南开大学

Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

实验一

张政泽

年级：2022 级

专业：信息安全

学号：2213573

2024 年 10 月 19 日

目录

一、 协议设计 **1**

二、 核心代码展示及讲解 **1**

（一） 服务端 1

（二） 客户端 9

三、 程序运行说明及结果展示 **15**

四、 实验反思及分析 **17**

一、 协议设计

按照实验要求，协议设计如下：

1. 概述：此协议支持多人聊天功能，支持用户之间的实时中英文文本通信。适用于网络中的多个客户端与一个服务器端之间的通信。
2. 通信模型：通信模型采用 C/S 模型，使用 TCP 传输协议，并选用流式套接字，支持 IPv4 地址，每个客户端通过 TCP 连接到服务器，服务器负责维护连接并将消息转发给所有连接的客户端。采用多线程的方式，发送信息和接收信息分线程进行。
3. 消息类型：消息分为用户消息和系统消息。用户消息即用户的聊天内容；系统消息即用户加入或离开的通知。
4. 消息格式：由于采用 TCP 传输协议，因此消息包括头部 (包括消息长度、消息类型、时间戳等)、数据负载、尾部 (数据完整性校验信息) 三个部分，其中数据负载部分的格式为”name:+Message”，name 为客户端程序启动时要求输入的用户名，Message 为发送的消息内容。
5. 消息传输：服务器和客户端通过 recv() 和 send() 函数来接收消息和发送消息。
6. 连接管理：
   (a) 客户端发送连接请求到服务器，服务器响应确认连接。
   (b) 设置最大连接数量为 10，当超出最大连接数量时，服务端将不会再 accept，直至连接数量小于 10.
   (c) 设置单次发送的消息最多为 50 字节，超出部分无法发送。
   (d) 使用 socket() 建立 socket, 使用 connect() 系统调用建立 TCP 连接, 使用 closesocket() 关闭连接以及使用 WSACleanup() 清理资源。
   (e) 客户端或服务器可以发起连接关闭请求。
7. 日志记录：服务器和客户端均存在日志记录功能，日志格式为：Time+Message，即时间 + 消息
8. 错误处理：定义了错误处理代码，若客户端连接超时，将等待 1s 后重新连接；若客户端异常退出，则服务器清理其资源；若服务器异常退出，则客户端记录错误信息并清理资源后退出。

二、 核心代码展示及讲解

(一) 服务端

首先给出服务端完整代码：

Socket-Server

```
#include<iostream>
#include<winsock2.h>
#include<ws2tcpip.h>
#include<windows.h>
```

```
#include<list>
#include<vector>
#include <algorithm>
#include <ctime>
#include<fstream>
#include <locale>
using namespace std;
ofstream outfile("./log.txt",std::ios::app);
std::time_t now;
std::tm* local_now;
bool TF;
int MaxClientNum = 10;
vector<SOCKET> ClientSocketList; // 存储客户端套接字
SOCKET sockSer; // 服务器套接字
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE); // 获取控制台句柄
void LogSet(string str){
    now = std::time(0);  // 获取当前时间
    local_now = std::localtime(&now);  // 转换为本地时间
    outfile<<asctime(local_now);
    outfile<<str<<endl;
}
DWORD WINAPI ListenExit(LPVOID lpParam) {
    string message;
    while(true){
        getline(cin,message);
        if(message == "./exit"){

            //发送响应
         for(int i=0;i<ClientSocketList.size();i++)
                if(send(ClientSocketList[i], "Server will be closed in 5 
                    seconds", strlen("Server will be closed in 5 seconds"), 
                    0)== SOCKET_ERROR){
                    cout<<i<<"send error"<<endl;
                }
            cout<<"Server will be closed in 5 seconds"<<endl;
            LogSet("Server will be closed in 5 seconds");
            for(int i=0;i<ClientSocketList.size();i++)
                if(send(ClientSocketList[i], "5", strlen("5"), 0)== 
                    SOCKET_ERROR){
                    cout<<i<<"send error"<<endl;
                }
            cout<<"5"<<endl;

            Sleep(1000);
            for(int i=0;i<ClientSocketList.size();i++)
                if(send(ClientSocketList[i], "4", strlen("4"), 0)== 
                    SOCKET_ERROR){
                    cout<<i<<"send error"<<endl;
```

```
49                  }
50              cout<<"4"<<endl;
51
52              Sleep(1000);
53              for(int i=0;i<ClientSocketList.size();i++)
54                  if(send(ClientSocketList[i], "3", strlen("3"), 0)==
                        SOCKET_ERROR){
55                      cout<<i<<"send error"<<endl;
56                  }
57              cout<<"3"<<endl;
58
59              Sleep(1000);
60              for(int i=0;i<ClientSocketList.size();i++)
61                  if(send(ClientSocketList[i], "2", strlen("2"), 0)==
                        SOCKET_ERROR){
62                      cout<<i<<"send error"<<endl;
63                  }
64              cout<<"2"<<endl;
65
66              Sleep(1000);
67              for(int i=0;i<ClientSocketList.size();i++)
68                  if(send(ClientSocketList[i], "1", strlen("1"), 0)==
                        SOCKET_ERROR){
69                      cout<<i<<"send error"<<endl;
70                  }
71              cout<<"1"<<endl;
72
73              for(int i=0;i<ClientSocketList.size();i++)
74                  if(send(ClientSocketList[i], "Exiting...", strlen("Exiting...
                        "), 0)== SOCKET_ERROR){
75                      cout<<i<<"send error"<<endl;
76                  }
77              cout<<"Exiting..."<<endl;
78              LogSet("Exiting...");
79              if(ClientSocketList.size() == 0){
80                  cout<<"No client connected"<<endl;
81                  LogSet("No client connected");
82                  // 关闭套接字
83                  closesocket(sockSer);
84                  break;
85              }
86              TF = false;
87              Sleep(1000);///为了等待客户端退出
88              break;
89          }
90      }
91  }
92  DWORD WINAPI ThreadFunction(LPVOID lpParam) {
```

```
93
94      SOCKET ClientSocket =(SOCKET)lpParam;
95      ClientSocketList.push_back(ClientSocket);
96
97      char  recvBuf[50];
98      memset(recvBuf,0,sizeof(recvBuf));
99      int recvSize;
100
101 while(true){
102     if(TF == false) break;
103     // 接收数据并输出
104     recvSize=recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
105     if(recvSize == SOCKET_ERROR){ //说明退出了
106
107         cout<<"Client disconnected"<<endl;
108         LogSet("Client disconnected");
109         // 从列表中移除客户端套接字
110         auto it = std::find(ClientSocketList.begin(), ClientSocketList.end(),
                ClientSocket);
111         // 如果找到了元素，计算它的位置并删除
112         if (it != ClientSocketList.end())  ClientSocketList.erase(it);
113         // 关闭套接字
114         closesocket(ClientSocket);
115         break;
116     }
117
118     else{
119         recvBuf[recvSize] = '\0'; // 确保字符串以NULL结尾
120
121         SOCKET valueToFind = ClientSocket;
122         auto it = std::find(ClientSocketList.begin(), ClientSocketList.end(),
                valueToFind);
123         int position;
124         // 如果找到了元素，计算它的位置
125         if (it != ClientSocketList.end())  position = std::distance(
                ClientSocketList.begin(), it);
126
127         if(strstr(recvBuf, ": have exited!") != NULL){
128             cout<<recvBuf<<endl;
129             LogSet(recvBuf);
130
131             Sleep(1000);
132             // 关闭套接字
133             for(int i=0;i<ClientSocketList.size();i++)
134                 closesocket(ClientSocketList[i]);
135             closesocket(sockSer);
136             LogSet("SOCKET closed Successfully!");
137             break;
```

```
138
139         }
140         if(strcmp(recvBuf, "SuccessExit!") == 0){
141             ClientSocketList.erase(it);
142             LogSet("SOCKET erased Successfully!");
143             // 关闭套接字
144             closesocket(ClientSocket);
145             break;
146         }
147
148         // 设置字体颜色
149         switch(position){
150             case 0: SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);break
                    ;
151             case 1: SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);break;
152             case 2: SetConsoleTextAttribute(hConsole, FOREGROUND_RED);break;
153             case 3: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_GREEN);break;
154             case 4: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_BLUE);break;
155             case 5: SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN |
                        FOREGROUND_BLUE);break;
156             case 6: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_GREEN | FOREGROUND_BLUE);break;
157             case 7: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY);
                        break;
158             case 8: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY);
                        break;
159             case 9: SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
                        FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY);
                        break;
160             }
161         cout<<recvBuf<<endl;
162         LogSet(recvBuf);
163         // 重置字体颜色为默认值
164         SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
                FOREGROUND_BLUE);
165
166         //发送响应
167         for(int i=0;i<ClientSocketList.size();i++)
168             if(ClientSocketList[i]!=ClientSocket)
169                 if(send(ClientSocketList[i], recvBuf, recvSize, 0)==
                        SOCKET_ERROR){
170                     cout<<i<<"send error"<<endl;
171                 }
172
```

```
173         if(strstr(recvBuf, "Exit") != NULL){
174             if(send(ClientSocket, "OK! I got it", strlen("OK! I got it"), 0)
                    == SOCKET_ERROR){
175                 cout<<WSAGetLastError()<<endl;
176                 LogSet("send error. Errno:"+to_string(WSAGetLastError()))
                        ;
177                 cout<<"send error"<<endl;
178                 }
179         }
180     }
181
182     }
183
184     return 0;
185 }
186
187 int main()
188 {
189
190     LogSet("Server Start");
191
192     WSADATA wsaData;
193     WORD wVersionRequested = MAKEWORD(2, 2);
194     WSAStartup(wVersionRequested, &wsaData);
195     // 创建套接字
196     sockSer = socket(AF_INET, SOCK_STREAM, 0);
197     if(sockSer == -1)
198     {
199         cout<<"Socket creation failed"<<endl;
200         LogSet("Socket creation failed");
201     }
202
203     else
204     {
205         cout<<"Socket created successfully"<<endl;
206         LogSet("Socket created successfully");
207     }
208
209
210     // 定义地址结构—服务端
211     sockaddr_in addressSrv;
212     addressSrv.sin_family = AF_INET; // 使用IPv4
213     addressSrv.sin_addr.s_addr = INADDR_ANY; // 绑定到所有可用接口
214     addressSrv.sin_port = htons(8087); // 端口号, htons确保端口字节序正确
215     // 绑定套接字到地址和端口
216     if(bind(sockSer,(sockaddr *)&addressSrv, sizeof(addressSrv)) == -1)
217         {
218             cout<<"Bind failed"<<endl;
```

```
219             LogSet("Bind failed");
220         }
221     else
222         {
223             cout<<"Bind success"<<endl;
224             LogSet("Bind success");
225         }
226
227     // 监听连接
228     if(listen(sockSer, 5) == -1)
229     {
230         cout<<"Listen failed"<<endl;
231         LogSet("Listen failed");
232     }
233
234     else
235         {
236             cout<<"Listen success"<<endl;
237             LogSet("Listen success");
238         }
239
240
241     //创建线程监听服务器退出指令
242     HANDLE ListenhThread = CreateThread(
243                 NULL, // 默认安全属性
244                 0,    // 默认堆栈大小
245                 ListenExit, // 线程函数
246                 NULL, // 传递给线程函数的参数
247                 0,    // 创建立即运行的线程
248                 NULL  );
249
250     sockaddr_in  clientAddr;
251     int clientAddrSize = sizeof(clientAddr);
252     HANDLE hThread;
253     TF=true;
254     while(1){
255         if(TF==false) break;
256         if(MaxClientNum==ClientSocketList.size()) {
257             //超出了最大连接数, 等待一段时间再次尝试
258             Sleep(1000); // 等待一段时间再次尝试
259             continue; // 跳过当前循环的剩余部分
260         }
261         SOCKET sockConnection = accept(sockSer, (SOCKADDR*)&clientAddr, &
                clientAddrSize);
262         if(sockConnection == INVALID_SOCKET){
263             if(TF==false) break;
264             int error = WSAGetLastError();
265             if(error == WSAEWOULDBLOCK){
```

```
266                 // 没有新的连接，稍后重试
267                 Sleep(100); // 等待一段时间再次尝试
268                 continue; // 跳过当前循环的剩余部分
269             }else{
270                 // 输出其他错误信息
271                 printf("Accept failed with error: %d\n", error);
272                 LogSet("Accept failed with error: "+to_string(error));
273                 break; // 退出循环
274             }
275         }
276         else
277             {
278             hThread = CreateThread(
279                 NULL, // 默认安全属性
280                 0,    // 默认堆栈大小
281                 ThreadFunction, // 线程函数
282                 LPVOID(sockConnection), // 传递给线程函数的参数
283                 0,    // 创建立即运行的线程
284                 NULL
285             );
286             if (hThread == NULL) {
287                 printf("Create thread failed\n");
288                 LogSet("Create thread failed");
289             }
290             CloseHandle(hThread);
291             }
292 
293     }
294     Sleep(2000);///等待线程结束
295     WSACleanup();
296     cout<<"Exit Success"<<endl;
297     LogSet("Exit Success");
298     LogSet("Server End");
299     return 0;
300 }
```

接下来给出代码解析：

1. 三个函数：

   (a) 函数 LogSet：此函数的功能就是记录日志，函数通过参数传入一个字符串，此字符串附上时间信息后会被记录进日志。

   (b) 函数 ThreadFunction：此函数是一个线程函数，其功能是接受客户端发来的信息并进行响应的发送。具体来说，此函数首先通过参数传入与客户端建立连接时使用的 SOCKET，接着将此 SOCKET 放入 Vector 容器 ClientSocketList 当中，此容器的作用就是在服务器端保存各个客户端的 SOCKET，且被设置了最大数量 MaxClient-Num=10。

   接下来是一个 while(true) 的死循环，在这个死循环当中，会不断地通过 recv 接收 SOCKET 的信息。在收到正常的用户消息时，首先会根据此用户的身份信息来将控

制台字体设置为相应的颜色 (以便视觉上进行区分), 紧接着将用户发送的信息输出到控制台。最后，为了使其他用户获取此用户发送的消息，服务端紧接着会通过使用 ClientSocketList 这个容器中保存的其他的客户端的 SOCKET 来将此消息发送给其余的客户端。

(c) 函数 ListenExit：此函数的作用是监听服务端通过输入'./exit' 指令退出，详细的服务端退出机制会在之后解释。

2. main 函数：在 main 函数中我们先后经历了创建套接字–> 定义地址结构–> 绑定套接字到指定地址和端口–> 监听连接这几个基本的准备阶段；紧接着我们创建了监听服务端退出进程 (使用上述的 ListenExit 函数)；然后进入 while(true) 死循环，在循环内部不断地通过 accept 函数来接收连接请求，当成功建立连接后，会创建另一个进程来接收客户端的输入 (使用上述的 ThreadFunction 函数)；最后，当 while(true) 循环因为某些原因 break 后，进行清理工作。

3. ./exit 退出机制: 当服务端的监听退出线程收到输入的'./exit' 指令后，程序便会运行退出机制。首先要向各个客户端发送服务器端将要退出的信息：“Server will be closed in 5 seconds”、”5”、“4”、“3”、“2”、“1”、“Exiting”，用户端会将这些信息正常打印到控制台界面；接着，若此时无客户端连接，则直接关闭 SOCKET，并结束线程，不久后 main 函数中的 while 循环会 break，紧接着便会进行一些清理工作后正常退出程序。若此时存在客户端连接，监听退出的线程在发送完上述信息后便会将一个全局 Bool 变量 TF 置为 false(此全局变量用于控制 while 循环，当 TF==false 时，while 循环会 break)，接着会 Sleep(1000)，目的是等待客户端的 SOCKET 全部关闭连接。而在客户端部分，在收到以上信息 (”Server will be closed in 5 seconds”) 后，便会在接收线程中进行特定的处理：接收下来的'5”4”3”2”1' 并向服务端发送 name+”: have exited!” 这样的信息 (原因是，服务端使用的 recv 函数处于阻塞状态，若不发送该消息则无法正常地运行写在 recv 函数之后的代码)，接着客户端便会关闭套接字、清理资源、记录日志、退出。返回来解析服务端，当服务端收到客户端发来的 name+”: have exited!” 信息后，首先将该信息打印到控制台，接着关闭 SOCKET，然后 break 出接收信息的 while 循环，然后线程结束。而在 main 函数中，此时由于 accept 长时间无法接收到新的连接便会在 Sleep(1000) 后进入下一次循环，而每次循环开始前都会对 TF 的值进行检查，由于此时 TF 被置为了 false，因此循环 break，在等待线程结束、清理资源后，程序正常结束。

(二) 客户端

同样，首先给出客户端完整代码:

Socket-Client

```
#include<iostream>
#include<winsock2.h>
#include <ws2tcpip.h>
#include<string.h>
#include<ctime>
#include<fstream>
#include <limits> // 包含 std::numeric_limits 的定义
using namespace std;
ofstream outfile("./ClientLog.txt",std::ios::app);
std::time_t now;
```

```
std::tm* local_now;
string name;
bool IsOver = false;
SOCKET sockClient; // 全局变量，用于保存客户端套接字
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
HANDLE hEvent;  // 事件句柄
void LogSet(string name){
    now = std::time(0);  // 获取当前时间
    local_now = std::localtime(&now);  // 转换为本地时间
    outfile<<asctime(local_now);
    outfile<<name<<endl;
}
DWORD WINAPI ThreadFunction(LPVOID lpParam) {

    SOCKET ClientSocket =(SOCKET)lpParam;
    char  recvBuf[50];
    memset(recvBuf,0,sizeof(recvBuf));
    int recvSize;

while(true){

    if(IsOver == true)break;
    // 接收数据并输出
    recvSize=recv(ClientSocket,recvBuf,sizeof(recvBuf),0);

    if(recvSize == SOCKET_ERROR){
        cout<<WSAGetLastError()<<endl;
        LogSet("ERROR"+to_string(WSAGetLastError()));
        // 重置字体颜色为默认值
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
            FOREGROUND_BLUE);
        // 关闭套接字
        closesocket(ClientSocket);
        WSACleanup();
        LogSet("Socket closed Successfully");
        exit(1);
    }

    else{
        recvBuf[recvSize] = '\0'; // 确保字符串以NULL结尾
        // 重置字体颜色为默认值
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
            FOREGROUND_BLUE);
        cout<<endl;
        if(strcmp(recvBuf, "OK! I got it") == 0){
            IsOver = true;
            send(sockClient, "SuccessExit!", strlen("SuccessExit!"), 0);
            closesocket(ClientSocket);
```

```
57              continue;
58          }
59
60          if(strcmp(recvBuf, "Server will be closed in 5 seconds") == 0){
61
62              cout<<"Server will be closed in 5 seconds"<<endl;
63              LogSet("Server will be closed in 5 seconds");
64              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
65              cout<<recvBuf[0]<<endl;
66              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
67              cout<<recvBuf[0]<<endl;
68              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
69              cout<<recvBuf[0]<<endl;
70              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
71              cout<<recvBuf[0]<<endl;
72              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
73              cout<<recvBuf[0]<<endl;
74               for(int i=0;i<50;i++){
75                   recvBuf[i] = ' ';
76              }
77              recv(ClientSocket,recvBuf,sizeof(recvBuf),0);
78              cout<<recvBuf<<endl;
79              LogSet(recvBuf);
80
81              if(send(ClientSocket, (name+": have exited!").c_str(), strlen((
                    name+": have exited!").c_str()), 0)== SOCKET_ERROR){
82                  cout<<"Client have exited!--Send failed"<<endl;
83                  LogSet("Client have exited!--Send failed");
84              }
85              //关闭套接字
86              closesocket(ClientSocket);
87              WSACleanup();
88              cout<<"Socket closed Successfully"<<endl;
89              cout<<"Client Exit Successfully"<<endl;
90              LogSet("Client Exit Successfully");
91              exit(0);
92          }
93          cout<<recvBuf<<endl;
94          LogSet(recvBuf);
95          SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
96          cout<<name<<":";
97          LogSet(name+":");
98      }
99
100     }
101     // 完成工作后设置事件
102     //cout<<"SetEventSuccess"<<endl;
103     SetEvent(hEvent);
```

```
104     return 0;
105 }
106 int main()
107 {
108     LogSet("Client Start");
109     WSADATA wsaData;
110     WORD wVersionRequested = MAKEWORD(2, 2);
111     WSAStartup(wVersionRequested, &wsaData);
112 L1:
113     cout<<"Please input your name:"<<endl;
114     cin>>name;
115     if(strlen(name.c_str())>10){
116         cout<<"Name is too long"<<endl;
117         goto L1;
118     }
119      // 清空 cin 的缓冲区
120     std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // 丢
        弃缓冲区中的字符
121     std::cin.clear(); // 清除错误标志
122
123     // 创建套接字
124     sockClient = socket(AF_INET, SOCK_STREAM, 0);
125     if(sockClient == -1)
126         {
127             cout<<"Socket creation failed"<<endl;
128             cout<<WSAGetLastError()<<endl;
129             LogSet("Socket creation failed");
130             exit(0);
131         }
132     else
133         {
134             cout<<"Socket created successfully"<<endl;
135             LogSet("Socket created successfully");
136         }
137
138     // 定义地址结构—服务端
139     sockaddr_in addressSrv;
140     addressSrv.sin_family = AF_INET; // 使用IPv4
141     addressSrv.sin_addr.s_addr = inet_addr("127.0.0.1"); // 使用本地回环地址
142     addressSrv.sin_port = htons(8087); // 端口号, htons确保端口字节序正确
143
144     // 连接服务器
145     if(connect(sockClient, (sockaddr*)&addressSrv, sizeof(addressSrv)) ==
        SOCKET_ERROR)
146         {
147             cout<<"Connection failed"<<endl;
148             cout<<WSAGetLastError()<<endl;
149             LogSet("Connection failed.Error Code:"+to_string(WSAGetLastError
```

```
                ()));
150             closesocket(sockClient);
151             WSACleanup();
152             LogSet("Socket closed Successfully");
153         }
154     else
155         {
156             cout<<"Connected to server"<<endl;
157             LogSet("Connected to server");
158         }
159
160     send(sockClient, (name+" Enter.").c_str(), strlen(name.c_str())+7, 0);
161     cout<<endl;
162     cout<<endl;
163     cout<<endl;
164
165     //建立线程接收数据
166     HANDLE hThread = CreateThread(
167                 NULL, // 默认安全属性
168                 0,    // 默认堆栈大小
169                 ThreadFunction, // 线程函数
170                 LPVOID(sockClient), // 传递给线程函数的参数
171                 0,    // 创建立即运行的线程
172                 NULL  // 不需要线程ID
173             );
174             if (hThread == NULL) {
175                 printf("Create thread failed\n");
176                 LogSet("Create thread failed");
177             }
178             CloseHandle(hThread);
179
180     hEvent = CreateEvent(
181         NULL,                   // 默认安全性
182         TRUE,                   // 手动重置事件
183         FALSE,                  // 初始状态为无信号
184         NULL                    // 匿名事件
185     );
186
187     char sendBuf[50];
188     memset(sendBuf,0,sizeof(sendBuf));
189     string buffer;
190
191     cout<<"Press './exit' to exit"<<endl;
192 while(true){
193
194     for(int i=0;i<50;i++){
195         sendBuf[i]=0;
196     }
```

```
197     // 设置字体颜色为绿色
198     SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
199     cout<<name<<":";
200     getline(cin, buffer);
201     if(strcmp(buffer.c_str(), "./exit") == 0){
202         if(send(sockClient, (name+" Exit").c_str(), strlen(name.c_str())+6,
            0) == SOCKET_ERROR)
203                 {
204                     cout<<"Send failed"<<endl;
205                     LogSet("Message:"+(name+" Exit")+"Send failed");
206                 }
207         ///睡眠以等待发送成功
208         Sleep(1000);
209         break;
210     }
211     for(int i=0;i<strlen(name.c_str());i++)
212         {
213             sendBuf[i]=name[i];
214         }
215     sendBuf[strlen(name.c_str())]=':';
216     for(int t=strlen(name.c_str())+1;t<strlen(name.c_str())+1+strlen(buffer.
        c_str());t++)
217         {
218             sendBuf[t]=buffer[t-strlen(name.c_str())-1];
219         }
220
221     if(send(sockClient, sendBuf, strlen(sendBuf)+1, 0)== SOCKET_ERROR)
222         cout<<"Send failed"<<endl;
223     else
224         LogSet("Message:"+string(sendBuf)+"Send successfully");
225
226 }
227     // 等待子线程结束
228     WaitForSingleObject(hEvent, INFINITE);
229     cout<<"Thread Over!"<<endl;
230     LogSet("Thread Over!");
231     // 关闭套接字及句柄
232     CloseHandle(hEvent);
233     closesocket(sockClient);
234     WSACleanup();
235     cout<<"Client closed Success"<<endl;
236     LogSet("Client closed Success");
237     return 0;
238 }
```

接下来给出代码解析:

1. 两个函数

(a) LogSet 函数：此函数的功能就是记录日志，函数通过参数传入一个字符串，此字符串附上时间信息后会被记录进日志。在线程函数及主函数部分会出现大量地对此函数的调用，作用就是记录日志，因此接下来的分析部分将不再关注此函数。

(b) ThreadFunction 函数：此函数是一个线程函数，其功能是接收服务端发来的响应消息。同样地，通过函数的参数获取到使用的 SOCKET；接着进入 while(true) 死循环，利用此 SOCKET 不断地接收数据，若收到的消息正常，则将此消息打印到控制台并进入下一次循环；若因为某些原因此循环 break，就会通过 SetEvent() 函数来设置时间，这与 main 函数中的 WaitForSingleObject 函数相照应。

2. main 函数：在 main 函数中，首先会要求用户输入其用户名，目的是为了进一步区分不同的用户。接着在经过了创建套接字–> 定义地址结构–> 连接服务端的过程后，创建了一个用于接收数据的线程 (使用上述的线程函数 ThreadFunction) 并定义了一个 Event 事件；然后进入了 while(true) 循环部分，在此循环中会不断地接收用户的终端输入内容并将其发送到服务端。若因为某些原因此循环 break，那么 main 函数接下来就会等待其之前创建的线程结束，然后关闭 SOCKET、清理资源，最后安全退出。

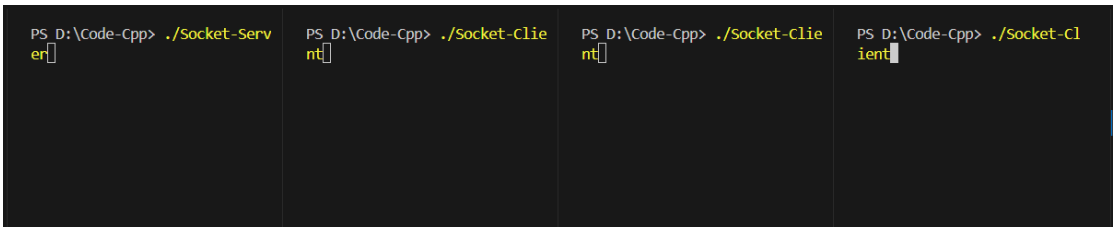
3. ’./exit 退出机制’：客户端输入’./exit’ 后表示此用户要退出聊天室，则客户端会向服务端发送一条信息 name+” Exit”，此时我们希望在接收信息的线程正常结束后再进行 SOCKET 的关闭和资源清理工作，而同样地由于 recv 处于阻塞态，我们必须在接收完服务端的一条消息后才能执行接下来的代码，因此，当服务端收到客户端发送的 name+” Exit” 信息后便会给客户端回复响应信息”OK! I got it”，客户端收到这条消息后便会进入特定处理部分，将全局变量 IsOver 置为 true，同时再发送一条消息”SuccessExit!”，接着关闭 SOCKET，循环 break；在循环 break 后便会调用 setevent() 设置事件，接着退出；当事件被设置后，main 函数中的 WaitForSingleObject() 函数等待结束，阻塞态结束，main 函数接着执行进行资源的清理工作后退出。再返回到服务端部分，当服务端收到消息”SuccessExit!” 后，便会清理这一用户对应的 SOCKET 接着此线程的 while 循环 break，线程结束。

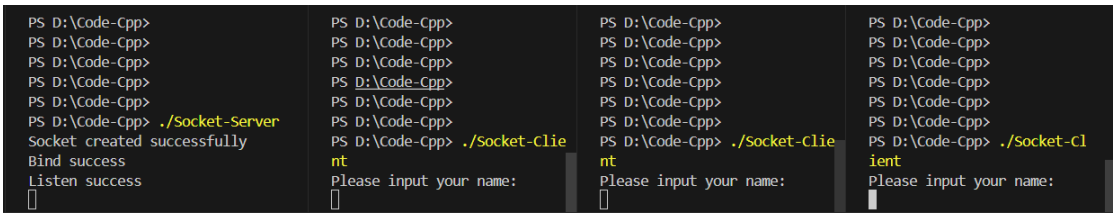
三、 程序运行说明及结果展示

使用如下指令编译 CPP 程序为 EXE 程序并执行：

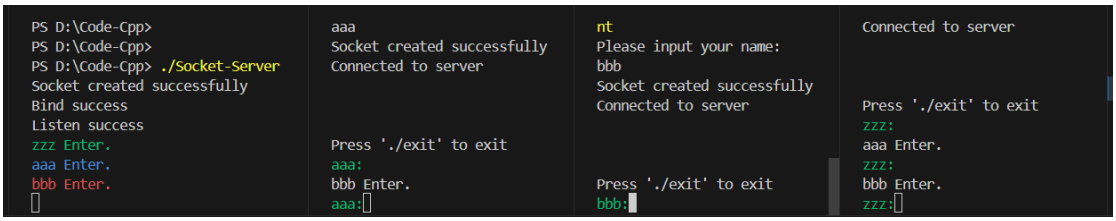
```
///服务端
g++ -o Socket-Server Socket-Server.cpp -lws2_32
./Socket-Server

///客户端
g++ -o Socket-Client Socket-Client.cpp -lws2_32
./Socket-Client
```

接下来分别运行服务端和客户端程序，为了展示多人聊天功能，下方展示部分启动了一个服务端程序以及三个客户端程序（服务端程序在第一个位置）



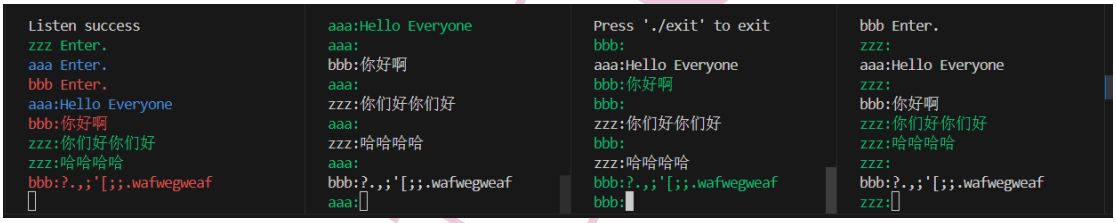


在启动程序后，服务端程序在创建完 SOCKET 并绑定地址后进入监听状态；而客户端程序则要求输入名称后才能进行下一步建立连接。



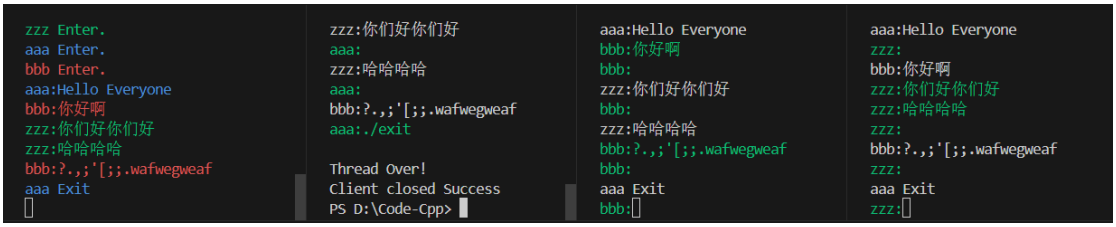
在客户端输入名称后，客户端程序会进一步创建 SOCKET 并连接到服务端，同时也可以看到服务端会出现相应的'XXX Enter' 的提示信息；另外对于客户端，先进入聊天室的用户 (本例中是 zzz) 在其他用户进入聊天室后也会得到服务器端发来的相应的提示信息。

接下来输入一系列信息来展示聊天功能。



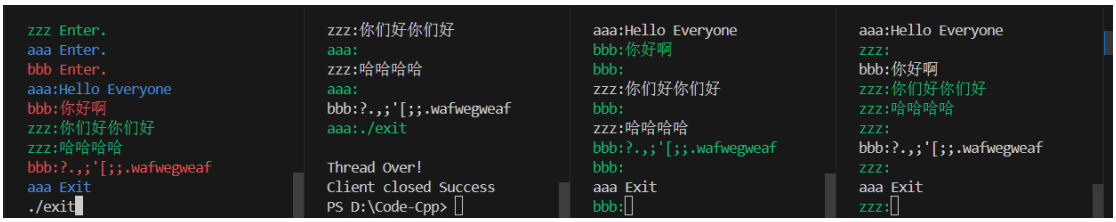
在上图展示的部分可以看到，在服务端部分完整地展示了三个用户的聊天内容并使用了不同的颜色进行区分；而在每个客户端部分则会显示除自己之外的其他用户发送的消息，并且自己发送的消息使用绿色，其他用户发送的消息使用白色来进行区分。另外在上图中还展示了此聊天程序提供中/英文的支持，并且对于空格这样的信息也可以正确处理。

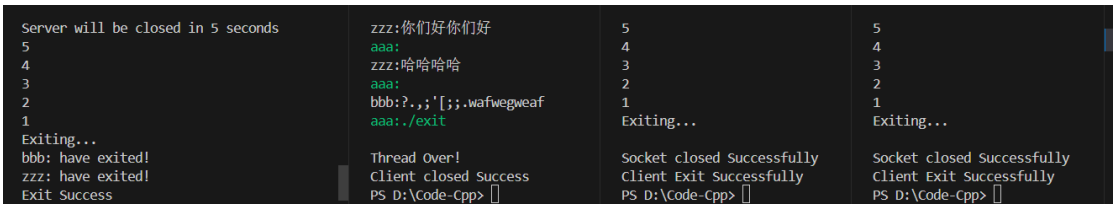
接下来展示程序的正常退出功能。



在客户端部分，提供退出聊天室功能：当用户输入'./exit' 后，便会退出聊天室，同时服务端以及其他的客户端也会得到用户退出的提示信息.(在此例中，用户 aaa 输入'./exit' 退出聊天室，服务端和其他的客户端都出现相应的用户 aaa 退出提示)

接下来是服务器端的退出展示。同样，输入'./exit' 后服务器端会退出，而在服务器端退出前会先关闭与各个客户端的连接。





如图中所展示的，在服务器端输入'./exit' 命令后，会先提示客户端" 将在 5 秒后退出"，之后各个客户端程序先后关闭连接，待关闭了所有客户端的连接后服务器端程序退出。

最后，程序中还提供了日志的记录，日志中记录了程序运行中的各种提示信息（包括连接创建、建立、退出、错误等内容）以及用户聊天信息等。下图展示了日志中的部分内容

四、 实验反思及分析

1. 此实验过程中我遇到的印象最深刻的问题就是在设置退出机制时，会出现一连串的问题。例如客户端要退出，那么在关闭 SOCKET 后，服务端可能因为客户端关闭 SOCKET 出

现错误；另外客户端这部分由于 SOCKET 的关闭，其处于阻塞态的 recv 等函数也会出现错误。最后我的解决方法是在某端退出前不要直接关闭 SOCKET，而是增加了一些客户端和服务端之间的信息交流，制定了一个退出流程。

2. 另一个比较麻烦的问题就是在控制台输出信息的格式上，比如当某个客户端正在输入信息时，此时另一个客户端发来一条消息，那么此用户已经输入的消息虽然已经写到了输入缓冲区中，但在控制台上已经与后续的输入分割开了，这是一个用户体验方面的问题。我对此问题的解决思路是通过 UI 界面来解决，将聊天框与输入框分隔开，例如微信那样；或者将输入部分和聊天信息部分拆分在两个终端中。

3. 最后一个问题是在退出时，主线程与其子线程之间信息如何传递的问题。正如上述代码部分所描述的，我设置了一个 bool 变量 TF 来表示程序是否要退出。