# 实验三——高级聊天程序

SA20225085 朱志儒

## 实验目的

熟悉Socket编程

## 实验环境

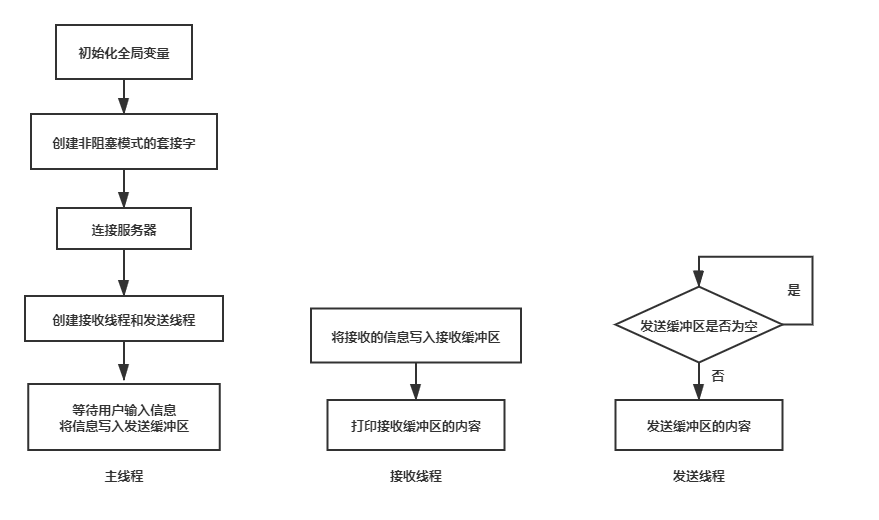
操作系统：Windows 10

编程环境：C++

## 实验内容

**客户端**

**流程图：**



**主要变量：**

1. SOCKET socketClient;
2. char sendBuffer[MAX\_BUFF];
3. bool isConnect, isSend = false;
4. CRITICAL\_SECTION criticalSection;

sockClient表示客户端新建的socket连接。

sendBuffer表示发送缓冲区，每当用户输入时，先将数据缓存在sendBuffer，发送线程将读取sendBuffer，然后将数据发送给服务器。

isConnect表示客户端是否与服务器建立连接。

isSend表示sendBuffer中是否有数据需要发送。

criticalSenction是临界区标志位，每当用户有输入时，进入临界区，将isSend置为true，将输入缓冲区的内容复制到sendBuffer中；每当isSend为true，表示有数据需要发送时，发送线程进入临界区，将sendBuffer中的数据发送给服务器。

**主要函数：**

接收线程：接收服务器发送来的数据，并打印已接收的数据。

1. *// 接收数据线程*
2. DWORD \_\_stdcall receiveDataThread(void\* param) {
3. int res;
4. char receiveBuffer[MAX\_BUFF];
5. memset(receiveBuffer, 0, MAX\_BUFF);
6. while (isConnect) {
7. res = recv(socketClient, receiveBuffer, MAX\_BUFF, 0);
8. if (res > 0) {
9. cout << receiveBuffer << endl;
10. }
11. else if (0 == res) {
12. isConnect = false;
13. isSend = false;
14. memset(receiveBuffer, 0, MAX\_BUFF);
15. cerr << "服务器关闭了连接！" << endl;
16. return 0;
17. }
18. else if (SOCKET\_ERROR == res) {
19. if (WSAEWOULDBLOCK == WSAGetLastError()) {
20. continue;
21. }
22. else {
23. isConnect = false;
24. cerr << "接收缓冲区不可用！" << endl;
25. return 0;
26. }
27. }
28. }
29. return 0;
30. }

发送线程：依据isSend判断是否有数据需要发送，如果需要，则进入临界区，读取并发送sendBuffer中的数据到服务器。

1. *// 发送数据线程*
2. DWORD \_\_stdcall SendDataThread(void\* param) {
3. while (isConnect) {
4. if (isSend) {
5. EnterCriticalSection(&criticalSection);
6. while (true) {
7. int res = send(socketClient, sendBuffer, MAX\_BUFF, 0);
8. if (SOCKET\_ERROR == res) {
9. if (WSAEWOULDBLOCK == WSAGetLastError()) {
10. continue;
11. }
12. else {
13. LeaveCriticalSection(&criticalSection);
14. cerr << "发送缓冲区不可用！" << endl;
15. return 0;
16. }
17. }
18. isSend = false;
19. break;
20. }
21. LeaveCriticalSection(&criticalSection);
22. }
23. }
24. return 0;
25. }

创建非阻塞模式套接字：

1. *// 创建套接字*
2. int res;
3. WSADATA wsData;
4. res = WSAStartup(MAKEWORD(2, 2), &wsData); *// 初始化 Windows Sockets Dll*
5. socketClient = socket(AF\_INET, SOCK\_STREAM, 0);
6. if (INVALID\_SOCKET == socketClient) {
7. cerr << "套接字创建失败！" << endl;
8. return -1;
9. }
10. unsigned long ul = 1;
11. res = ioctlsocket(socketClient, FIONBIO, (unsigned long\*)&ul);
12. if (SOCKET\_ERROR == res) {
13. cerr << "设置套接字非阻塞模式失败！" << endl;
14. return -1;
15. }

连接服务器：

1. *// 连接服务器*
2. sockaddr\_in serverAddress;
3. serverAddress.sin\_family = AF\_INET;
4. serverAddress.sin\_port = htons(ServerPort);
5. serverAddress.sin\_addr.S\_un.S\_addr = inet\_addr(ServerIP);
6. while (true) {
7. res = connect(socketClient, (sockaddr\*)&serverAddress, sizeof(serverAddress));
8. if (0 == res) {
9. break;
10. }
11. if (SOCKET\_ERROR == res) {
12. int errorCode = WSAGetLastError();
13. if (WSAEWOULDBLOCK == errorCode || WSAEINVAL == errorCode) {
14. continue;
15. }
16. else if (WSAEISCONN == errorCode) {
17. break;
18. }
19. else {
20. cerr << "连接服务器失败！" << endl;
21. return -1;
22. }
23. }
24. }
25. cerr << "成功连接服务器！" << endl;
26. isConnect = true;

创建接收线程和发送线程：

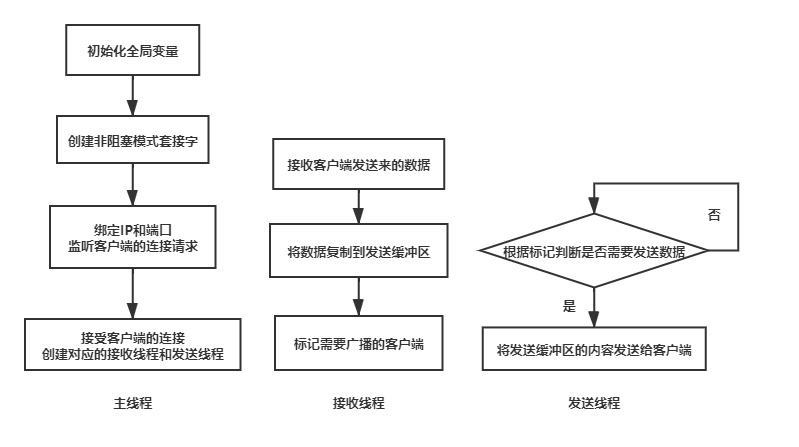
1. *// 创建接收和发送线程*
2. unsigned long theadID;
3. HANDLE threadReceive = CreateThread(nullptr, 0, receiveDataThread, nullptr, 0, &theadID);
4. if (nullptr == threadReceive) {
5. cerr << "创建接收线程失败！" << endl;
6. return -1;
7. }
8. HANDLE threadSend = CreateThread(nullptr, 0, SendDataThread, nullptr, 0, &theadID);
9. if (nullptr == threadSend) {
10. cerr << "创建发送线程失败！" << endl;
11. return -1;
12. }

等待用户输入：

1. *// 等待用户的输入*
2. char inputBuffer[MAX\_BUFF];
3. while (isConnect) {
4. memset(inputBuffer, 0, MAX\_BUFF);
5. cin.getline(sendBuffer, MAX\_BUFF);
6. EnterCriticalSection(&criticalSection);
7. memcpy(sendBuffer, inputBuffer, strlen(inputBuffer));
8. LeaveCriticalSection(&criticalSection);
9. isSend = true;
10. cin.sync();
11. }

**服务器**

流程图：



**主要变量：**

1. char writeBuffer[MAX\_BUFF];
2. SOCKET socketServer;
3. CRITICAL\_SECTION criticalSection;
4. map<SOCKET, bool> clientMap;

socketServer表示服务器用于监听和接收客户端连接请求的套接字。

clientMap表示广播标记，每当有新的客户端连接服务器时，clientMap新增一项 {连接该客户端的socket：false}；每当服务器接收到客户端A发送来的消息时，遍历clientMap将除A之外的所有客户端对应的value置为true，以标记这些客户端需要被广播消息。

writeBuffer表示发送缓冲区，每当有客户端向服务器发送消息时，服务器接收这些消息并将其缓存到writeBuffer。

criticalSection表示临界区标志位，使用criticalSection以互斥访问writerBuffer和clientMap。

**主要函数：**

接收线程：接收客户端A发送来的消息，打印该消息，进入临界区，将消息缓存到writeBuffer，遍历clientMap将除A之外的所有客户端对应的value置为true。

1. *// 接收线程*
2. DWORD \_\_stdcall receiveDataThread(void\* param) {
3. Param\* client = (Param\*)param;
4. int res;
5. char buffer[MAX\_BUFF];
6. while (true) {
7. memset(buffer, 0, MAX\_BUFF);
8. res = recv(client->socket, buffer, MAX\_BUFF, 0);
9. if (0 == res) {
10. cerr << inet\_ntoa(client->address.sin\_addr) << ':' << ntohs(client->address.sin\_port) << "\t断开连接！\t" << "error code: " << res << endl;
11. break;
12. }
13. if (SOCKET\_ERROR == res) {
14. if (WSAEWOULDBLOCK == WSAGetLastError()) {
15. continue;
16. }
17. else {
18. cerr << inet\_ntoa(client->address.sin\_addr) << ':' << ntohs(client->address.sin\_port) << "\t断开连接！\t" << "error code: " << WSAGetLastError() << endl;
19. break;
20. }
21. }
22. if (res > 0) {
23. cout << inet\_ntoa(client->address.sin\_addr) << ':' << ntohs(client->address.sin\_port) << "\t" << buffer << endl;
24. EnterCriticalSection(&criticalSection);
25. memset(writeBuffer, 0, MAX\_BUFF);
26. string str = inet\_ntoa(client->address.sin\_addr);
27. str += ":";
28. str += to\_string((unsigned int)ntohs(client->address.sin\_port));
29. str += "说： ";
30. str += buffer;
31. memcpy(writeBuffer, str.c\_str(), str.length());
32. for (auto iter = clientMap.begin(); iter != clientMap.end(); ++iter) {
33. if (iter->first != client->socket) {
34. iter->second = true;
35. }
36. }
37. LeaveCriticalSection(&criticalSection);
38. memset(buffer, 0, MAX\_BUFF);
39. }
40. }
41. return 0;
42. }

发送线程：进入临界区，依据clientMap判断是否需要发送数据，若需要，则将writeBuffer中的数据发送到客户端。

1. *// 发送线程*
2. DWORD \_\_stdcall sendDataThread(void\* param) {
3. Param\* client = (Param\*)param;
4. while (true) {
5. EnterCriticalSection(&criticalSection);
6. if (clientMap[client->socket]) {
7. int res = send(client->socket, writeBuffer, strlen(writeBuffer), 0);
8. if (SOCKET\_ERROR == res) {
9. if (WSAEWOULDBLOCK == WSAGetLastError()) {
10. continue;
11. }
12. else {
13. cerr << inet\_ntoa(client->address.sin\_addr) << ':' << ntohs(client->address.sin\_port) << "\t断开连接！" << endl
14. << "error code: " << WSAGetLastError() << endl;
15. LeaveCriticalSection(&criticalSection);
16. break;
17. }
18. }
19. clientMap[client->socket] = false;
20. }
21. LeaveCriticalSection(&criticalSection);
22. }
23. return 0;
24. }

创建非阻塞模式套接字：

1. *// 创建非阻塞模式套接字*
2. int res;
3. WSADATA wsData;
4. res = WSAStartup(MAKEWORD(2, 2), &wsData);
5. socketServer = socket(AF\_INET, SOCK\_STREAM, 0);
6. if (INVALID\_SOCKET == socketServer) {
7. cerr << "套接字创建失败！" << endl;
8. return -1;
9. }
10. unsigned long ul = 1;
11. res = ioctlsocket(socketServer, FIONBIO, (unsigned long\*)&ul);
12. if (SOCKET\_ERROR == res) {
13. cerr << "设置套接字非阻塞模式失败！" << endl;
14. return -1;
15. }

绑定IP和端口，监听客户端的连接请求：

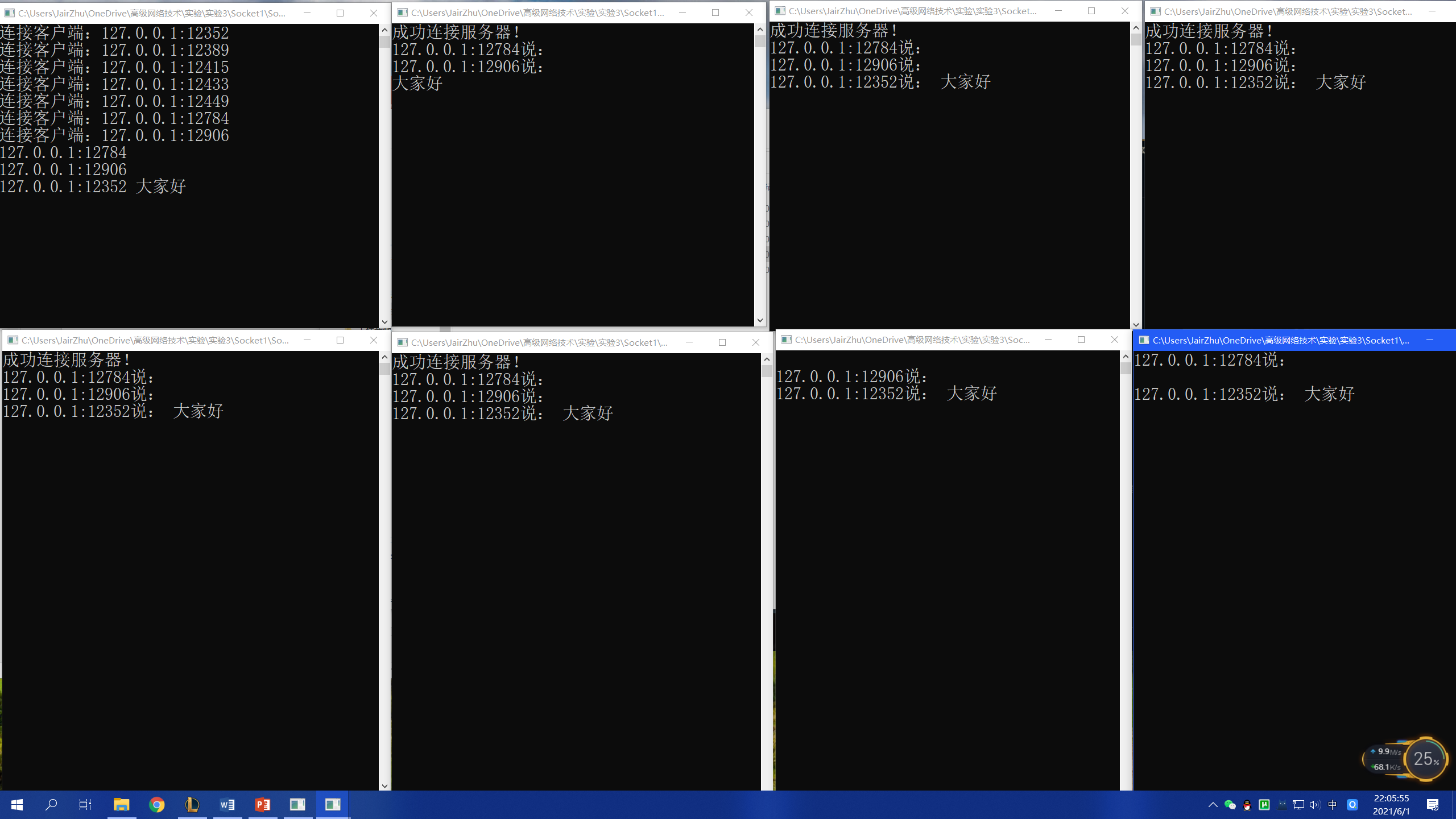
1. *// 绑定IP和端口，监听客户端的连接请求*
2. sockaddr\_in serverAddress;
3. serverAddress.sin\_family = AF\_INET;
4. serverAddress.sin\_port = htons(ServerPort);
5. serverAddress.sin\_addr.S\_un.S\_addr = INADDR\_ANY;
6. res = bind(socketServer, (sockaddr\*)&serverAddress, sizeof(serverAddress));
7. if (SOCKET\_ERROR == res) {
8. cerr << "套接字绑定失败！" << endl;
9. return -1;
10. }
11. res = listen(socketServer, MAX\_CONNECT);
12. if (SOCKET\_ERROR == res) {
13. cerr << "监听套接字失败！" << endl;
14. return -1;
15. }

接受客户端连接：每接受一个客户端的连接请求时，建立两个与之对应的线程分别处理接收数据任务和发送数据任务。

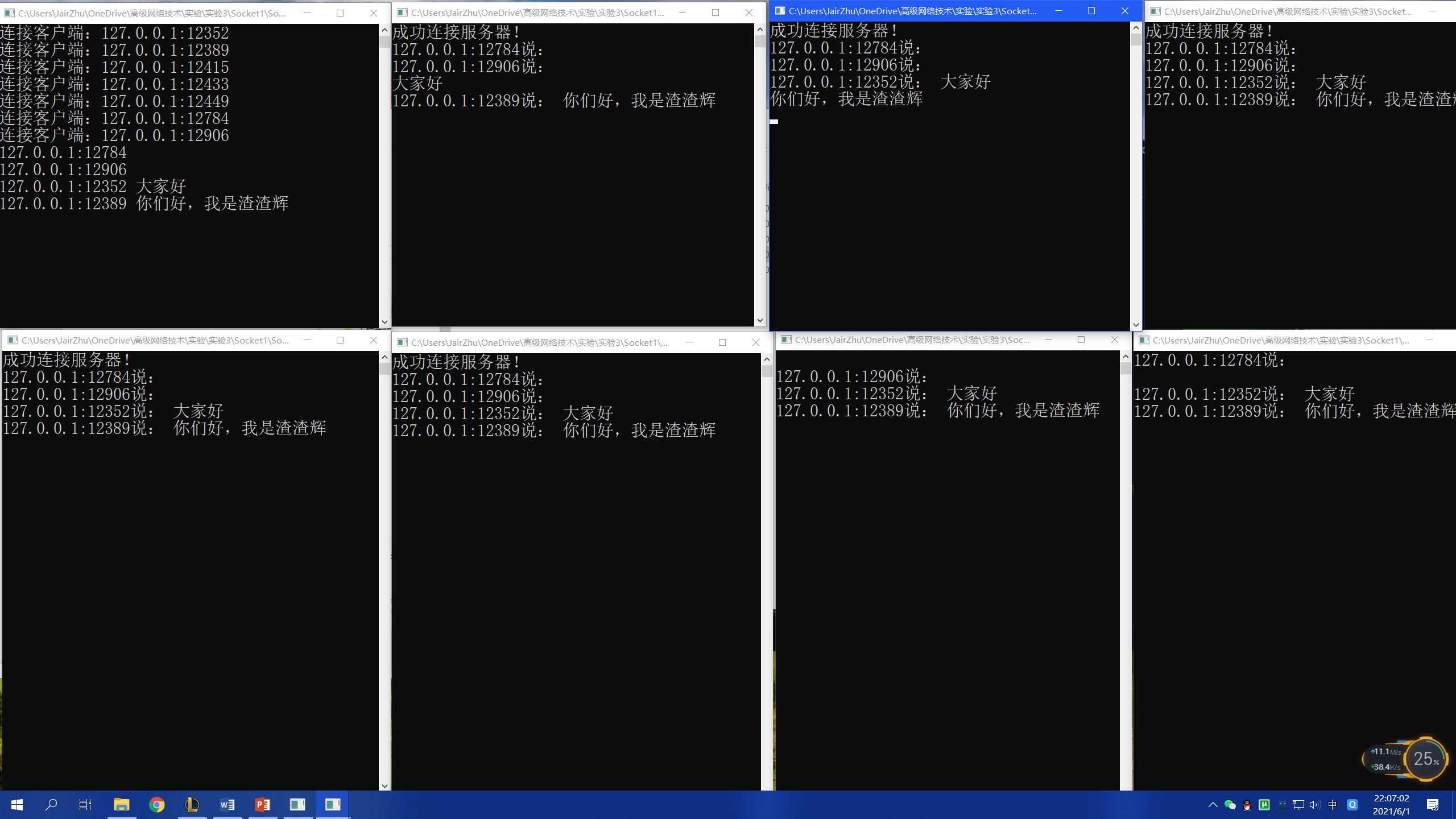
1. *// 接受客户端的连接*
2. SOCKET socketAccept;
3. sockaddr\_in clientAddress;
4. while (true) {
5. memset(&clientAddress, 0, sizeof(sockaddr\_in));
6. int sockaddr\_in\_len = sizeof(sockaddr\_in);
7. socketAccept = accept(socketServer, (sockaddr\*)&clientAddress, &sockaddr\_in\_len);
8. if (INVALID\_SOCKET == socketAccept) {
9. if (WSAEWOULDBLOCK == WSAGetLastError()) {
10. Sleep(500);
11. continue;
12. }
13. else {
14. cerr << "接受客户端的连接失败！" << endl;
15. cerr << "error code: " << WSAGetLastError() << endl;
16. break;
17. }
18. }
19. else {
20. cout << "连接客户端：" << inet\_ntoa(clientAddress.sin\_addr) << ":" << ntohs(clientAddress.sin\_port) << endl;
21. unsigned long ul;
22. clientMap[socketAccept] = false;
23. HANDLE receiveThread = CreateThread(nullptr, 0, receiveDataThread, new Param(socketAccept, clientAddress), 0, &ul);
24. if (nullptr == receiveThread) {
25. cerr << "接收数据线程创建失败！" << endl;
26. break;
27. }
28. HANDLE sendThread = CreateThread(nullptr, 0, sendDataThread, new Param(socketAccept, clientAddress), 0, &ul);
29. if (nullptr == sendThread) {
30. cerr << "发送数据线程创建失败！" << endl;
31. break;
32. }
33. }
34. }

## 实验结果

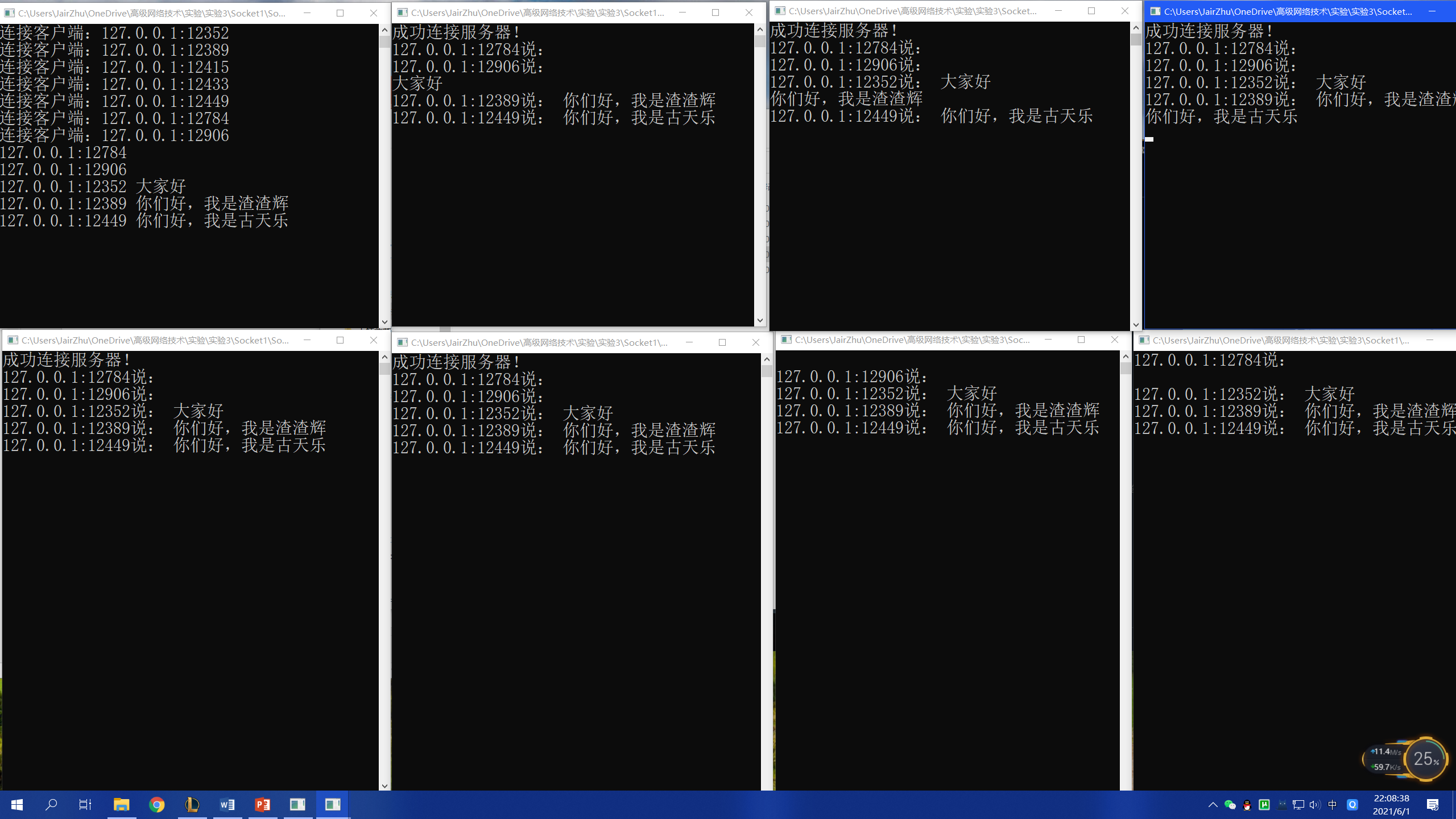
端口分别为12352，12389，12415，12433，12449，12784，12906的客户端分别连接服务器，进入聊天室，端口为12352的客户端向聊天室发送消息：



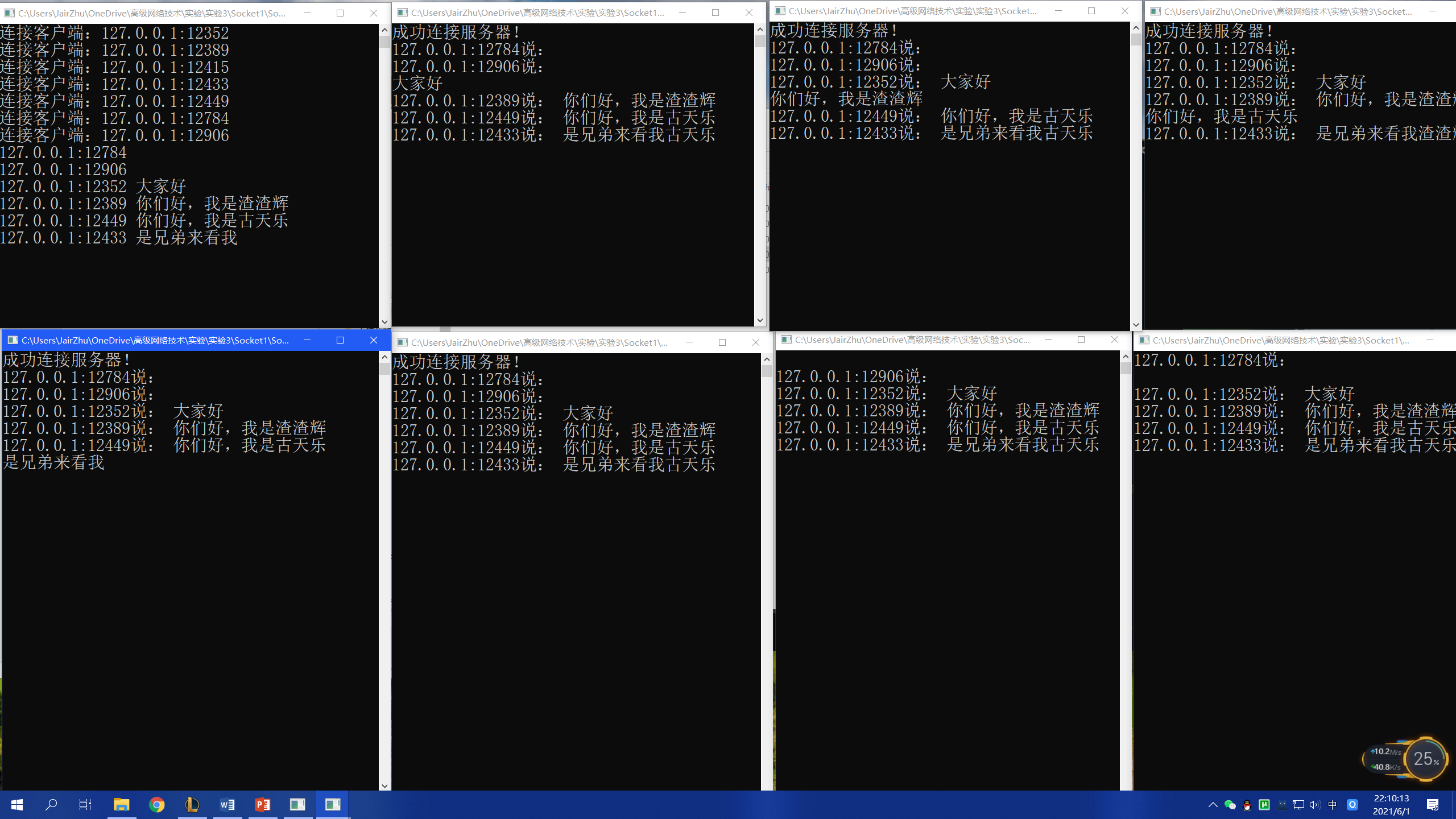
端口为12389的客户端向聊天室发送消息：



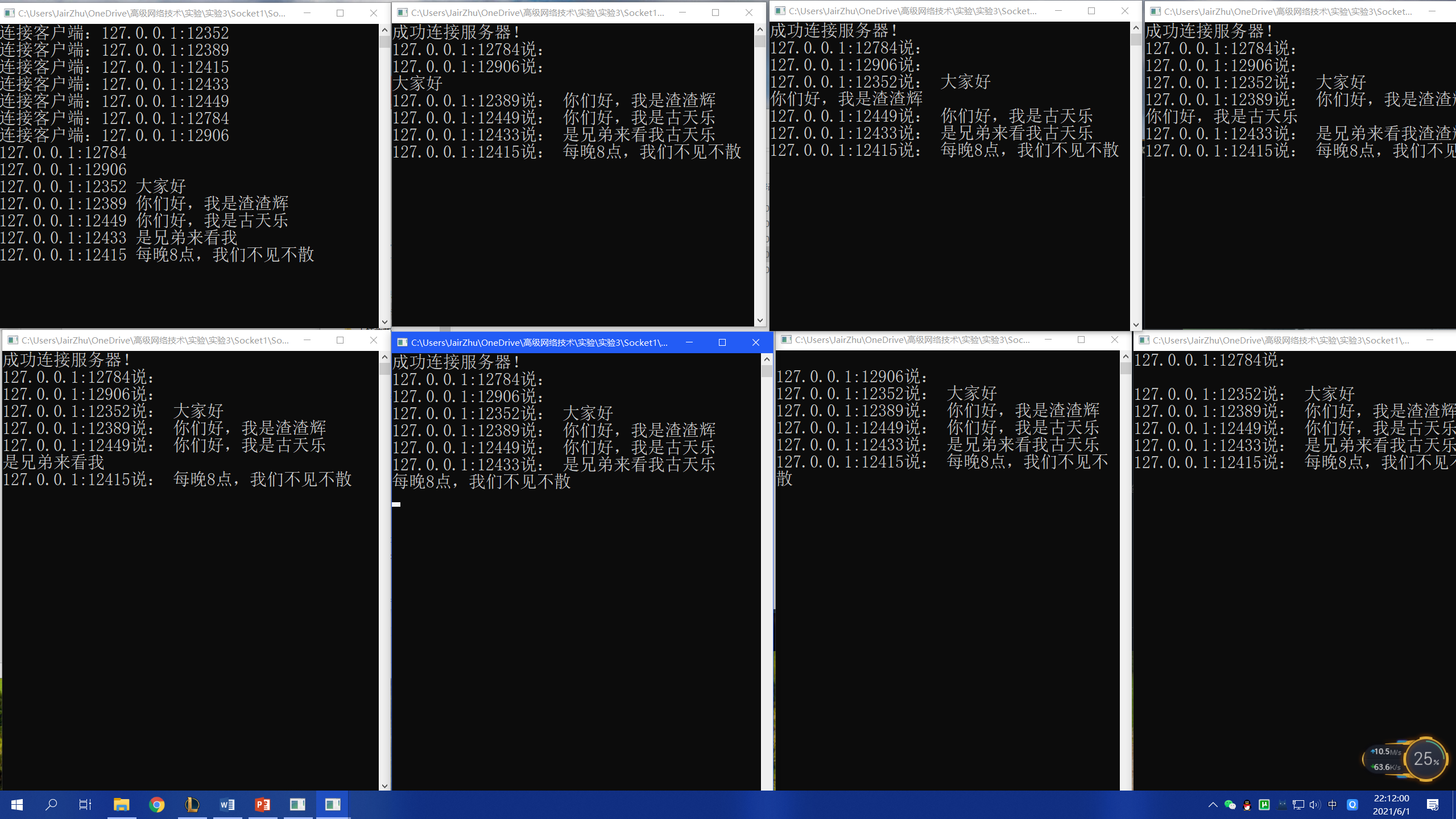
端口为12449的客户端向聊天室发送消息：



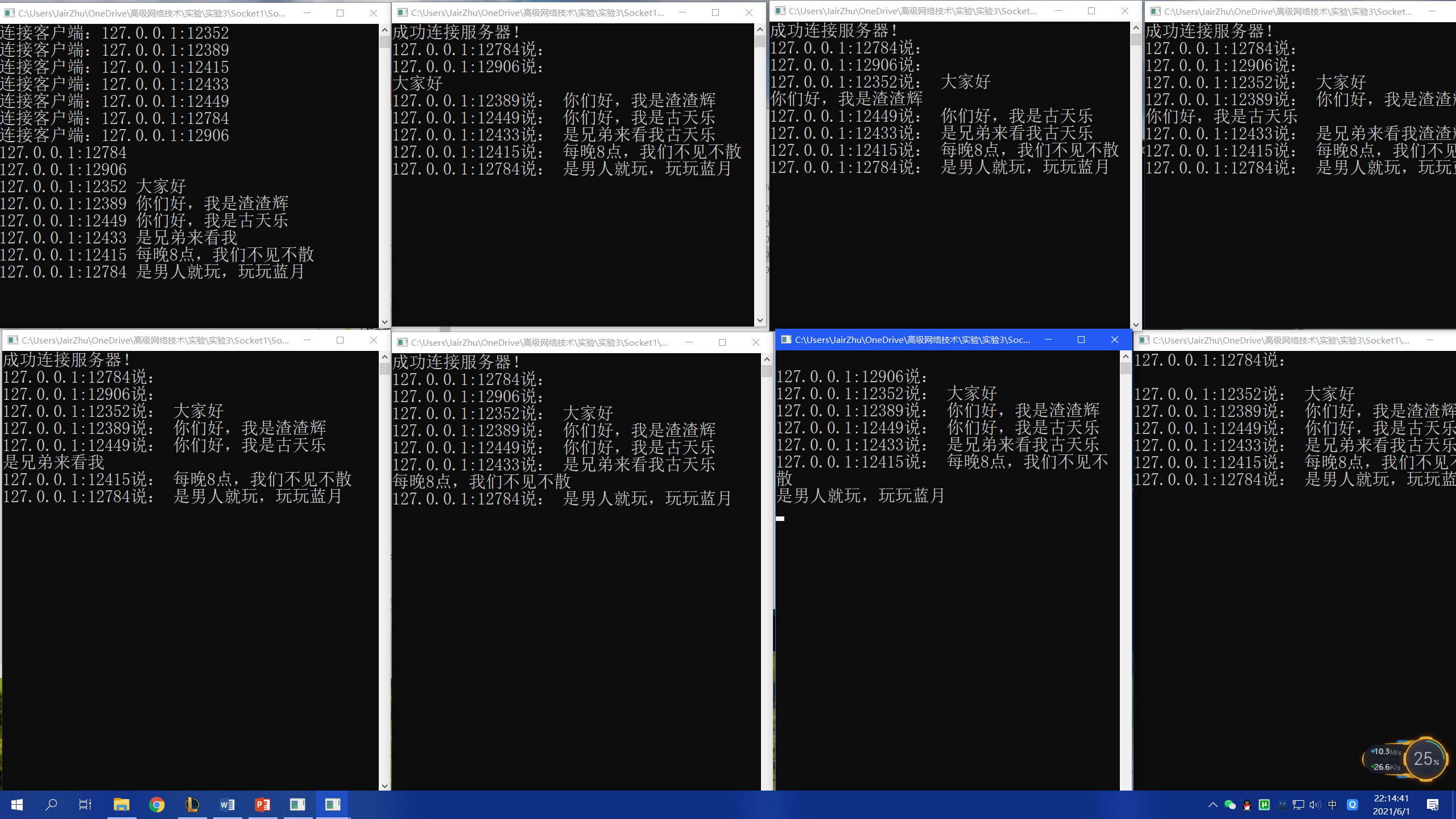
端口为12433的客户端向聊天室发送消息：



端口为12415的客户端向聊天室发送消息：



端口为12784的客户端向聊天室发送消息：



端口为12906的客户端向聊天室发送消息：

