廣東工業大學

计算机网络课程设计

课程名称 计算机网络

题目名称 基于 P2P 的局域网即时聊天系统

学生学院 计算机学院

专业班级 软件工程 4 班

学　　号 3115005372

学生姓名 杨宇杰

指导教师 姜文超

2017　年 6 月 18 日

目录

基于 P2P 的局域网即时通信系统

一、设计要求

1．掌握 P2P 原理。

2．实现一个图形用户界面局域网内的消息系统。

3．功能：建立一个局域网内的简单的 P2P 消息系统，程序既是服务器又是客户，服务器端口（自拟服务器端口号并选定）。

3.1 用户注册及对等方列表的获取：对等方 A 启动后，用户设置自己的信息（用户名，所在组）；扫描网段中在线的对等方（服务器端口打开），向所有在线对等方的服务端口发送消息，接收方接收到消息后，把对等方 A 加入到自己的用户列表中，并发应答消息；对等方 A 把回应消息的其它对等方加入用户列表。双方交换的消息格式自己根据需要定义，至少包括用户名、IP 地址。

3.2 发送消息和文件：用户在列表中选择用户，与用户建立 TCP 连接，发送文件或消息。

4．用户界面：界面上包括对等方列表；消息显示列表；消息输入框；文件传输进程显示及操作按钮或菜单。

二、软件开发工具及运行环境

1、软件开发工具：

a、编程语言：Java

b、开发环境：Eclipse

c、JDK 版本：1.8

d、操作系统：windows 10

2、运行环境：

操作系统无关性，Windows、Linux、Mac OS X 下安装了 Java 的运行环境 JRE 即可运行。

三、程序开发的基础知识

1．学习 Socket 和 TCP 的基本原理和通信机制

1.1 TCP 连接

电脑能够使用联网功能是因为电脑底层实现了 TCP/IP 协议，可以使电脑终端通过无线网络建立 TCP 连接。TCP 协议可以对上层网络提供接口，使上层网络数据的传输建立在“无差别”的网络之上。

建立起一个 TCP 连接需要经过“三次握手”：

第一次握手：客户端发送 syn 包(syn=j)到服务器，并进入 SYN_SEND 状态，等待服务器确认；

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=j+1），同时自己也发送一个 SYN 包（syn=k），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN＋ACK 包，向服务器发送确认包 ACK(ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP 连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开 TCP 连接的请求，断开过程需要经过“四次握手”（过程就不细写了，就是服务器和客户端交互，最终确定断开）

1.2 SOCKET 原理

1.2.1 套接字（socket）概念

套接字（socket）是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。

应用层通过传输层进行数据通信时，TCP 会遇到同时为多个应用程序进程提供并发服务的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP / IP 协议交互提供了套接字(Socket)接口。应用层可以和传输层通过 Socket 接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

1.2.2 建立 socket 连接

建立 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ，另一个运行于服务器端，称为 ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

1.3 SOCKET 连接与 TCP 连接

创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（TCP 或 UDP），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。

2．功能设计和界面设计

2.1 主要运用知识

a. Java socket 编程

b. Java GUI 编程

c. Java 继承和事件绑定

d. Java 异常与捕获

四、总体设计

1.设计思路

1.1 经典的 TCP 通信服务器客户端架构

服务器有一个服务器等待用户连接的线程，该线程循环等待客户端的 TCP 连接请求。一旦用 ServerSocket.accept()捕捉到了连接请求，就为该 TCP 连接分配一个客户服务线程，通过该消息传递线程服务器与客户端通信。服务器发送消息通过该客户服务线程的方法在主线程完成，而接收消息全部在客户服务线程中循环接收并处理。

客户机能发起一个向服务器的 socket 连接请求，一旦收到服务器成功响应连接请求，客户机便为这个 socket 分配一个消息接收线程，否则关闭该 socket。和服务器任务分配类似，发送消息作为非常用方法在主线程中完成，而接收消息在消息接收线程中不停刷新并作相应处理。

1.2 统一 ASCII 码级文本传输协议

为了实现客户机对服务器命令的响应、服务器对客户机需求的解读以及客户机与客户机之间的消息传递，我为服务器和客户端之间通信定义了一组文本传输协议。协议属于变长文本传输协议，用@作为各字段分隔符，所有消息的首节一定是消息类型，方便解析。协议定义了以下按发送方分类的消息格式：

服务器信息格式：

[1] 聊天消息	**MSG**	@	**to**	@	**from**	@	**content**
			ALL		SERVER		xxx
			ALL		USER		xxx
			USER		USER		xxx
[2] 登陆状态	**LOGIN**	@	**status**	@	**content**		
			SUCCESS		xxx		
			FAIL		xxx		
[3] 用户操作	**USER @**	**type**	@	**other**			
		ADD		USER			
		DELETE		USER			
		LIST		number	{@	USER}+	
[4] 错误信息	**ERROR**	@	**TYPE**				
			xxx				
[5] 服务器关闭	**CLOSE**						

客户机消息格式：

[1] 聊天消息	**MSG**	**@**	**to**	**@**	**from**	**@** **content**
			ALL		USER	xxx
			USER		USER	xxx
[2] 登陆通知	**LOGIN**	**@**	**USER**			
[3] 登出通知	**LOGOUT**					

注：　[1] 其中USER是USER_NAME%USER_IPADDR的简写。
　　[2] 消息变长，字段数也变长，所有解析工作以来自于对Message_Type的判断

1.3 MVC 分层模式

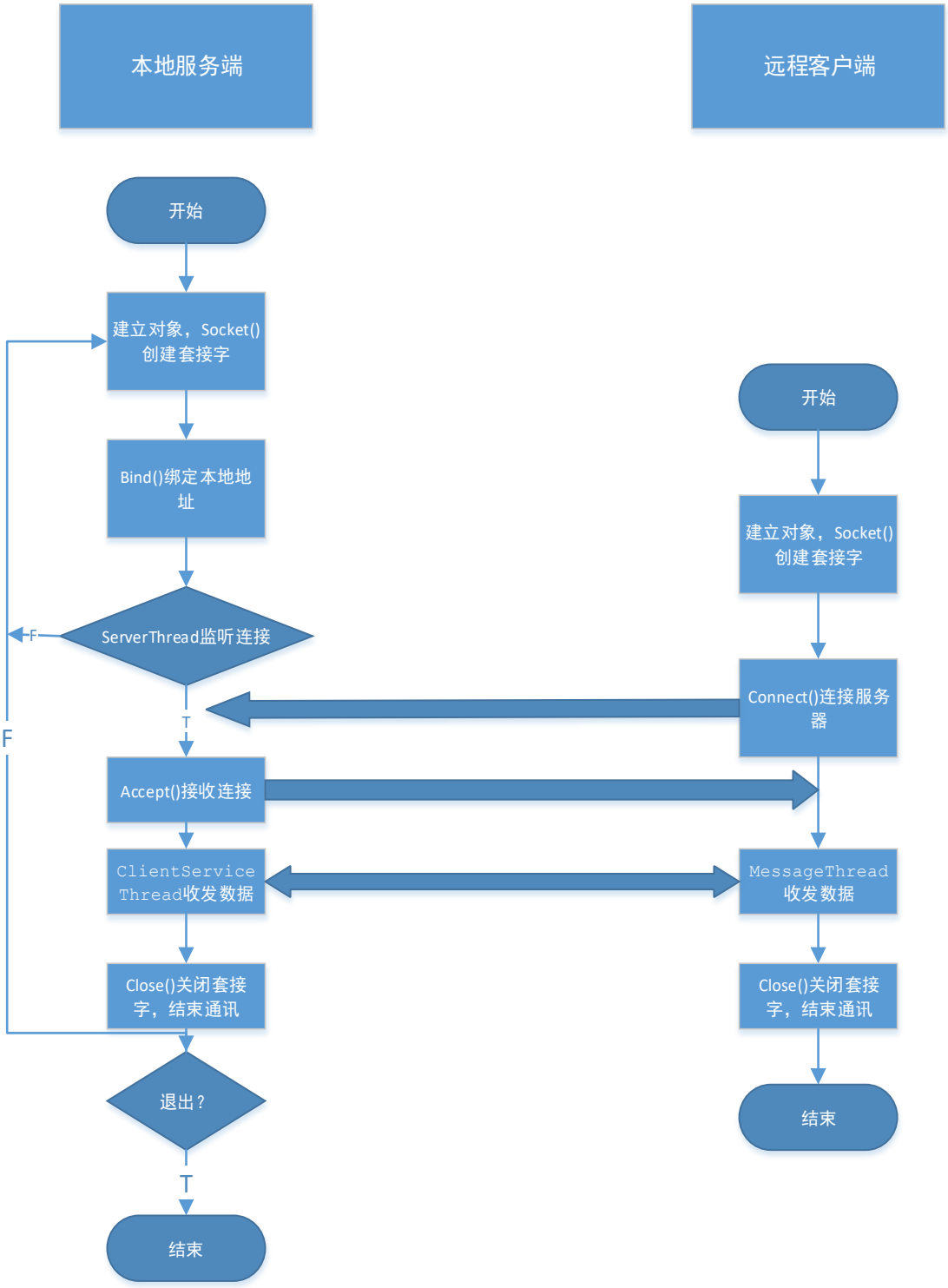
Model-View-Controller 是经典的应用程序开发设计模式，它讲究数据管理、界面显示和用户交互、程序维护管理分别封装在 MVC 三种类中，够成松耦合关系。本次课程设计中我也利用 MVC 的设计思路，独立了 Model 类 User 用于保存客户机用户信息，DefaultListModel 模型类用于储存在线用户队列；将 View 单独放在一个包中，Controller 监听用户操作事件，反映给 Model 类处理并在 View 中更新。

MVC 的思想即是 M 和 V 之间不要直接产生联系，业务逻辑均封装在 MC 中，而 V 仅仅负责显示。本实验为 V 类绑定了各自的 Listener 监听用户操作，在 C 中完成业务逻辑处理，保存并更新 User 和 DefaultListModel，最后再显示到 UI 界面上。

1.4 concurrentHashMap 管理线程队列和用户列表

concurrentHashMap 是 java.util.concurrent 包中定义的多线程安全的哈希表，利用哈希表管理线程队列和用户列表可以快速索引，多线程安全也保证了多个用户服务线程之间共享资源的数据一致性。

2. 程序流程图


本地服务端
远程客户端
开始
建立对象，Socket()创建套接字
Bind()绑定本地地址
ServerThread监听连接
F
T
F
Accept()接收连接
ClientServiceThread收发数据
Close()关闭套接字，结束通讯
退出？
T
结束
开始
建立对象，Socket()创建套接字
Connect()连接服务器
MessageThread收发数据
Close()关闭套接字，结束通讯
结束

3.关键数据结构

3.1 User 模型类

（1）构造方法：

有两个，一个是用独立的 name 和 IP 实例化一个 User，另一个是用 name%IP 拼接而成的字符串实例化 User

（2）只读字段

name 和 ipAddr 均是 private 的，给他们配置一个只读的 getter

（3）description()用户描述

返回 name%IP 拼接而成的字符串，用以代表一个独立的用户

3.2 ServerView 类

（1）UI 相关的方法

构造函数中的 initUI()大部分是设置 UI 界面，其中用到了 GridLayout 和 BorderLayout。用 serviceUISetting(false)把所有连接状态才起作用的 button 和 textField 全部关闭了（false 改为 true 开启他们，并关闭所有设置相关的 button 和 textField）。

3.3 ServerMain 类

（1）startServer()开启服务器方法———startButton 绑定

先检查 maxClientNum 和 port 的合法输入，如果不合法弹出出错窗口并退出。

接着初始化管理客户服务线程队列的并发哈希表 clientServiceThreads，初始化监听客户机连接请求的 serverSocket，并且初始化和开启一个监听连接请求的线程。最后有一些差错处理以及服务器 log 日志。

（2）请求监听线程 ServerThread 类

isRunning 作为线程运行标志位控制线程存活，线程 start 后会调用的函数 run()里完成了监听逻辑。如果开启则一直循环，serverSocket.accept()是阻塞的，线程不会运行直到有其他线程/进程向其请求 Socket 连接。这也是我下面提到的一个 bug 的原因：accept()阻塞了线程它一直在等待，仅仅用标志位来结束线程并不能使之跳出阻塞状态（还没有循环到下一次 while 的判断），因此我在 closeThread()中强行关闭 serverSocket 会报出一个异常！

收到连接请求后 accept()返回一个 socket，这个 socket 用于和请求连接的客户机通信。至此时 TCP 建立连接 3 次握手已经完成，全部被 serverSocket 类封装起来了。获取了通信 socket 之后检查服务器在线人数是否已满，向客户机发送一个登陆成功或失败的消息。若在线人数未满连接成功，则为客户机分配一个 clientServiceThread 线程专门用于发送和接受客户机的 TCP 包。

（3）监听客户机消息的 ClientServiceThread 线程类

该类比较庞大，我挑重点介绍

[1] 关键字段

```
private Socket socket;

private User user;

private BufferedReader reader;

private PrintWriter writer;

private boolean isRunning;
```

分别保存了通信 socket、当前连接用户 Model、绑定在 socket 输入流上的 BufferedReader、绑定在 socket 输出流上的 PrintWriter 以及线程运行控制标志位 isRunning。reader 用来读取客户机消息输入，readLine 方法也是阻塞的，直到客户机有消息发送过来。writer 有一个写缓冲区，有 flush()函数强制发送消息并刷新缓冲区，我把写消息封装在 sendMessage(String)中。

［2］初始化

初始化中先绑定 reader 和 writer 到 socket 响应流，在判断用户 socket 请求发送的消息格式是否正确（不正确线程将不能执行）。接着向所有已上线的用户通知一遍这个新用户上线了，发送通知需要遍历整个服务线程队列并发送文本传输协议中定义的各式的通知。注意到这时候该服务线程并没有加入到服务线程队列中，是在初始化完成之后加入的。

通知了其他用户这个新客户机上线后，再告诉该客户机现在已经有哪些用户在线了，这也是用协议中的格式发送通知即可。这里用到了 StringBuffer 类，多字符串连接时该类比 String 的+的效率要高。

［3］线程 run

收到客户机消息后判断消息类型，若是 LOGOUT 通知客户机下线，则向所有其他客户端进程发送该用户下线的信息，并删除 model 类里的该用户对象和线程队列里的该线程。

如果是消息则交与 dispatchMessage(String)方法专门分发消息。

［4］分发消息方法 dispatchMessage(String)

该方法解析 MSG 类消息的 to 字段，根据 to 字段选择是将消息发给特定用户还是直接群发。发给特定用户的话根据 to 字段（userDescription）做索引，快速从服务线程队列找出服务该用户客户机的线程来发送信息。

［5］其他

绑定时间如 stopServer 关闭服务器和 sendAll 群发消息都比较直白便省略介绍，主要需要注意一下其中的差错控制。关闭服务器还需要更新 UI 控制逻辑。

［6］说明

ServerMain 类虽然通过 ClientServiceThread 里的 writer 发送消息，并且也是调用封装在这个 Thread 内部类中的，但是调用 writer 来 sendMessage 并不是一定在该线程内完成的（该线程内指的是 run()里的 while 循环内部），sendMessage 是非阻塞的我们没有必要专门在线程中执行。ClientServiceThread 主要工作是收听各个客户端向服务器发送的消息。

3.4 ClientView 类

Client 和 Server 稍微有点不一样，只有一个辅助线程 MessageThread 用于接收服务器消息。由于只需要绑定在一个 socket 上，所以 writer 和 reader 只有一个，是直接属于 Client 实例的字段。

［1］UI 相关方法

构造函数里的 init 和 Server 中几乎完全一样，这部分属于代码复用。注意需要多绑定一个监听器：

javax.swing.event.ListSelectionListener 类用来监听用户选择 JList 框里的条目，JList 框里固定一个所有人的项（点击选中表示消息发送给所有人，默认发送给所有人，目标对象下线后也是自动把对象转变成所有人），其他则是在线用户。点击这些列表项时触发一个选择事件，通过判断 index 来判断用户的选择，并更新模型记录 sendTarget 和 UI 中 messageToLabel 显示的 text。

3.5 ClientMain 类

［1］connect 连接到服务器

差错检测这里没有判断 IP 地址合法性，判断也不是很麻烦。用户输入合法时，根据服务器 IP 地址和端口实例化一个 socket，这个 socket 用于将来和服务器通信。

获取客户机本地 IP 地址并用这个 IP 地址实例化，通过 socket 给服务器发送一条自己用户信息（name 和 IP）的消息表示请求。发送完毕后立即开启 MessageThread 等待服务器的回应。

［2］MessageThread 接受服务器消息线程

reader.readLine()阻塞读取服务器消息。一直忘记介绍 StringTokenizer 类，这里说明一下。StringTokenizer 类通过一个 String 和一个分割字符串实例或一个 tokenizer，通过分割得到一系列记号流通过 tokenizer.nextToken()获取这些记号字符串。不难发现其作用和 String.split(String)一样也是做字符串分割，但是其效率显著优于 split 方法（百度搜索两者比较会有较详细的性能分析）。

根据 tokenizer 返回的记号流我们来判断消息类型，

服务器关闭：向服务器发送一个下线信息，关闭 socket, write 和 read，清空记录 Model，最后退出线程。

服务器错误：log 错误类型，啥也不干进入下一轮循环。

登陆信息，

成功：log 成功，进入下一轮循环。

失败：log 失败，关闭 socket, write 和 read，清空记录 Model，最后退出线程。

4.关键性代码

```
private class ClientServiceThread extends Thread {
        private Socket socket;
        private User user;
        private BufferedReader reader;
        private PrintWriter writer;
        private boolean isRunning;

        private synchronized boolean init() {
            try {
                reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                writer = new PrintWriter(socket.getOutputStream());

                String info = reader.readLine();
                StringTokenizer tokenizer = new StringTokenizer(info, "@");
                String type = tokenizer.nextToken();
                if (!type.equals("LOGIN")) {
                    sendMessage("ERROR@MESSAGE_TYPE");
                    return false;
                }
                user = new User(tokenizer.nextToken());
                sendMessage("LOGIN@SUCCESS@" + user.description() + "与服务器连接
成功！");
                int clientNum = clientServiceThreads.size();
                if (clientNum > 0) {
                    //告诉该客户端还有谁在线
                    StringBuffer buffer = new StringBuffer();
                    buffer.append("@");
                    for (Map.Entry<String, ClientServiceThread> entry :
clientServiceThreads.entrySet()) {
```

```
                    ClientServiceThread serviceThread = entry.getValue();
                    buffer.append(serviceThread.getUser().description() +
"@");
                    //告诉其他用户此用户在线
                    serviceThread.sendMessage("USER@ADD@" +
user.description());
                }
                sendMessage("USER@LIST@" + clientNum + buffer.toString());
            }
            return true;
        } catch(Exception e) {
            e.printStackTrace();
            return false;
        }
    }
    public ClientServiceThread(Socket socket) {
        this.socket = socket;
        this.isRunning = init();
        if (!this.isRunning) {
            logMessage("服务线程开启失败！");
        }
    }
    public void run() {
        while (isRunning) {
            try {
                String message = reader.readLine();
               // System.out.println("recieve message: " + message);
                if (message.equals("LOGOUT")) {
                    logMessage(user.description() + "下线...");

                    int clientNum = clientServiceThreads.size();

                    //告诉其他用户该用户已经下线
                    for (Map.Entry<String, ClientServiceThread> entry :
clientServiceThreads.entrySet()) {
                        entry.getValue().sendMessage("USER@DELETE@" +
user.description());
                    }
                    //移除该用户以及服务器线程
                    listModel.removeElement(user.getName());
                    clientServiceThreads.remove(user.description());

                    close();
                    return;
```

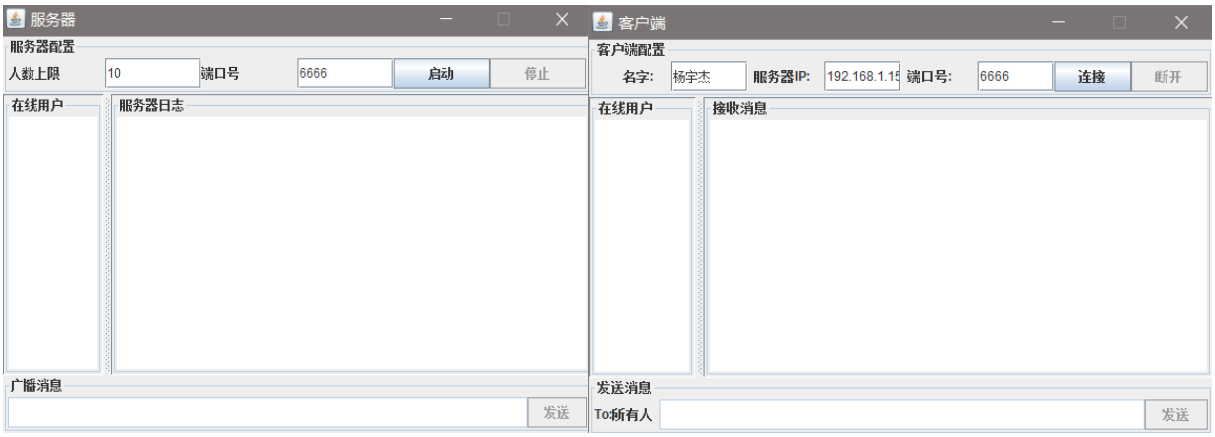
```
                } else {  //发送消息
                    dispatchMessage(message);
                }
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
    public void dispatchMessage(String message) {
        StringTokenizer tokenizer = new StringTokenizer(message, "@");
        String type = tokenizer.nextToken();
        if (!type.equals("MSG")) {
            sendMessage("ERROR@MESSAGE_TYPE");
            return;
        }
        String to = tokenizer.nextToken();
        String from = tokenizer.nextToken();
        String content = tokenizer.nextToken();
        logMessage(from + "->" + to + ": " + content);
        if (to.equals("ALL")) {
            //send to everyone
            for (Map.Entry<String, ClientServiceThread> entry :
clientServiceThreads.entrySet()) {
                entry.getValue().sendMessage(message);
            }
        } else {
            //发送给某一个人
            if (clientServiceThreads.containsKey(to)) {
                clientServiceThreads.get(to).sendMessage(message);
            } else {
                sendMessage("ERROR@INVALID_USER");
            }
        }
    }
    public void close() throws IOException {
        this.isRunning = false;
        this.reader.close();
        this.writer.close();
        this.socket.close();

    }
    public void sendMessage(String message) {
        writer.println(message);
        writer.flush();
```

```
    }
    public User getUser() {
        return user;
    }
    }
```

五、测试

1.运行 ServerMain 和 ClientMain

本程序支持单服务器多客户端，我打开了 1 个 Server 和 2 个 Client 作为演示。进程开启后界面如下：



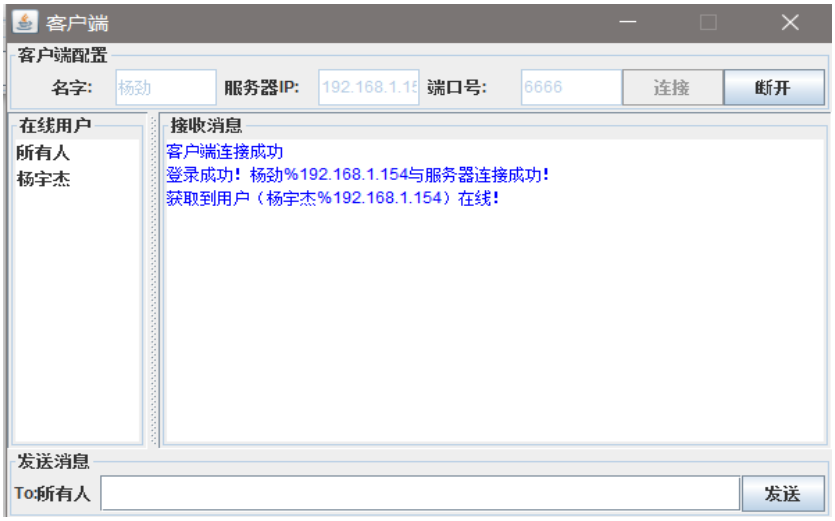
服务器和客户端在未开启 / 连接的状态下无法发送消息只能修改配置，而在开启 / 连接状态下无法修改配置只能发送消息，UI 逻辑均由开启 / 连接状态决定。

服务器配置中可以修改监听端口和人数上限，开启后接收所有来自客户端的消息，模仿解析包动作进行转发，服务器可以群发消息。

客户端可以设置自己的名字、服务器 IP 和服务器端口（默认均是 192.168.1.154 本地 IP 和端口 6666）。客户端连接后可以在在线用户列表中选择所有人或其他在线用户发送消息。其他用户上下线信息在在线用户列表和聊天消息框中都有提示。

2. 开启服务器并登陆客户端

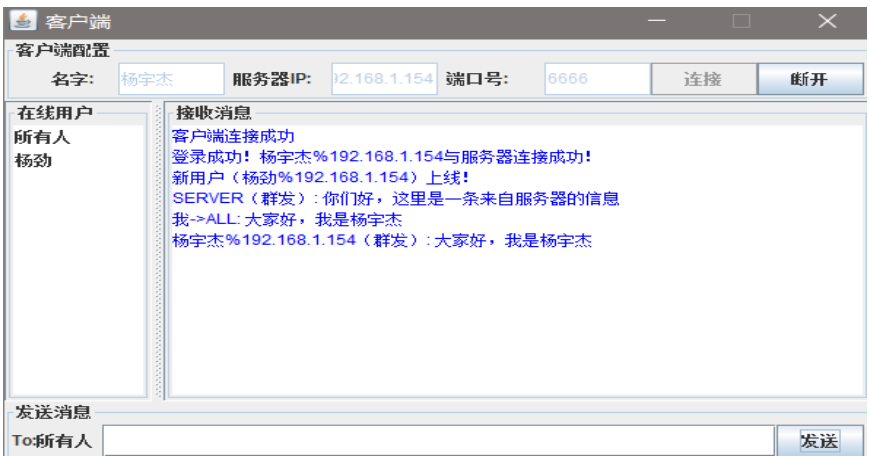
下图是第二位登陆的用户杨劲登录时，从服务器接受其他用户，显示在左侧在线用户列表中。



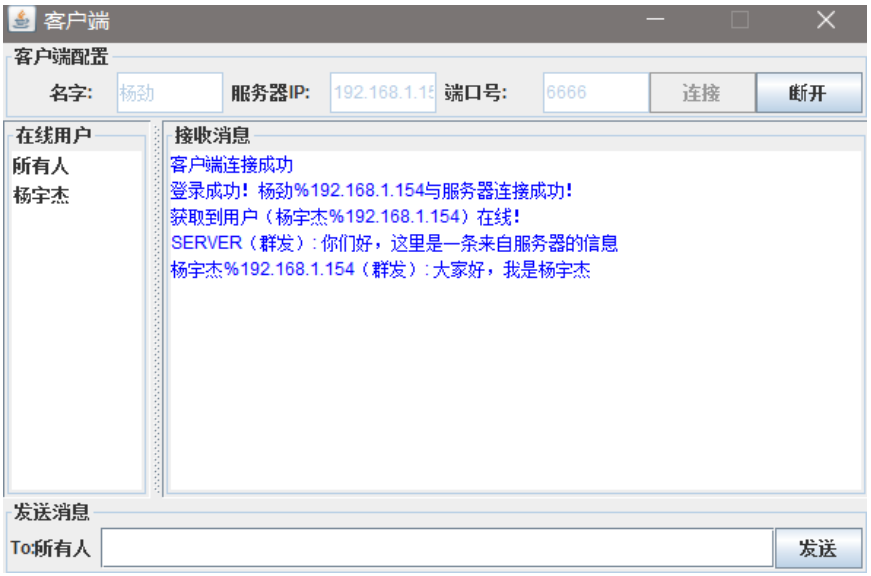
3.服务器群发消息

4. 客户端群发消息

客户端 杨宇杰 目标选择 所有人，并发送消息：

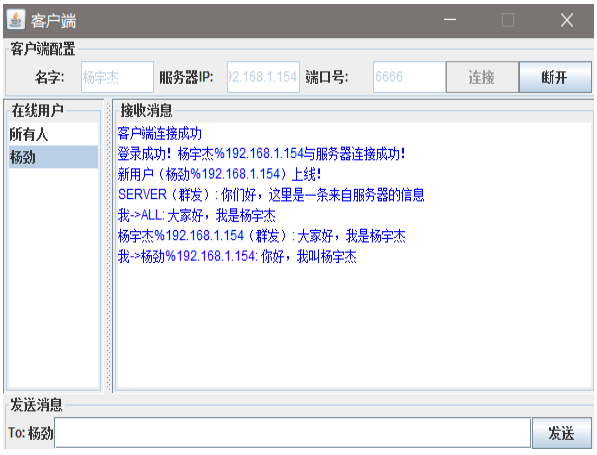


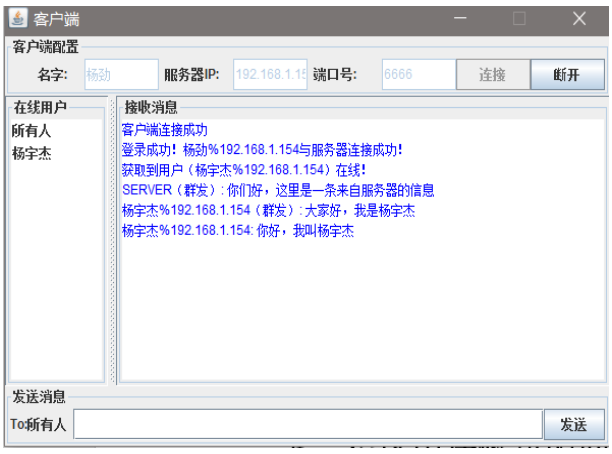
客户端 杨劲 收到了消息，并标记为群发：



5. 客户端发送私聊消息：

客户端 杨宇杰 和客户端 杨劲 私聊：



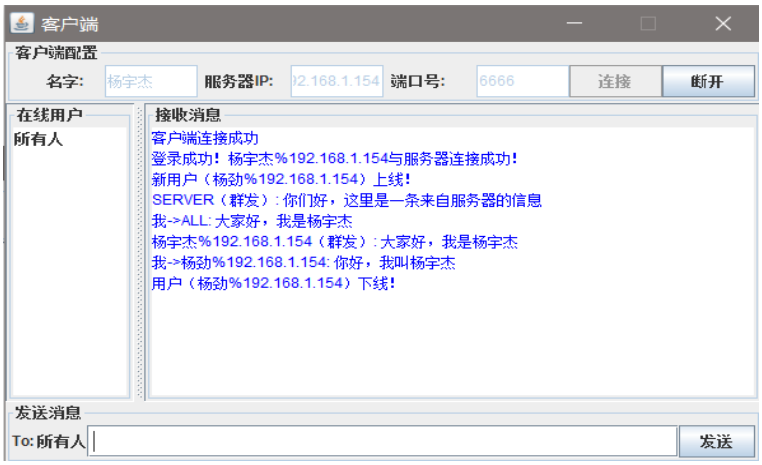


6. 客户端下线

客户端下线时所有在线用户收到其下线消息，在线列表中不再出现此用户：

客户端 杨劲 下线，通知服务器，服务器转发其下线消息：

客户端 杨宇杰 收到 杨劲 的下线通知，左侧在线列表中已经没有 杨劲 了：



六、开发过程中遇到的问题及解决办法

本次完成这个课程设计还是花了不少精力和时间，由于对于 Java 网络编程这一块并不是很擅长，开发过程中老是遇到各种大大小小的问题，例如一开始不太清楚 tokenizer.nextToken()的意思以及用法，在参考了一些书籍以及百度各种关于网络编程的文章，才得以顺利完成本次课设。

七、程序中待解决的问题及改进方向

程序可能还存在不少小 bug，后期有时间再进行修复。目前发现的一个 Bug 是服务器关闭时，我关了服务器接收 Socket 请求的线程并 close 了该 ServerSocket，但是该线程仍然继续执行了一次 ServerSocket.accept()。我尝试用了 synchronized 方法并判断 ServerSocket 是否关闭，但这个异常还是会出现。我捕捉了该异常，仅仅 printStackTrace 而没有做其他错误处理，幸运的是这小 Bug 并不影响服务器关闭，所有客户端都能正确的接收服务器关闭的消息。希望以后有时间能把这个错误给改过来。