



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 5](#_Toc485901929)

[一、设计要求 5](#_Toc485901930)

[二、软件开发工具及运行环境 5](#_Toc485901931)

[1、软件开发工具： 5](#_Toc485901932)

[2、运行环境： 5](#_Toc485901933)

[三、程序开发的基础知识 6](#_Toc485901934)

[1．学习Socket和TCP的基本原理和通信机制 6](#_Toc485901935)

[2．功能设计和界面设计 7](#_Toc485901936)

[四、总体设计 8](#_Toc485901937)

[1.设计思路 8](#_Toc485901938)

[2. 程序流程图 10](#_Toc485901939)

[3.关键数据结构 11](#_Toc485901940)

[4.关键性代码 15](#_Toc485901941)

[五、测试 18](#_Toc485901942)

[1.运行ServerMain和ClientMain 18](#_Toc485901943)

[2. 开启服务器并登陆客户端 18](#_Toc485901944)

[3.服务器群发消息 19](#_Toc485901945)

[4. 客户端群发消息 19](#_Toc485901946)

[5. 客户端发送私聊消息： 20](#_Toc485901947)

[6. 客户端下线 20](#_Toc485901948)

[六、开发过程中遇到的问题及解决办法 21](#_Toc485901949)

[七、程序中待解决的问题及改进方向 21](#_Toc485901950)

# 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

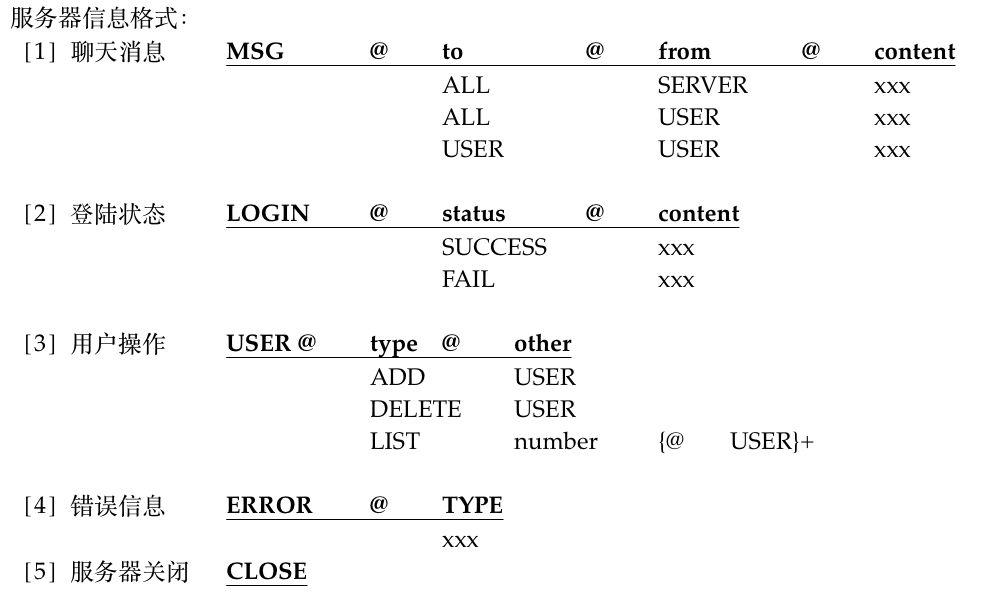
#### 1.1经典的TCP通信服务器客户端架构

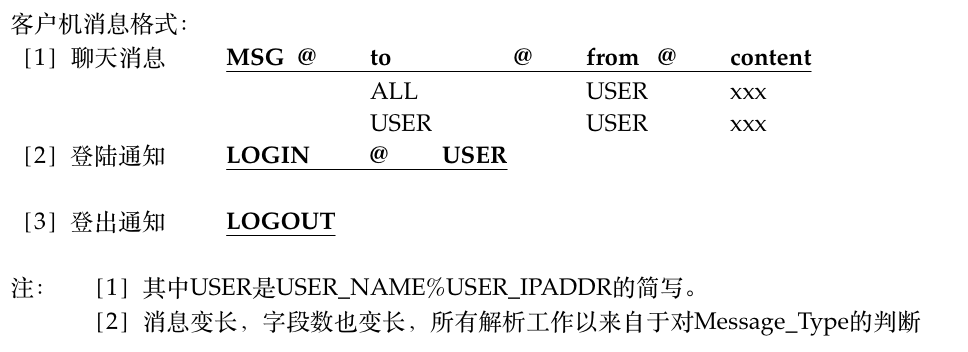
服务器有一个服务器等待用户连接的线程，该线程循环等待客户端的TCP连接请求。一旦用ServerSocket.accept()捕捉到了连接请求，就为该TCP连接分配一个客户服务线程，通过该消息传递线程服务器与客户端通信。服务器发送消息通过该客户服务线程的方法在主线程完成，而接收消息全部在客户服务线程中循环接收并处理。

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：





#### 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



### 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;

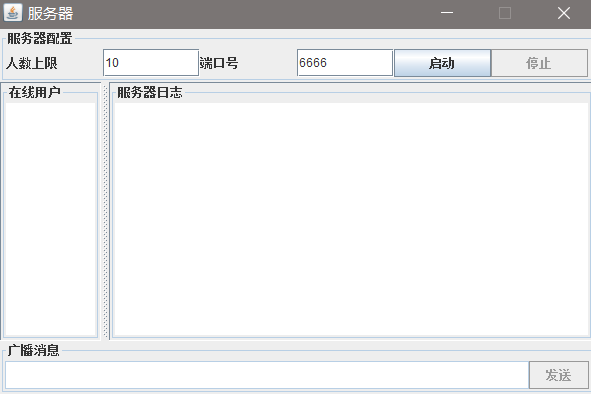
}

}

## 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：



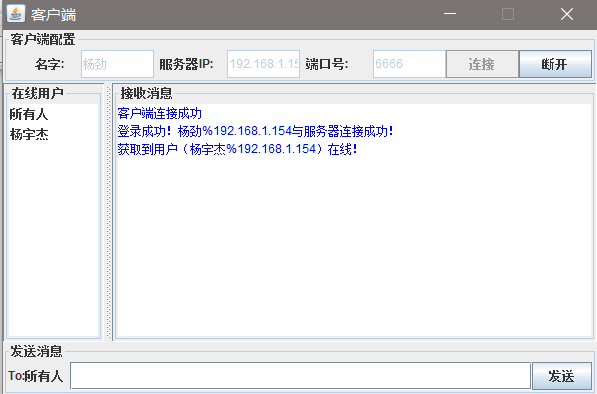
服务器和客户端在未开启／连接的状态下无法发送消息只能修改配置，而在开启／连接状态下无法修改配置只能发送消息，UI逻辑均由开启／连接状态决定。

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。

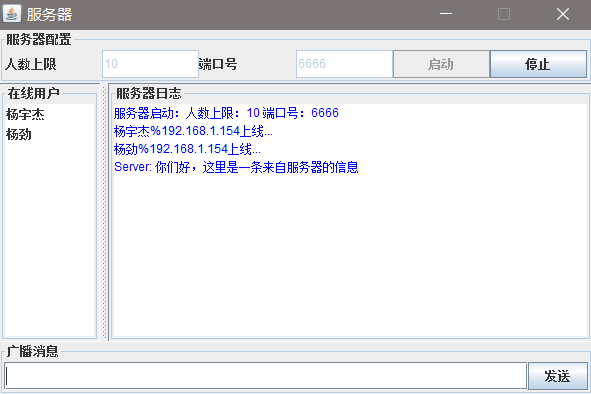
客户端可以设置自己的名字、服务器IP和服务器端口（默认均是192.168.1.154本地IP和端口6666）。客户端连接后可以在在线用户列表中选择所有人或其他在线用户发送消息。其他用户上下线信息在在线用户列表和聊天消息框中都有提示。

### 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。

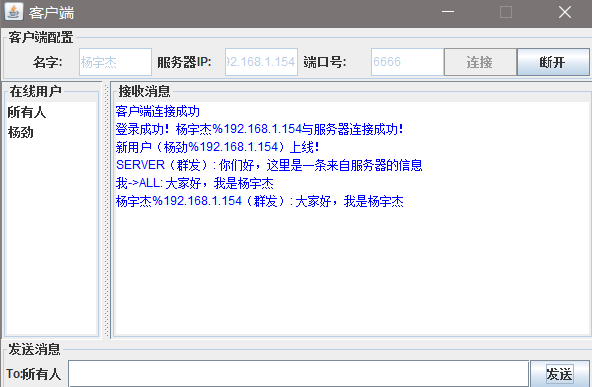


### 3.服务器群发消息

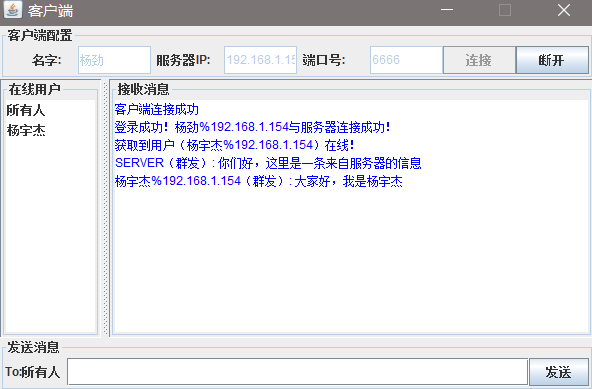


### 4. 客户端群发消息

客户端 杨宇杰 目标选择 所有人，并发送消息：

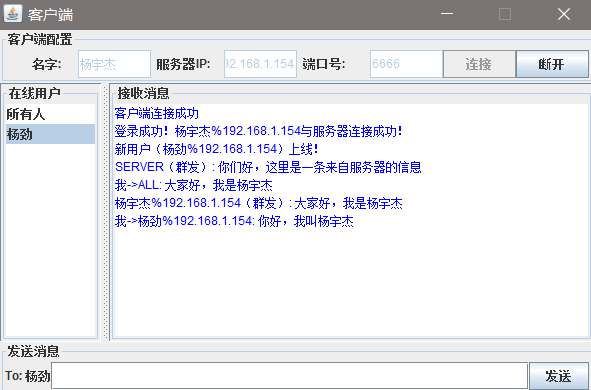
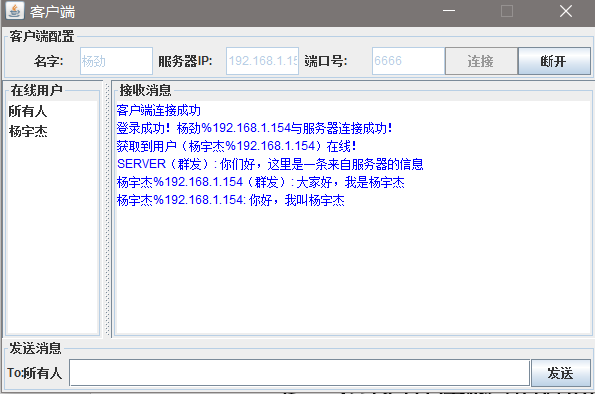


客户端 杨劲 收到了消息，并标记为群发：



### 5. 客户端发送私聊消息：

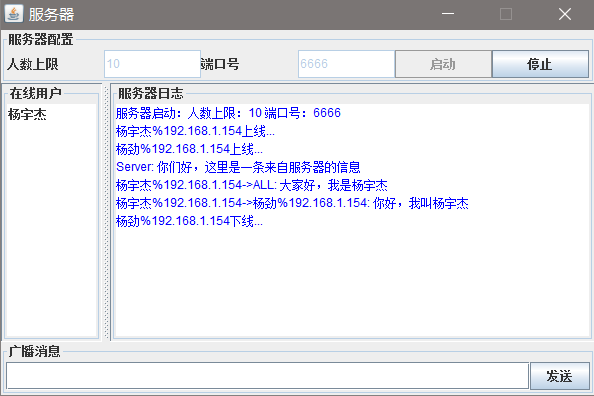
客户端 杨宇杰 和客户端 杨劲 私聊：

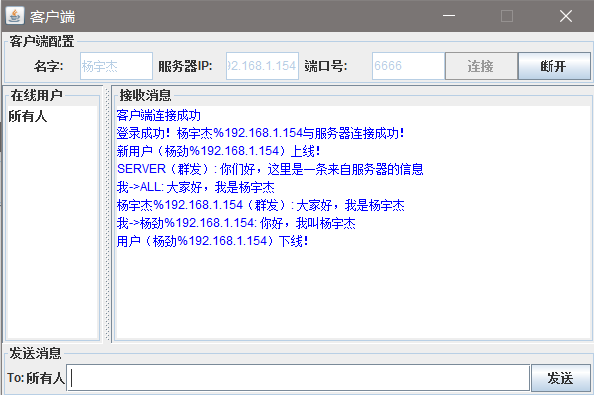
### 6. 客户端下线

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。