages.
- **File Type Compatibility**: Files can be sent of multiple types such as, image, text, video etc.

2 System Design

2.1 System Architecture

Key architectural components are:

☞**MainFrame**: The main user interface that interacts with both senders and receivers to inter-connect, select files, send or receive files and initiate the overall transfer process. It also displays the status of the ongoing transfers.
☞**Client**: The client triggers the file transfer request and sends file to the server created. It establishes connection with the server through broadcasting and multi-casting, making all nearby devices accessible over a local network.
☞**Server**: The server receives files from the client, reserves those and later supplies to other clients (as receivers) when demanded. Server does necessary processing like error handling, client(user) verification, username & IP address access and so on.
☞**Sender**: The sender manages the actual transferring process from the client to the server. It logically handles various types of files e.g. images, videos, audios, documents etc by converting all into binary files. Datagram approach is used with java sockets for network communication between client and server.

2.2 UML Diagrams

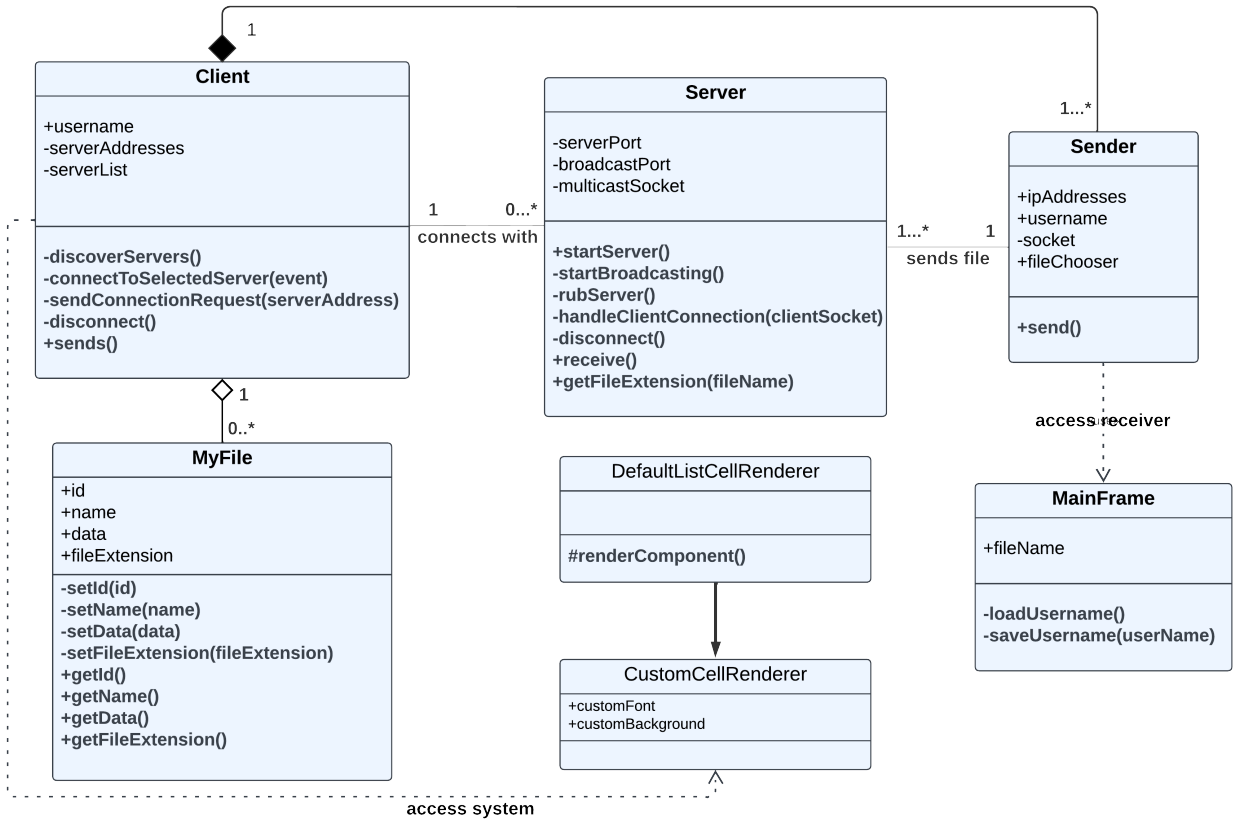


Figure 1: UML class diagram of XChange

3 Implementation Details

3.1 High-Level Overview

XChange consists of five main classes: Server, Sender, MyFile, Client, and MainFrame. Firstly, it connects two devices to each other after exchanging requests between them. Then it handles the selection of the file to be sent and sends it over the network then receives it from receiving end. As for the implemented functionalities:

1. **MainFrame:** This is the main class at which the application begins. It offers the first window which includes the username bar which shows the current username if selected along with the "set username" button. It is used to navigate between the client and server components and to initialize the components needed.

2. **Server:** Main task of the server is to broadcast the device with its username in the network and to inform potential clients that the server is available for connections by using a multi-cast socket and send broadcast messages at regular intervals. It manages incoming client connections and upon approval, connects to the selected device and receives files. It also handles reception of files and stores the received files from the sender device locally in the server. It has mechanisms to read incoming file data from clients as "Data Input Stream", displaying progress bars for real-time progress of file transfer and writing the received file data to the local file system.

3. **Client:** The client class is defined to allow users to search for devices it wants to send the file. It manages connection to the server by listing servers, sending requests and handling connection to the desired server by obtaining the IP address and providing it to the sender class. It uses the same multi-cast socket to send requests for client connections in the same address and port of the server.

4. **Sender:** The GUI for choosing and transmitting files to the designated IP addresses is provided by the Sender class. It lets choosing necessary files, managing file names and sizes, and delivering files to the receiving device.

5. **MyFile:** Lastly MyFile class is used to define a data structure for storing file information such as id for the file, file name as string, data in the form of a byte array and the file extension.

3.2 Design Choices and Algorithms Employed

- **User Interface Design:** The GUI framework has been developed using Java Swing that gives the application a clean intuitive UI. Nimbus look and feel with custom dark colors has been used for the GUI components to create a modern and sleek interface.

- **File Transfer Protocol:** The use of TCP secures the fast delivery of files and also makes the transfer to be efficient and lossless, preventing any damage to the data.

- **Multi-threading:** Multiple files can be sent concurrently using multi-threading without causing the user interface to lock up. Broadcasting is carried out on a separate thread to avoid interfering with the primary server's functionality.

- **Multi-cast Socket:** The client actively connects to a multi-cast group and continuously broadcasts messages with the server's identifier.
- **Client Handling:** In case of each link receiving a connection, a new thread is generated to control the interaction with the client and allows the server to work with other clients simultaneously.
- **Progress Monitoring:** Organizes a report of how far the file that is being received has gone through the process with a GUI Graph to indicate how far it is.
- **File Reading and Writing:** The application reads and writes files in chunks to efficiently handle large files without consuming excessive memory.
- **File Metadata Management:** Managing the file information centrally is beneficial for the application, that is why the MyFile class contains all the necessary information about the file.
- **Socket Communication:** Sockets are used by the server and client to establish a connection. The server simply waits for the Data stream based on the IP address while the client initiates connection and transfers files for Data stream.

3.3 Critical Code Section

1. **discoverServers() method in client class**
    - Creates and joins a multi-cast group on the specified port.

```
private void discoverServers() {
    Executors.newSingleThreadExecutor().execute(() -> {
        try {
            multicastSocket = new MulticastSocket(BROADCAST_PORT);
            InetAddress group = InetAddress.getByName(MULTICAST_ADDRESS);
            multicastSocket.joinGroup(group);
            int i=1;
            byte[] buf = new byte[256];
            while (true) {
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                try {
                    multicastSocket.receive(packet);
                } catch (SocketException e) {
                    // Socket closed, exit loop
                    break;
                }
                String serverName = new String(packet.getData(), 0, packet.getLength());
                serverAddress = packet.getAddress().getHostAddress();
                // Skip own server broadcasts and already discovered servers
                if (!serverAddresses.contains(serverAddress) && !serverAddress.equals(InetAddress.getLocalHost().getHostAddress())) {
                    serverAddresses.add(serverAddress);
                    serverListModel.addElement(i + ". " + serverName + " (" + serverAddress + ")");
                    i++;
                }
            }
```

Figure 2: discoverServers() method

- Runs an infinite loop to continuously receive packets. The loop closes when the socket is closed. The packets are the ones broadcast by the server in the port.

- For every packet received, it checks if the server address is already discovered or if it is a local server. If not, it adds the server to the list and updates the server to the list created inside the class and updates the server list model with the new server details.

2. **sendConnectionRequest(serverAddress) method in client class**

- Attempts to create a socket connection to the server using the provided server address and port. It also sets up an output and input streams for communication.

```
private void sendConnectionRequest(String serverAddress) {
    try (Socket socket = new Socket(serverAddress, SERVER_PORT);
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {
        out.println(username);
        String response = in.readLine();
        JOptionPane.showMessageDialog(frame, response);
        if(response.equals("Connection Successful")){
            disconnectButton.setEnabled(true);
            sendButton.setEnabled(true);
            this.address.add(serverAddress);
        }
        else{
            disconnectButton.setEnabled(false);
            sendButton.setEnabled(false);
        }
          this.address.add(serverAddress);
        System.out.println(serverAddress);
    } catch (IOException e) {
        e.printStackTrace();
    }
```

Figure 3: sendConnectionRequest() method

- It sends the username to create a request on the server side. Reads the server's response and stores it in 'response' variable.
- If connection is successful, the send and disconnect options are enabled for use.

3. **Send Button Action in sender class**

- Firstly the frame for the progress bar is created, progress bar is created and configured, speed label is created and configured, setting layout and adding components is done.
- A 'FileInputStream' is initialized for the file to be sent, allowing the program to read file's contents.
- A socket connection is formed using the IP address obtained during initialization from the client class.
- Data output stream is initialized to send data through the socket.
- The file name after its length is sent to the recipient in bytes. The length is sent to make it easy for the server side code to execute.

```
sendButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (fileToSend[0] == null) {
            filenameLabel.setText("Please, choose a file first");
        } else {
            for(int i = 0;i<count;i++){
                JFrame progress_frame = new JFrame("Sending File");
                progress_frame.setSize(400, 150);
                progress_frame.setLocationRelativeTo(frame);
                progress_frame.getContentPane().setBackground(new Color(45, 45, 45));
                JProgressBar j_progress_bar = new JProgressBar(0, (int) fileToSend[0].length());
                j_progress_bar.setStringPainted(true);
                j_progress_bar.setForeground(new Color(41, 128, 185));
                j_progress_bar.setBackground(new Color(60, 63, 65));
                JLabel j_l_speed = new JLabel("Speed: 0 B/s", SwingConstants.CENTER);
                j_l_speed.setFont(new Font("Roboto", Font.PLAIN, 16));
                j_l_speed.setForeground(new Color(210, 210, 210));
                progress_frame.setLayout(new BoxLayout(progress_frame.getContentPane(), BoxLayout.Y_AXIS));
                progress_frame.add(j_progress_bar);
                progress_frame.add(j_l_speed);
                progress_frame.setVisible(true);
```

Figure 4: sendButton Action (i)

- File size is sent in bytes as integer. A buffer is created to hold file-data chunks to be read and sent. This part is responsible for establishing a socket connection, sending the file's metadata (name and size), and preparing to transfer the file's data in chunks. The transfer speed and progress can be monitored using the 'totalBytesRead' and 'startTime' variables.

```
int finalI = i;
new Thread(() -> {
    try {
        FileInputStream file_input_stream = new FileInputStream(fileToSend[0].getAbsolutePath());
        System.out.println(ipAdd.get(finalI) + "Sender");
        Socket socket = new Socket(ipAdd.get(finalI), 1234);

        DataOutputStream data_output_stream = new DataOutputStream(socket.getOutputStream());
        String file_name = fileToSend[0].getName();
        byte[] file_name_byte = file_name.getBytes();
        data_output_stream.writeInt(file_name_byte.length);
        data_output_stream.write(file_name_byte);
        data_output_stream.writeInt((int) fileToSend[0].length());

        byte[] buffer = new byte[4096];
        int bytesRead;
        long totalBytesRead = 0;
        long startTime = System.currentTimeMillis();
```

Figure 5: sendButton Action (ii)

4. **Choose Button action in sender class**

- UIManager is used to choose the "Nimbus" look and feel. This sets the look and feel of the file chooser to match the system's native look.

```
chooseButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            // UIManager.put("FileChooserUI", "javax.swing.plaf.metal.MetalFileChooserUI");
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException | UnsupportedLookAndFeelException ex) {
            ex.printStackTrace();
        }
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Choose a File to Send");
        try {
            if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
                fileToSend[0] = fileChooser.getSelectedFile();
                filenameLabel.setText("The file you want to send is: " + fileToSend[0].getName());
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(frame, "Error selecting file: " + ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
});
```

Figure 6: ChooseButton Action

- Inside the second try catch block it shows the file chooser dialog and checks if the user approves the file selection. If a file is selected, it sets "fileToSend[0]" to the selected file and updates "filenameLabel" with the file name. It catches and prints exceptions if any error occurs during file selection and shows an error message dialog.

5. **sends() method in client class**

- Sender object of sender class is initialized.

```
public void sends() {
    Sender sender = new Sender(username);

    int siz = address.size();
    for(int i = 0;i<siz;i++){
        sender.addDevice(address.get(i));
    }
    frame.dispose();

    sender.send(); // Start the sender
}
```

Figure 7: sends() method

- "address" list is used to store all the devices' IP addresses which are selected for connection. The loop iterates through every element and uses 'addDevice()' method to send the IP addresses to the sender class.
- After that the frame is disposed and sender is initialized and the frame in sender object is shown.

6. **startBroadcasting() method in server class**
   It handles the multicast broadcasting part. It starts a new thread to broadcast the server's availability to clients' in a multicast group. Breakdown of the process:

```
private void startBroadcasting() {
    Executors.newSingleThreadExecutor().execute(() -> {
        try {
            multicastSocket = new MulticastSocket(BROADCAST_PORT);
            InetAddress group = InetAddress.getByName(MULTICAST_ADDRESS);
            multicastSocket.joinGroup(group);

            byte[] msg = username.getBytes();
            while (running) {
                DatagramPacket packet = new DatagramPacket(msg, msg.length, group, BROADCAST_PORT);
                multicastSocket.send(packet);
                Thread.sleep(5000); // Broadcast every 5 seconds
            }
            multicastSocket.leaveGroup(group);
            multicastSocket.close();
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
```

Figure 8: startBroadcasting() method

- Opens a multicast socket on the 'BROADCAST_PORT'.
- Joins the multicast group specified by 'MULTICAST_ADDRESS'.
- Continuously sends the server's username to the multicast group every 5 seconds as long as the server is running so that the clients will continuously see the availability of the server in the multicast group.
- Leaves the multicast group and closes the socket when the server stops running.

7. **runServer() method in server class**

```
private void runServer() {
    Executors.newSingleThreadExecutor().execute(() -> {
        try (ServerSocket serverSocket = new ServerSocket(SERVER_PORT)) {
            while (running) {
                Socket clientSocket = serverSocket.accept();
                handleClientConnection(clientSocket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
}
```

Figure 9: runServer() method

Server Socket Listening occurs in this part.

- Opens a server socket on the 'SERVER_PORT'.
- Continuously listens for incoming client connections.
- Accepts a client connection and passes the client socket to the 'handleClientConnection()' method for further handling

8. **handleClientConnection(clientSocket) method in server class**
   Manages the client connection request and response process in a server application.

```
private void handleClientConnection(Socket clientSocket) {
    Executors.newSingleThreadExecutor().execute(() -> {
        try (BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)) {

            String clientName = in.readLine();
            int response = JOptionPane.showConfirmDialog(frame,
                    clientName + " wants to connect with you. Accept?",
                    "Connection Request",
                    JOptionPane.YES_NO_OPTION);

            if (response == JOptionPane.YES_OPTION) {
                out.println("Connection Successful");
                textArea.append("Connected to " + clientName + "\n");
                disconnectButton.setEnabled(true);
                Receive();
            } else {
                out.println("Connection Denied");
                textArea.append("Connection denied for " + clientName + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
```

Figure 10: handleClientConnection() method

- Creating a new thread with "Executors.newSingleThreadExecutor().execute(() -¿" to handle the client connection asynchronously.
- Opens a 'BufferedReader' to read input from the client and a 'PrintWriter' to send output to the client. These streams are automatically closed when the try block exits.
- If the user accepts the connection (with JOptionPane.YES_OPTION):
  ☞ Sends a "Connection Successful" message to the client.
  ☞ Updates the text area to show the connection status.
  ☞ Enables the disconnectButton.
  ☞ Calls the Receive() method to start receiving data from the client.
- If the user denies the connection:
  ☞ Sends a "Connection Denied" message to the client.
  ☞ Updates the text area to show that the connection was denied.

9. **Receive() method in server class**
Continuously listens for incoming file transfer connections, receives and saves the file locally.
☞ Waits and accepts an incoming connection from a client.

```
Socket socket = server_socket.accept();
```

☞ Reads length of the file name, file name itself, and length of the file content from data stream.

```
DataInputStream data_input_stream = new DataInputStream(socket.getInputStream());
int file_name_length = data_input_stream.readInt();
if (file_name_length > 0) {
    byte[] file_name_bytes = new byte[file_name_length];
    data_input_stream.readFully(file_name_bytes, 0, file_name_bytes.length);
    String file_name = new String(file_name_bytes);
    int file_content_length = data_input_stream.readInt();
    if (file_content_length > 0) {
        byte[] file_content_bytes = new byte[file_content_length];
```

☞ Continuously reads chunks of the file data from the data stream until the entire file is received. Also handles the loading of the progress bar along with its speed.

```
while (bytesRead < file_content_length) {
    int bytesRemaining = file_content_length - bytesRead;
    int chunkSize = Math.min(bytesRemaining, 4096);
    byte[] buffer = new byte[chunkSize];
    int read = data_input_stream.read(buffer, 0, chunkSize);
    if (read > 0) {
        System.arraycopy(buffer, 0, file_content_bytes, bytesRead, read);
        bytesRead += read;
        j_progress_bar.setValue(bytesRead);

        long currentTime = System.currentTimeMillis();
        double elapsedTime = (currentTime - startTime) / 1000.0; // seconds
        double speed = (bytesRead / 1024.0) / elapsedTime; // KB per second
        j_l_speed.setText(String.format("Speed: %.2f KB/s        Time: %.2f s", speed, elapsedTime));

    }
}

File file_to_save = new File(file_name);
try (FileOutputStream file_output_stream = new FileOutputStream(file_to_save)) {
    file_output_stream.write(file_content_bytes);
}
```

Figure 11: Receive() method

☞ Writes received file content to a new file on local file system using 'FileOutputStream'.

4 GUI Design

The main user interface of the "XChange Client" application is designed using the Java Swing framework. The design aims to provide a modern, clean, and intuitive interface for users to interact with the file transfer functionalities.

4.1 UI design layout

1. **Main Frame**
   (a) Main frame is the primary window of the application.
   (b) It is configured with a custom title, size to allow for custom styling.
2. **Custom Title Bar**
   (a) A custom title bar is implemented using a JPanel to replace the default window decorations.
   (b) It includes a title label and a close button styled to match the application's theme.
   (c) Title bar is placed at the top of the frame using the BorderLayout
3. **Content Panel**
   (a) A main content panel (JPanel) using BoxLayout for vertical alignment of components.
   (b) It includes the username panel, description label, message label, and action buttons.
   (c) The panel is set with a background color and padding to enhance the visual structure.
4. **Action Buttons**
   (a) Two primary action buttons, "Send" and "Receive," are created using JButton.
   (b) Each button is styled for a modern look with specific fonts, colors, and sizes.
   (c) They are placed in a grid layout within a panel for even spacing and alignment.
5. **Progress Frame**
   (a) Displayed during file sending.
   (b) Contains a JProgressBar and a JLabel to show sending progress and speed.
   (c) Progress updated using 'SwingUtilities.invokeLater'.

4.2 Design Ideology

- **Nimbus Look and Feel:** The code sets the Nimbus Look and Feel for the GUI to achieve a modern, aesthetically pleasing appearance. The idea is to give our application a dark and light theme. A dark theme is implemented by setting various UIManager properties to dark colors. Nimbus allows dynamic switching between themes, enabling applications to offer users the choice between different visual styles. The default color scheme of Nimbus is designed to provide high contrast, which improves readability and usability, especially for users with visual impairments. Polished visuals with smooth gradients and professional look take it to a new level. In our application Nimbus is used for a dark-themed interface that aligns with these principles.

- **Model-View-Controller Design Pattern** Following the MVC pattern, 'MainFrame' acts as the view, 'Client' and 'Sender' handle the controller logic, 'Server' manages the model (file storage and processing).

- **Custom Styling:**
  Dark Theme: The GUI elements are styled to have a dark theme using custom colors set via UIManager. Colors like dark gray, light gray, and sky blue are used to maintain consistency and improve aesthetics.
  Fonts and Borders: Custom font 'Roboto' and 'Raleway' are used for labels and buttons. Borders and padding are used to adjust the layout and spacing of elements.

4.3 User Interaction and Navigation

1. **Setting the Username**

   - **Launch Application** Users start by launching the application.

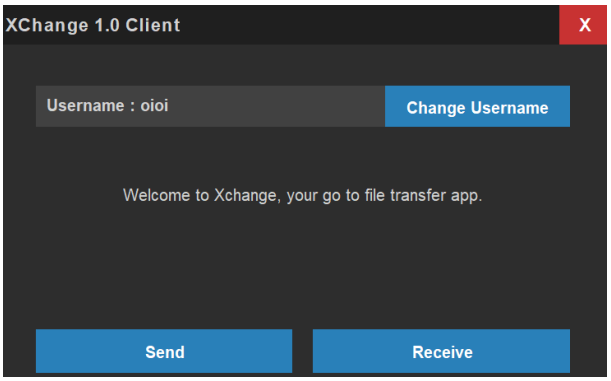


Figure 12: Main Menu

   - **Set Username** A dialog box prompts users to enter their username, which will be used as their client's name for identification.

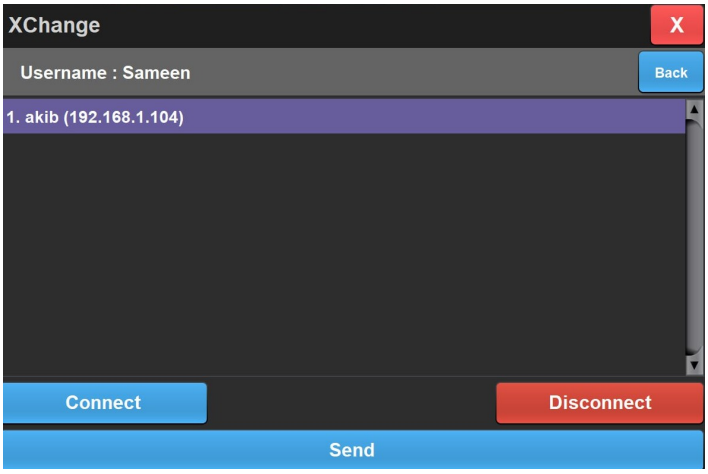


Figure 13: Connecting to available devices

- **Establish Connection** Users confirm the entered username to proceed. This username will be displayed and used for connection purposes.

2. **Sending a File**

- **Click Send Button** Users click the "Send" button on the main frame.
- **Open File Sending Window** A new window for sending files opens.

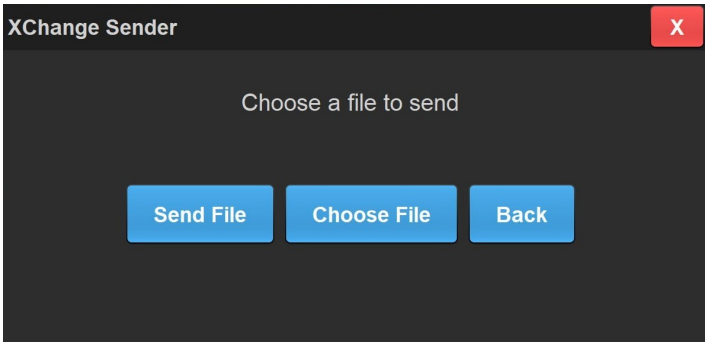


Figure 14: File Sending Window

- **Enter Receiver's IP Address** In the provided box, users find the receiver's IP address.
- **Click IP Address** Users click on the IP address to select it.
- **Press Connect** Users press the "Connect" button to establish a connection with the receiver site.
- **Connection Confirmation** Once connected, the user can see the connection status.
- **Send File** Users can now select the file they want to send and click the "Send" button to transfer the file to the receiver.

3. **Receiving a File**

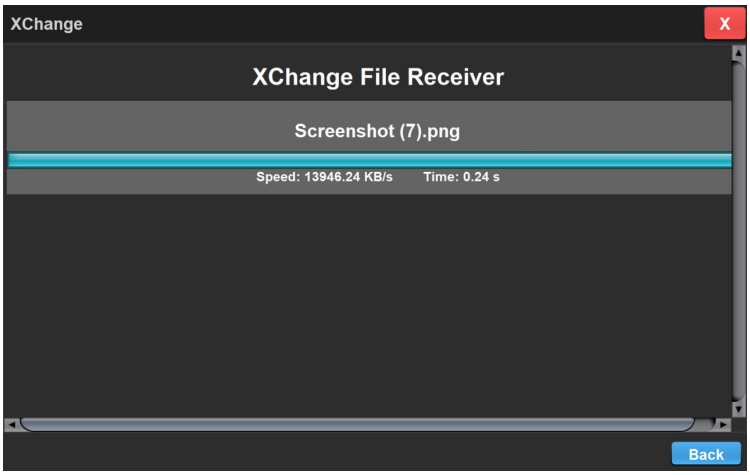


Figure 15: File Receiving Window

- **Click Receive Button** Hit the "Receive" button on the main frame.
- **Wait for Connection** If the sender is connected to the receiver site, the receiver will be ready to accept files.
- **Receive File** When the sender sends a file, the receiver automatically receives the file.

5 Testing and Evaluation

5.1 Testing strategies

First of all, unit testing has been performed for the sender class before building the MainFrame interface; only to test the successful transfer of files to devices active in the local network with destination port and IP addresses manually given. Likewise, server has been unit tested to broadcast the device with its username in the network and to inform potential clients that the server is available for connections by using a multi-cast socket. Finally, upon successful completion of those, sender went through both unit testing and integration testing for successful connection to multicast group of the server ensuring delivery of files using TCP protocol. Speed testing and progress monitor were also tested.

5.2 Bug Fixing and Challenges

☞ **Issue:** First problem encountered was that the code needed the IP address of the receiving device needed to be hard coded.
☞ **Solution:** The code was updated majorly by adding the feature of using multicasting where a multicast packet is received and processing content the address is obtained. In details, the client side was able to pick the devices it wants to set to and the program using multicasting sockets and packets ip address was obtained in the Client class and sent to the Sender class. The Sender class was then was able to form a socket connection using the specific IP address to send the chosen file.

☞ **Issue:** Initially, the application was able to select one device and send a file to that device only. Hence it was not possible to send the file to multiple devices at the same time.
☞ **Solution:**

1. The address variable in Client class which was initially a variable was converted to a list so that the client can select multiple receivers and thus able to store all of their IP addresses

2. In the sends() method of Client class a loop was run to send all the IP addresses to Sender class. Besides an addDevice() method was introduced in Sender class to take in all the IP addresses in a list inside the Sender class.

3. A count variable keeps count of the number of devices connected in the Sender class. Now inside the action listener of the "Send" button, all the codes were taken inside a for loop running on connect so that upon a click of the "Send" button, the file would be sent to all the devices it is connected to one after another.

☞ **Issue:** Type of data while transferring the file was not consistent
☞ **Solution:** Filename and file content both of the data type were converted into binary form. filename length along with transformed filename data sent in a packet first and then main data length along with

transformed actual binary data were sent together in a packet. ArrayList data structure was used to deal with multiple filenames.

6 Conclusion

6.1 Key Achievements

One of the key achievements of the project was to develop a internet-independent file sharing system with less complexity. Other accomplishments are:

- File transferring over local network is possible. Only being connected under same network will do the job especially in critical times without internet.
- Transfer speed and efficiency of the communication process outperforms most other platforms.
- Modern minimalist design also stands out, easy to use and simple layout makes user comfortable to use it.
- Eliminates restriction of file sizes while transferring of some popular applications. (Most of the daily used apps like Gmail, Messenger, Discord etc has a limit of 25MB, only 16MB for WhatsApp)
- Highly applicable in student community, office, work groups and nearly any place with minimum setup requirements and software dependencies.

6.2 Future Improvement Scope

Some potential areas for improvement are as follows:

- Multiple file picking and transferring of files while opening "file picker" dialog box at one operation.
- "Drag and drop"-ing the files when sending
- Downloading files to a specific directory on users' choice.
- Temporarily run or read the file from the server while downloaded.
- One client can send to multiple devices but receiver cannot receive from multiple clients at a time.

References

1. javax.swing (Swing GUI components)
2. java.awt (Abstract Window Toolkit for GUI components)
3. java.io (I/O operations)
4. java.net (Networking operations)
5. UIManager (Setting Look and Feel in Swing)