三、实验过程或算法（源程序、关键代码或技术描述）

1. 实验设计思想

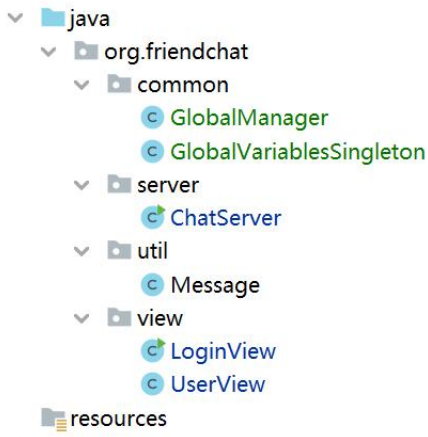
本实验基于 java 语言，使用 Java Socket API 套接字实现了一对一和一对多的多人聊天系统。总体分为 Server 服务器端和 Client 客户端，前端主要采用 java 的 swing 实现。

对于服务器端，使用 ServerSocket 建立服务器唯一套接字，使用 4919 端口建立起 socket 用于接收和转发客户消息或文件，并在后台实时打印在线人数情况和各个成员之间发送消息情况。

对于客户端，当用户登录至聊天系统时，通过本机地址和 4919 端口，使用 Socket 与服务器建立连接，每建立一次连接，服务器端会将该用户的 socket 存在一个数组中唯一标识该用户；当客户端断开连接时，服务器端会将其从数组中移除。每次新增或移除一个用户，服务器端均会向所有用户发送一次消息告知在线人数并在后台打印，以此实现动态展示在线人数列表。

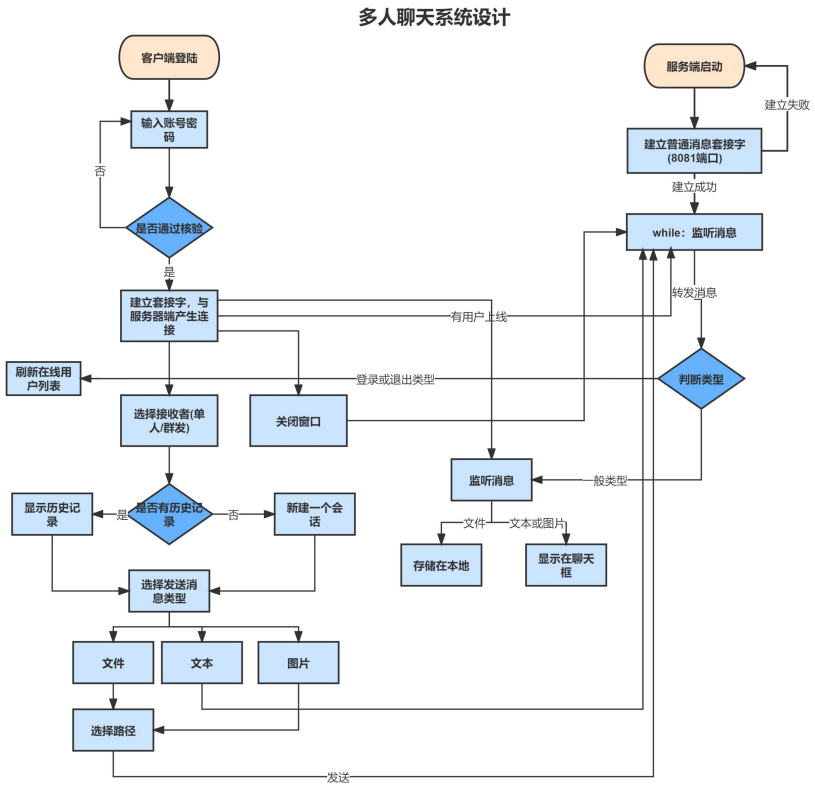
除用户自身外，用户可选择和其他在线用户进行一对一通信，也可以自行选择和其他成员进行通信，并保持各自通信记录的独立性。

2. 项目结构说明



Package	Class	说明
com	GlobalManager	管理用户全局变量
	GlobalVariablesSingleton	管理用户聊天记录
server	ChatSever	服务器端接收和管理与客户端建立起的套接字，接收和转发用户的消息
util	Message	用户发送的消息类
view	LoginView	客户登陆界面
	UserView	客户消息聊天界面

3. 项目的功能说明图如下：



4. 关键代码描述说明

4.1 服务器端建立

4.1.1 后台设计：

服务器端未设置单独的前端界面，仅在后台打印记录每一位用户的登录或退出情况，以及各个用户发送的每一条信息。from 代表消息发送者，to 为接收者(列表)，type 为消息类型，content 为消息正文，Online user 用于实时记录在线人数。

```
private static void serverLog(Message message) {
    System.out.println("MESSAGE FROM `" + message.from +
            "` TO `" + message.to +
            "` TYPE `" + message.type.name() +
            "` CONTENT `" + message.content + "`.\nOnline user: `" + users + "`.");
}
```

具体的消息类型如下：

```
public enum MESSAGE_TYPE {
    2 usages
    LOGIN,  //登录
    2 usages
    LOGOUT, //退出登录
    3 usages
    ONLINE_LIST, //在线列表
    5 usages
    MESSAGE,  //文本消息
    4 usages
    FILE,    //文件
    4 usages
    IMAGE   //图片·
}
```

4.1.2 套接字建立

在 ChatServer.java 类中设置 main 函数用于启动服务器端，当服务器端启动时，首先通过 SeverSocket 建立一个套接字，本实验中使用 4919 端口用于接收和发送消息。当建立套接字后使用 server.accept()监听等待建立连接，当建立连接后对建立连接的客户端开启相应的线程。

```
@SuppressWarnings("InfiniteLoopStatement")
public static void main(String[] args) {
    try {
        serverSocket = new ServerSocket( port: 4919);
        while (true) {
            Socket socket = serverSocket.accept();
            readMessage(socket);
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

每新增一个用户，服务器端就会将建立起的 socket 存放在一个数组 clients 中，并用 users 数组记录对应的用户名

```
switch (message.type) {
    case LOGIN -> {
        String user = message.from;
        if (!users.contains(user)) {
            users.add(user);
            clients.add(socket);
        }
        writeMessage(new Message( from: "SERVER", users, Message.MESSAGE_TYPE.ONLINE_LIST,  content: null));
    }
```

每退出一个用户，则将其 clients 和 users 数组中移除，并群发消息更新在线用户列表。

```
    } catch (IOException | ClassNotFoundException e) {
        int index = clients.indexOf(socket);
        String user = users.get(index);
        users.remove(index);
        clients.remove(index);
        writeMessage(new Message( from: "SERVER", users, Message.MESSAGE_TYPE.LOGOUT, user));
        writeMessage(new Message( from: "SERVER", users, Message.MESSAGE_TYPE.ONLINE_LIST, content: null));
    }
```

4.1.3 接收和转发消息

为了能实时监听是否有消息发送过来，服务器端使用 while 循环来监听是否有消息输入流，并根据接收消息的类型(使用 case 语句)进行不同的操作。

①实时更新在线人数列表：

对接收消息的类型进行判断。如果为登陆类型则加入在线人数列表，并建立起用户名与套接字的映射；若为退出登录类型则做相反的操作。

```
new Thread(() -> {
    try {
        while (true) {
            ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream());
            Message message = (Message) inputStream.readObject();
            switch (message.type) {
                case LOGIN -> {
                    String user = message.from;
                    if (!users.contains(user)) {
                        users.add(user);
                        clients.add(socket);
                    }
                    writeMessage(new Message( from: "SERVER", users, Message.MESSAGE_TYPE.ONLINE_LIST, content: null));
                }
```

②转发消息或文件

对发送方发送的消息获取转发者(message.to),采用 for 循环语句，使用 ObjectOutputStream 输出流，依据 socket 的唯一性对消息转发给对应的客户端。

```
private static void writeMessage(Message message) {
    for (String user : message.to) {
        Socket client = clients.get(users.indexOf(user));
        try {
            ObjectOutputStream outputStream = new ObjectOutputStream(client.getOutputStream());
            outputStream.writeObject(message);
            outputStream.flush();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

4.2 客户端建立

4.2.1 登陆界面设计：

在登录界面需要进行账号密码登录，验证通过后才能进入聊天。这里为了简化过程，用户的账号采用 1-100 的整数，密码统一设置为 123。

```
public LoginView() {
    JLabel loginLabel = new JLabel( text: "Login");
    loginLabel.setFont(new Font( name: null, Font.PLAIN,  size: 35));

    JPanel loginPanel = new JPanel();
    loginPanel.add(loginLabel);

    JLabel userLabel = new JLabel( text: "User");
    userTextField = new JTextField( columns: 20);

    JLabel passwdLabel = new JLabel( text: "Password");
    passwdTextField = new JPasswordField( columns: 20);

    JButton loginButton = getLoginButton();

    JPanel userPanel = new JPanel();
    userPanel.add(userLabel);
    userPanel.add(userTextField);
    userPanel.add(passwdLabel);
    userPanel.add(passwdTextField);
    userPanel.add(loginButton);
```

提供的登录逻辑如下：

```
private void login() {
    String user = userTextField.getText();
    String passwd = passwdTextField.getText();
    if (passwd.equals("123")) {
        //-----
        GlobalManager.getInstance().getGlobalVariablesSingleton(user).setContentMap(user,"**********欢迎来到聊天框!**********");
        new UserView(user);
        globalFrame.setVisible(false);
    } else {
        JOptionPane.showMessageDialog(
                globalFrame,
                 message: "Wrong user or password.",
                 title: "Login failed",
                JOptionPane.WARNING_MESSAGE);
        passwdTextField.setText("");
    }
```

4.2.2 客户端套接字建立

在用户登录校验成功后，会通过本机地址和 4919 端口与服务器端建立起连接。

```
public UserView(String _user) {
    self = _user;
    try {
        InetAddress address = InetAddress.getLocalHost();
        socket = new Socket(address,  port: 4919);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
```

4.2.3 客户端接收和发送消息

①实时显示在线用户列表

客户端同样采用 while 循环实时监听接收的消息，如果接收的消息类型为登录或退出操作，则从 Message 中获取在线用户列表 onlineList,并更新到聊天框中，这样就可以与服务器端实时同步在线用户列表。

- **登陆通知：**

```
switch (message.type) {
    case ONLINE_LIST -> {
        for (int i = 0; i < 100; ++i) {
            onlineList[i].setVisible(message.to.contains(String.valueOf(i)));
        }
    }
```

- **退出通知：**

```
case LOGOUT -> group.remove((String) message.content);
```

②接收和存储普通消息

每位在线用户均设置一个单选框，用于表示是否为消息发送方和接收方。从接收的消息中获取聊天者(message.to)，除用户自身外，将发送方和其余接收方进行勾选，无论是单人聊天还是多人聊天均能满足需求：

```
case MESSAGE -> {
    group = message.to;
    group.add(message.from);
    group.remove(self);
    updateRecordWhenReceive( contents: message.from + ": "+ message.content + "\n");
    for (int i = 0; i < 100; ++ i) {
        onlineList[i].setSelected((group.contains(String.valueOf(i)) || Objects.equals(message.from, String.valueOf(i))) && !Objects.equals(self, String.valueOf(i)));
    }
}
```

为了实现通讯的独立性(即不同的聊天保持各自聊天内容一致)，在界面右侧使用 DefaultListModel 来保存每一次聊天的对象，并使用 Hash 映射 map 来保持唯一性，避免重复：

```
4 usages
DefaultListModel<String>friendListModel = new DefaultListModel<>();  //显示好友列表
5 usages
Map<String,List<String>>getFriendListMap = new HashMap<>();  //好友列表名和成员的映射
```

每次接收消息时，首先判断当前聊天者是否有记录，如果有则追加历史记录并显示在聊天框，反之则用当前聊天者作为键，新建聊天记录：

```
/**
 * 接收消息时实时更新当前收到的消息：逻辑同上
 */
2 usages   new *
public void updateRecordWhenReceive(String contents){
    sortedGroup.clear();
    sortedGroup.addAll(group);
    Collections.sort(sortedGroup);
    String key = sortedGroup.toString();
    if(GlobalManager.getInstance().getGlobalVariablesSingleton(self).getContentMap(key)==null){
        GlobalManager.getInstance().getGlobalVariablesSingleton(self).setContentMap(key,"正在和"+key+"聊天\n");
    }
    String oldRecord = GlobalManager.getInstance().getGlobalVariablesSingleton(self).getContentMap(key);
    oldRecord += contents;
    GlobalManager.getInstance().getGlobalVariablesSingleton(self).updateContentMap(key,oldRecord);
    messageTextArea.setText(oldRecord);
    if (!getFriendListMap.containsKey(key)) {
        // 创建新的列表对象，确保每个键对应的值都是一个新的对象
        List<String> keyList = new ArrayList<>(sortedGroup);
        getFriendListMap.put(key, keyList);
        friendListModel.addElement(key);
    }
}
```

点击聊天列表时会获取当前聊天的信息，并自动勾选需要发送的对象。这样对于群聊和私人聊天均能满足需求：

```
/**
 * 点击右侧好友时的操作：更新历史记录
 */
1 usage   new *
public void showMessageWhenClicked(String friend){
    List<String>currMembers = getFriendListMap.get(friend);
    for (int i = 0; i < 100; ++ i) {
        onlineList[i].setSelected((currMembers.contains(String.valueOf(i))));
    }
    group = currMembers;
    messageTextArea.setText(GlobalManager.getInstance().getGlobalVariablesSingleton(self).getContentMap(group.toString()));
}
```

③发送消息

用户可以从单选框或聊天列表中发起聊天。若是新建聊天，首先判断是否有过往聊天记录。如果有则直接追加到该聊天中，并显示于聊天框；否则则新建一个聊天对话：

```
public void updateRecordWhenSend(String type,String contents) {
    sortedGroup.clear();
    sortedGroup.addAll(group);
    Collections.sort(sortedGroup);
    String key = sortedGroup.toString();

    if (GlobalManager.getInstance().getGlobalVariablesSingleton(self).getContentMap(key) == null) {
        GlobalManager.getInstance().getGlobalVariablesSingleton(self).setContentMap(key, "正在和" + key + "聊天\n");
    } //如果还没建立起会话过，新建一个

    String oldRecord = GlobalManager.getInstance().getGlobalVariablesSingleton(self).getContentMap(key);
    oldRecord += contents;
    //更新对话内容
    GlobalManager.getInstance().getGlobalVariablesSingleton(self).updateContentMap(key, oldRecord);
    messageTextArea.setText(oldRecord);

    if (!getFriendListMap.containsKey(key)) {
        // 创建新的列表对象，确保每个键对应的值都是一个新的对象
        List<String> keyList = new ArrayList<>(sortedGroup);
        getFriendListMap.put(key, keyList);
        friendListModel.addElement(key);
    }
}
```

首先保证发送的消息不能为空，且保证自己不与自己对话：

```
sendMessageButton.addActionListener(e -> {
    if(group.size()==0){
        JOptionPane.showMessageDialog(globalFrame, message: "Please Select friend(s)!", title: "error",JOptionPane.WARNING_MESSAGE);
    }
    else if(group.contains(self)){
        JOptionPane.showMessageDialog(globalFrame, message: "Can not chat with self!", title: "error",JOptionPane.WARNING_MESSAGE);
    }
    else {
        writeMessage(new Message(self, group, Message.MESSAGE_TYPE.MESSAGE, inputTextArea.getText()));
        updateRecordWhenSend( type: "TEXT", contents: self + ": " + inputTextArea.getText() + "\n");
        inputTextArea.setText("");
    }
}
);
```

设置发送方、接收方、消息类型和正文后，通过 ObjectInputStream 发送给服务器端。

```
3 usages   Xinkai Hu
private void writeMessage(Message message) {
    try {
        ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());
        outputStream.writeObject(message);
        outputStream.flush();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

④发送文件或图片

当用户点击“Send File”按钮后，利用 JFileChooser 获取系统的文件，当用户点击确定后，获取选中的文件在系统中的绝对路径用于传输处理：

```
private void showFileChooser(JFrame parent, Message.MESSAGE_TYPE type) {
    JFileChooser fileChooser = new JFileChooser();
    if (type == Message.MESSAGE_TYPE.IMAGE) {
        fileChooser.setFileFilter(new FileNameExtensionFilter( description: "image",
                ...extensions: "gif", "jpg", "jpeg", "jpg2", "png", "tif", "tiff", "bmp", "svg", "svgz", "webp", "ico"));
    }
    int result = fileChooser.showOpenDialog(parent);
    if (result == JFileChooser.APPROVE_OPTION) {
        File file = fileChooser.getSelectedFile();
        String path = file.getAbsolutePath();
```

并将文件对象转化为字节流，按字节的方式进行传输，使用 writeMessage 将数据传输给服务器端，同时记录文件或图片传输的情况：

```
int result = fileChooser.showOpenDialog(parent);
if (result == JFileChooser.APPROVE_OPTION) {
    File file = fileChooser.getSelectedFile();
    String path = file.getAbsolutePath();

    new Thread(() -> {
        try {
            byte[] bytes = Files.readAllBytes(Path.of(path));
            writeMessage(new Message(self, group, type, bytes));
            if (type == Message.MESSAGE_TYPE.FILE) {
                writeMessage(new Message(self, group, Message.MESSAGE_TYPE.MESSAGE,
                        content: "Send file at `" + path + "`."));
                updateRecordWhenSend( type: "FILE", contents: "You send file from `" + path + "`.\n");
            } else {
                writeMessage(new Message(self, group, Message.MESSAGE_TYPE.MESSAGE,
                        content: "Send image at `" + path + "`."));
                updateRecordWhenSend( type: "IMAGE", contents: "YOu send image from `" + path + "`.\n");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }).start();
}
```

⑤接收文件

客户端的文件接收依据消息类型进行判断，当有文件传输过来时，提供是否接收文件的选项(便于观察文件传输的过程).

```
case FILE -> {
    int result = JOptionPane.showConfirmDialog(
            globalFrame,
            message: "Confirm to receive the file?",
            title: "Information",
            JOptionPane.YES_NO_OPTION);
    if (result == JOptionPane.YES_OPTION) {
        messageTextArea.append("Received file.");
        byte[] bytes = (byte[]) message.content;
        saveFile(globalFrame, bytes,message.from);
        //-----
        group = message.to;
        group.add(message.from);
        group.remove(self);
        for (int i = 0; i < 100; ++ i) {
            onlineList[i].setSelected((group.contains(String.valueOf(i)) || Objects.equals(message.from, Str
        }
```

当用户选择接收文件时，将按字节流进行接收数据，接收完数据后又会将字节流转化为文件实体对象还原文件。在 saveFile 函数中，设置接收方可自行保存文件的路径，并保证路径不重复：

```
private void saveFile(JFrame parent, byte[] bytes, String send) {
    JFileChooser fileChooser = new JFileChooser();
    int result = fileChooser.showOpenDialog(parent);
    if (result == JFileChooser.APPROVE_OPTION) {
        File file = fileChooser.getSelectedFile();
        String path = file.getAbsolutePath();
        if (file.exists()) {
            JOptionPane.showMessageDialog(parent, message: "File path exists. Save file failed.");
        } else {
            try (FileOutputStream fileOutputStream = new FileOutputStream(path)) {
                fileOutputStream.write(bytes);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            updateRecordWhenReceive( contents: send + "send a file to you at "+path + "\n");
        }
    }
}
```

当用户选择拒收文件时，则关闭对应的文件写入流，不做文件保存即可。

当用户选择接收图片时时，将按字节流进行接收数据，接收完数据后又会将字节流转化为文件实体对象还原为图像，直接显示在界面：

```
        case IMAGE -> {
            byte[] bytes = (byte[]) message.content;
            File tmpFile = File.createTempFile( prefix: "tmpImage",  suffix: "tmpImage");
            String path = tmpFile.getAbsolutePath();
            try (FileOutputStream fileOutputStream = new FileOutputStream(path)) {
                fileOutputStream.write(bytes);
            }
            BufferedImage image = ImageIO.read(tmpFile);
            int height = image.getHeight();
            int width = image.getWidth();
            imageLabel.setIcon(new ImageIcon(image.getScaledInstance(
                    width: width * 128 / height,  height: 128, Image.SCALE_SMOOTH)));
            tmpFile.deleteOnExit();
        }
```

4.2.4 客户端关闭连接

当用户关闭对话窗口时，需向服务端发送“LOGOUT”类型的 message，当服务端接收后会更新对应的在线用户列表，再向在线客户端发送该列表。

4.3 全局变量的实现

GlobalManager 只被实例化一次，用于管理聊天记录。

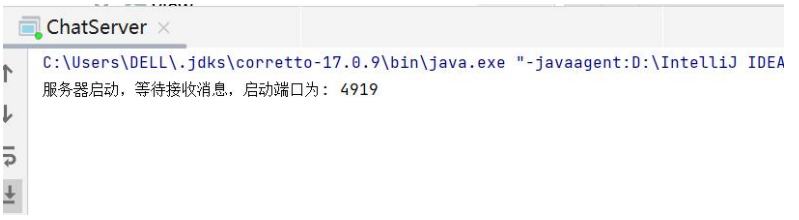
```java
public class GlobalManager {
    3 usages
    private static GlobalManager instance;
    3 usages
    private final Map<String, GlobalVariablesSingleton> instanceMap = new HashMap<>();

    1 usage
    private GlobalManager() {
        // 私有构造函数，防止直接实例化
    }

    11 usages
    public static GlobalManager getInstance() {
        if (instance == null) {
            instance = new GlobalManager();
        }
        return instance;
    }

    11 usages
    public GlobalVariablesSingleton getGlobalVariablesSingleton(String key) {
        if (!instanceMap.containsKey(key)) {
            GlobalVariablesSingleton newInstance = GlobalVariablesSingleton.getInstance();
            instanceMap.put(key, newInstance);
        }
        return instanceMap.get(key);
    }
```

GlobalVariablesSingleton 类一个实例代表一个用户，用于查询和修改该用户用其他用户的聊天消息。

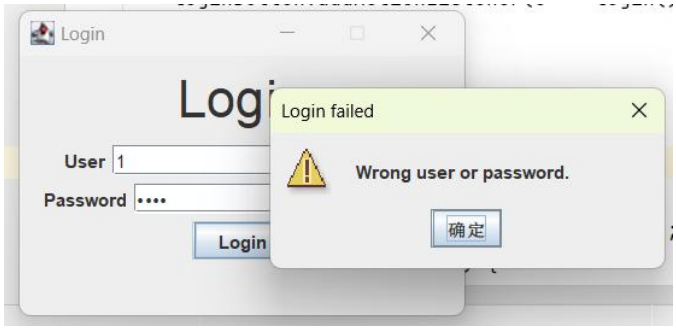
```java
public class GlobalVariablesSingleton {
    5 usages
    private static GlobalVariablesSingleton instance;
    3 usages
    private final Map<String, String> contentMap = new HashMap<>();
    6 usages
    public String getContentMap(String key) { return contentMap.get(key); }
    3 usages
    public void setContentMap(String s1, String s2) { instance.contentMap.put(s1, s2); }
    2 usages
    public void updateContentMap(String s1, String s2) { instance.contentMap.replace(s1, s2); }
    1 usage
    public static GlobalVariablesSingleton getInstance() {
        if (instance == null) {
            instance = new GlobalVariablesSingleton();
        }
        return instance;
    }
}
```

四、实验结果及分析和（或）源程序调试过程

1. 启动服务器端



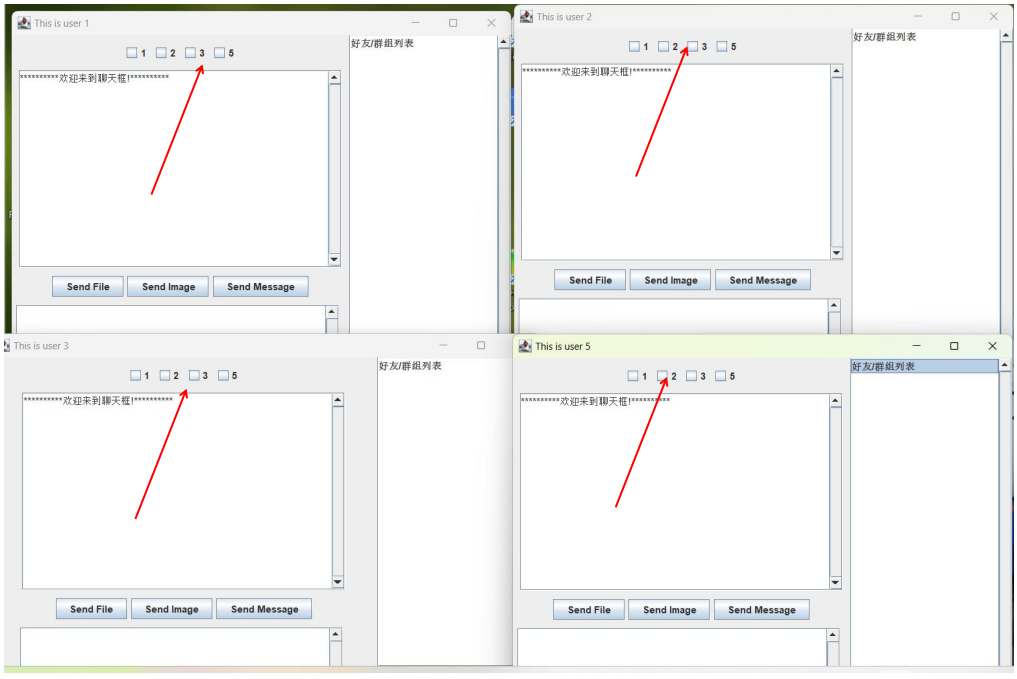
2. 用户登录



以登录用户 1，2，3，5 为例。当 1，2，3，5 登陆成功后服务器端会显示各个用户的操作信息。

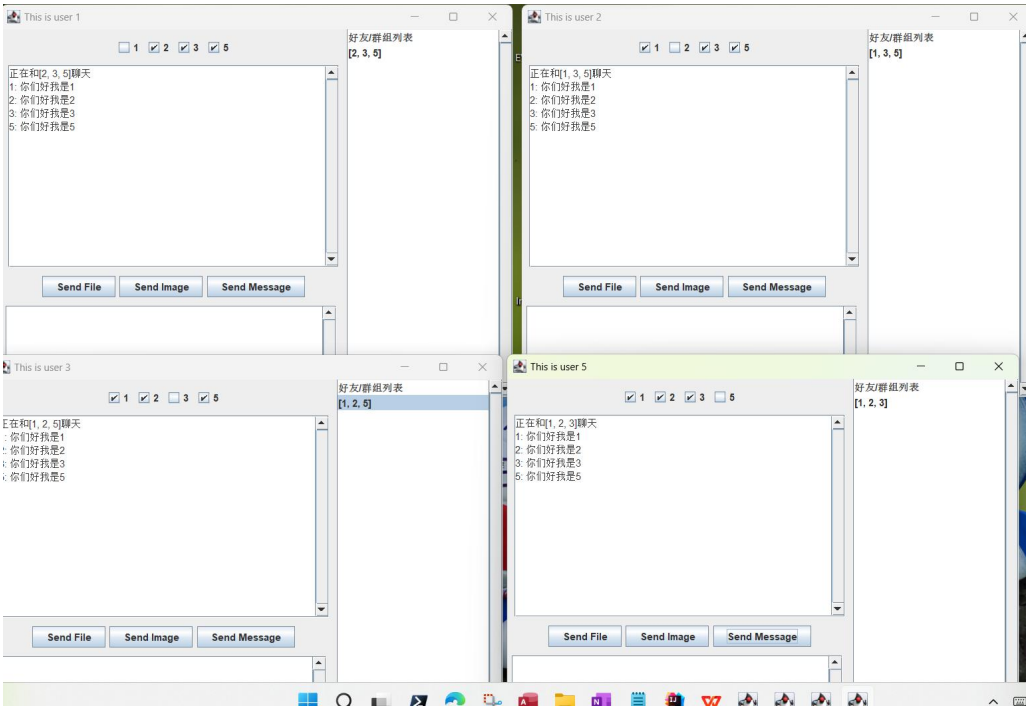
```
服务器启动，等待接收消息，启动端口为： 4919
MESSAGE FROM `1` TO `null` TYPE `LOGIN` CONTENT `null`.
Online user: `[1]`.
MESSAGE FROM `2` TO `null` TYPE `LOGIN` CONTENT `null`.
Online user: `[1, 2]`.
MESSAGE FROM `3` TO `null` TYPE `LOGIN` CONTENT `null`.
Online user: `[1, 2, 3]`.
MESSAGE FROM `5` TO `null` TYPE `LOGIN` CONTENT `null`.
Online user: `[1, 2, 3, 5]`.
```

各个客户端也会实时显示在线用户列表(各用户信息在左上角)：



3. 群发消息

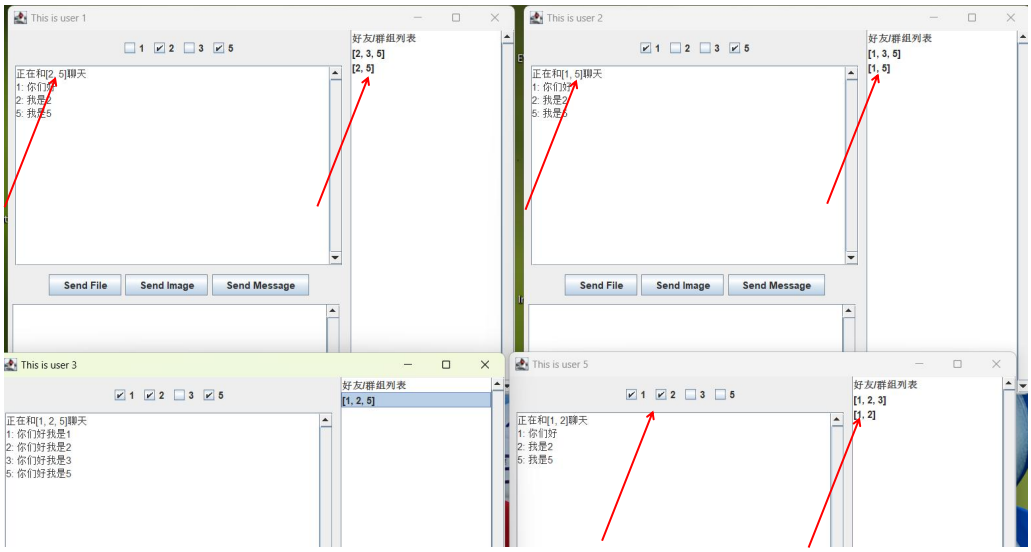
每位用户群发消息后得到的效果如下：



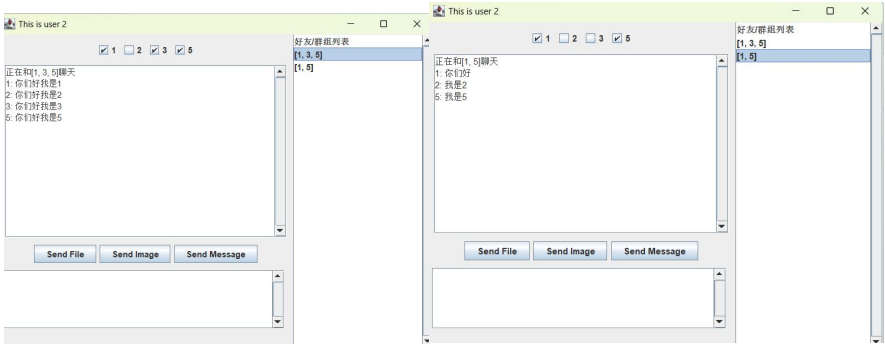
服务端也会接收到消息：

```
MESSAGE FROM `1` TO `[2, 3, 5]` TYPE `MESSAGE` CONTENT `你们好我是1`.
Online user: `[1, 2, 3, 5]`.
MESSAGE FROM `2` TO `[3, 5, 1]` TYPE `MESSAGE` CONTENT `你们好我是2`.
Online user: `[1, 2, 3, 5]`.
MESSAGE FROM `3` TO `[5, 1, 2]` TYPE `MESSAGE` CONTENT `你们好我是3`.
Online user: `[1, 2, 3, 5]`.
MESSAGE FROM `5` TO `[1, 2, 3]` TYPE `MESSAGE` CONTENT `你们好我是5`.
Online user: `[1, 2, 3, 5]`.
```

如果要自己新建群，比如 1，2，5 新建群聊，只需 1，2，5 其中任意一位选择发送即可。并且每位用户的聊天框均会保持在右侧，显示聊天记录。

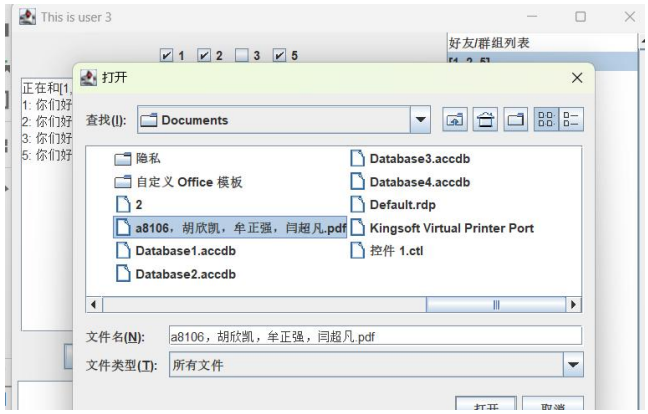


当点回初始群聊(以 2 为例，均会显示出独立的聊天记录)

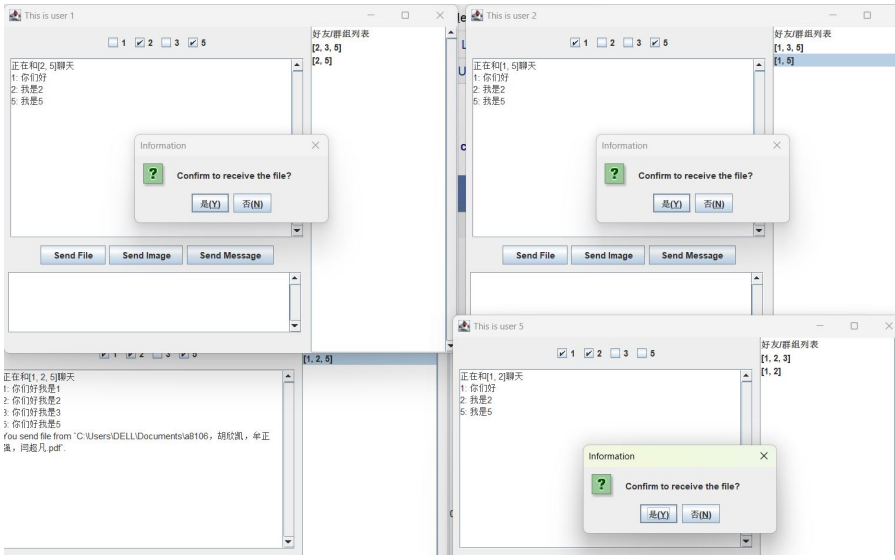


4. **群发文件**

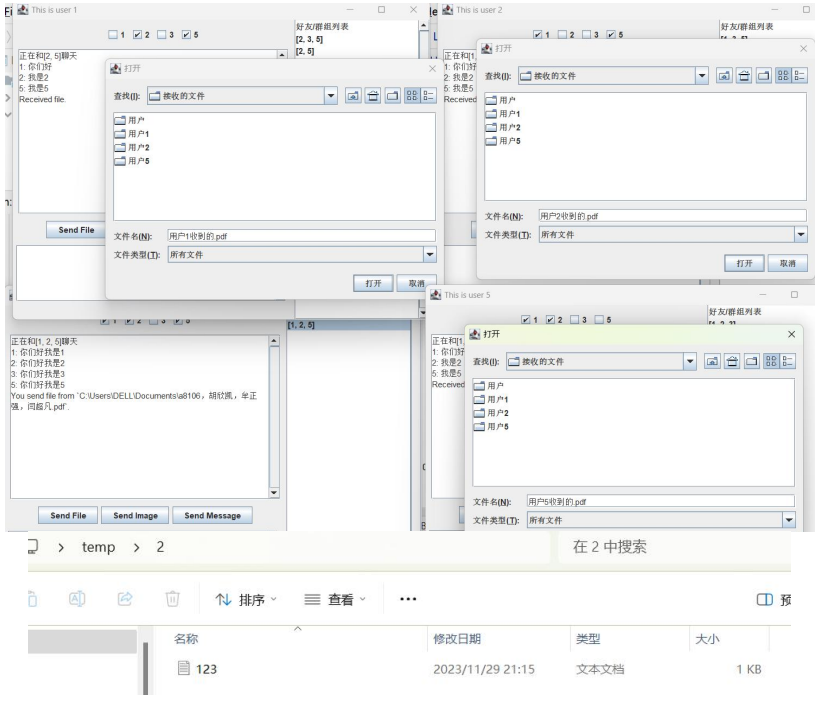
以用户 3 向 1，2，5 文件为例，首先选择文件：



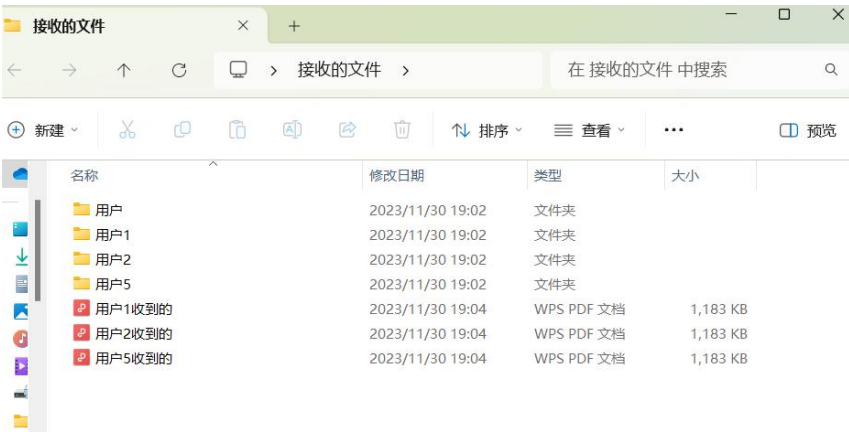
用户 1，2，5 收到接收提示：



此时用户 1，2，5 保存接收路径：



此时用户 1，2，5 均收到了文件：

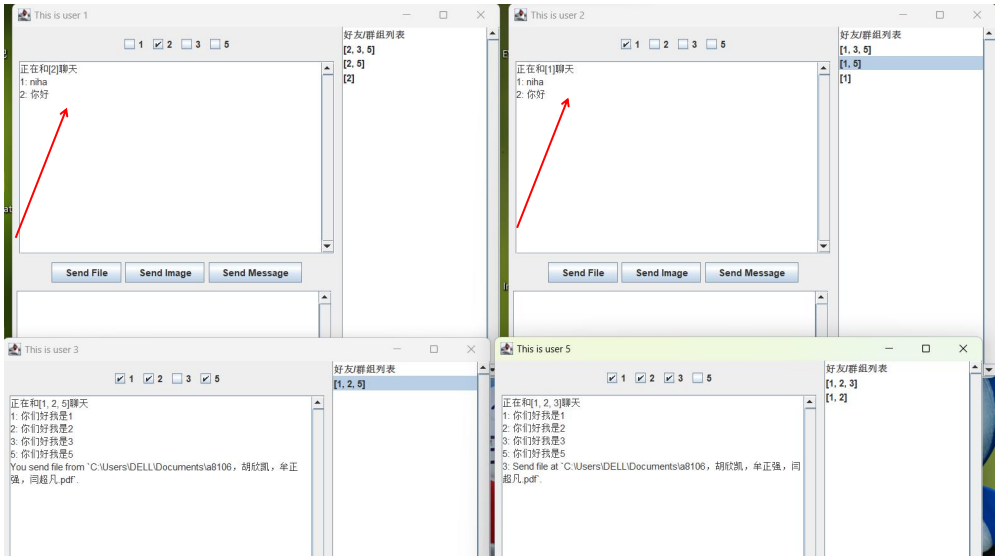


服务端也会有相应的输出：

```
MESSAGE FROM `3` TO `[1, 2, 5]` TYPE `FILE` CONTENT `[B@78b016c9`.
Online user: `[1, 2, 3, 5]`.
MESSAGE FROM `3` TO `[1, 2, 5]` TYPE `MESSAGE` CONTENT `Send file at `C:\Users\DELL\Documents\a8106，胡欣凯，牟正强，闫超凡.pdf`.`.
Online user: `[1, 2, 3, 5]`.
```

5. 私发消息

以用户 1，2 为例，只有 1 和 2 在通信，而 3，5 没有参与到其中。



私发图片：如 1 向 2 发送了一张图片.