

浙江大学

本科实验报告

课程名称： 计算机网络基础

实验名称： 基于 Socket 接口实现自定义协议通信

姓 名： 何煦明

学 院： 计算机学院

系：

专 业： 计算机科学与技术

学 号： 3210101822

指导教师： 陆系群

2023 年 12 月 16 日

浙江大学实验报告

实验名称： 基于 Socket 接口实现自定义协议通信 实验类型： 编程实验

同组学生： 宋开石 实验地点： 计算机网络实验室

一、 实验目的

- 学习如何设计网络应用协议
- 掌握 Socket 编程接口编写基本的网络应用软件

二、 实验内容

根据自定义的协议规范，使用 Socket 编程接口编写基本的网络应用软件。

- 掌握 C 语言形式的 Socket 编程接口用法，能够正确发送和接收网络数据包
- 开发一个客户端，实现人机交互界面和与服务器的通信
- 开发一个服务端，实现并发处理多个客户端的请求
- 程序界面不做要求，使用命令行或最简单的窗体即可
- 功能要求如下：
 1. 运输层协议采用 TCP
 2. 客户端采用交互菜单形式，用户可以选择以下功能：
 - a) 连接：请求连接到指定地址和端口的服务端
 - b) 断开连接：断开与服务端的连接
 - c) 获取时间：请求服务端给出当前时间
 - d) 获取名字：请求服务端给出其机器的名称
 - e) 活动连接列表：请求服务端给出当前连接的所有客户端信息（编号、IP 地址、端口等）
 - f) 发消息：请求服务端把消息转发给对应编号的客户端，该客户端收到后显示在屏幕上
 - g) 退出：断开连接并退出客户端程序
 3. 服务端接收到客户端请求后，根据客户端传过来的指令完成特定任务：
 - a) 向客户端传送服务端所在机器的当前时间
 - b) 向客户端传送服务端所在机器的名称
 - c) 向客户端传送当前连接的所有客户端信息
 - d) 将某客户端发送过来的内容转发给指定编号的其他客户端
 - e) 采用异步多线程编程模式，正确处理多个客户端同时连接，同时发送消息的情况
- 根据上述功能要求，设计一个客户端和服务端之间的应用通信协议
- 本实验涉及到网络数据包发送部分不能使用任何的 Socket 封装类，只能使用最底层的 C 语言形式的 Socket API
- 本实验可组成小组，服务端和客户端可由不同人来完成

三、 主要仪器设备

- 联网的 PC 机、Wireshark 软件
- Visual C++、gcc 等 C++集成开发环境。

四、操作方法与实验步骤

- 设计请求、指示（服务器主动发给客户端的）、响应数据包的格式，至少要考虑如下问题：
 - a) 定义两个数据包的边界如何识别
 - b) 定义数据包的请求、指示、响应类型字段
 - c) 定义数据包的长度字段或者结尾标记
 - d) 定义数据包内数据字段的格式（特别是考虑客户端列表数据如何表达）
- 小组分工：1 人负责编写服务端，1 人负责编写客户端
- 客户端编写步骤（需要采用多线程模式）
 - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
 - b) 编写一个菜单功能，列出 7 个选项
 - c) 等待用户选择
 - d) 根据用户选择，做出相应的动作（未连接时，只能选连接功能和退出功能）
 1. 选择连接功能：请用户输入服务器 IP 和端口，然后调用 `connect()`，等待返回结果并打印。连接成功后设置连接状态为已连接。然后创建一个接收数据的子线程，循环调用 `receive()`，如果收到了一个完整的响应数据包，就通过线程间通信（如消息队列）发送给主线程，然后继续调用 `receive()`，直至收到主线程通知退出。
 2. 选择断开功能：调用 `close()`，并设置连接状态为未连接。通知并等待子线程关闭。
 3. 选择获取时间功能：组装请求数据包，类型设置为时间请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印时间信息。
 4. 选择获取名字功能：组装请求数据包，类型设置为名字请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印名字信息。
 5. 选择获取客户端列表功能：组装请求数据包，类型设置为列表请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印客户端列表信息（编号、IP 地址、端口等）。
 6. 选择发送消息功能（选择前需要先获得客户端列表）：请用户输入客户端的列表编号和要发送的内容，然后组装请求数据包，类型设置为消息请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印消息发送结果（是否成功送达另一个客户端）。
 7. 选择退出功能：判断连接状态是否为已连接，是则先调用断开功能，然后再退出程序。否则，直接退出程序。
 8. 主线程除了在等待用户的输入外，还在处理子线程的消息队列，如果有消息到达，则进行处理，如果是响应消息，则打印响应消息的数据内容（比如时间、名字、客户端列表等）；如果是指示消息，则打印指示消息的内容（比如服务器转发的别的客户端的消息内容、发送者编号、IP 地址、端口等）。
- 服务端编写步骤（需要采用多线程模式）
 - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
 - b) 调用 `bind()`，绑定监听端口（请使用学号的后 4 位作为服务器的监听端口），接着调用 `listen()`，设置连接等待队列长度
 - c) 主线程循环调用 `accept()`，直到返回一个有效的 `socket` 句柄，在客户端列表中增加一个新客户端的项目，并记录下该客户端句柄和连接状态、端口。然后创建一个子线程后继续调用 `accept()`。该子线程的主要步骤是（刚获得的句柄要传递给子线程，子线程内部要使用该句柄发送和接收数据）：

- ✧ 调用 `send()`，发送一个 `hello` 消息给客户端（可选）
- ✧ 循环调用 `receive()`，如果收到了一个完整的请求数据包，根据请求类型做相应的动作：
 1. 请求类型为获取时间：调用 `time()` 获取本地时间，然后将时间数据组装进响应数据包，调用 `send()` 发给客户端
 2. 请求类型为获取名字：将服务器的名字组装进响应数据包，调用 `send()` 发给客户端
 3. 请求类型为获取客户端列表：读取客户端列表数据，将编号、IP 地址、端口等数据组装进响应数据包，调用 `send()` 发给客户端
 4. 请求类型为发送消息：根据编号读取客户端列表数据，如果编号不存在，将错误代码和出错描述信息组装进响应数据包，调用 `send()` 发回源客户端；如果编号存在并且状态是已连接，则将要转发的消息组装进指示数据包。调用 `send()` 发给接收客户端（使用接收客户端的 `socket` 句柄），发送成功后组装转发成功的响应数据包，调用 `send()` 发回源客户端。
- d) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 `Socket`，主程序退出。
- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

五、实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

- 源代码：客户端和服务端的代码分别在一个目录
- 可执行文件：可运行的 `.exe` 文件或 `Linux` 可执行文件，客户端和服务端各一个
- 描述请求数据包的格式（画图说明），请求类型的定义

请求数据包的格式如下：

请求类型	数据包格式
c	<div style="border: 1px solid black; padding: 2px; display: inline-block;">c</div>
t	<div style="border: 1px solid black; padding: 2px; display: inline-block;">t</div>
n	<div style="border: 1px solid black; padding: 2px; display: inline-block;">n</div>
l	<div style="border: 1px solid black; padding: 2px; display: inline-block;">l</div>
s	<div style="border: 1px solid black; padding: 2px; display: inline-block;">s receiver_id & message</div>

其中，`receiver_id` 表示接收方 `id`，`&`表示分隔符，`message` 表示发送的消息

请求数据包的定义如下：

请求类型	定义
c	请求服务器返回本客户端的 ID
t	请求服务器返回本地时间
n	请求服务器返回服务端的机器名
l	请求服务器返回当前连接的所有客户端 ID
s	向连接在同一服务器上的另一个用户发送指定信息

- 描述响应数据包的格式（画图说明），响应类型的定义

响应数据包的格式如下：

响应类型	数据包格式
c	<div>c user_id</div>
t	<div>t date & time</div>
n	<div>n server_name</div>
l	<div>l tot_usr_num usr_id1 ... usr_idn</div>
e	<div>e feedback_msg</div>

其中，`user_id` 表示该客户端的 `id`，`date&time` 表示服务端本地时间，`server_name` 表示服务端机器名，`tot_usr_num` 表示连接用户总数，`usr_id1` 表示第一个用户 `id`，`usr_idn` 表示第 `n` 个用户 `id`，`feedback_msg` 表示发送失败或者成功

响应数据包的定义如下：

响应类型	定义
c	服务器当前客户端 ID
t	服务器返回本地时间
n	服务器返回服务端的机器名
l	服务器返回当前连接的所有客户端 ID
e	服务器返回当前信息是否发送成功

- 描述指示数据包的格式（画图说明），指示类型的定义

指示数据包的格式如下：

响应类型	数据包格式			
s	<table><tr><td>s</td><td>sender_id</td><td>message</td></tr></table>	s	sender_id	message
s	sender_id	message		

其中 sender_id 表示发送方 id, message 表示信息

指示类型的定义如下:

指示类型	定义
s	服务器向目标客户端转发的信息

- 客户端初始运行后显示的菜单选项

```
[System]Please enter your user name:abc
[System]Hello, abc!
[System]Supported functions are as follows:
+-----+-----+
|      Command      |      Function      |
+-----+-----+
|      -c           | Connect to the chosen server. |
|      -e           | Exit.               |
|      -h           | Get help.           |
+-----+-----+
[abc]
```

这里系统首先要求用户输入用户名, 之后会弹出未连接状态下的可用命令列表。

- 客户端的主线程循环关键代码截图 (描述总体, 省略细节部分)

```
void Handle()
{
    char str[100];
    while(1){
        scanf("%s", str);
        if(str[0]!='-'){
            PrintSystemPrefix();
            printf("Not a command.\n");
        }
        else{
            char command=str[1];
            switch(command){
                case 'c':
                    connect_status=Connect();
                    break;
                case 'd':
                    Disconnect();
                    PrintUserPrefix();
                    break;
                case 't':
                    GetTime();
                    break;
                case 'n':
                    GetName();
                    break;
                case 'l':
                    UserList();
                    break;
                case 's':
                    Send();
                    break;
                case 'e':
                    Exit();
                    break;
                case 'h':
                    PrintMenu();
                    break;
                default:
                    PrintSystemPrefix();
                    printf("Not a command.\n");
                    PrintUserPrefix();
                    break;
            }
        }
    }
}
```

- 客户端的接收数据子线程循环关键代码截图（描述总体，省略细节部分）

```
DWORD WINAPI RecvHandleThread(LPVOID lpParameter)
{
    SOCKET s;
    cIntSock=(SOCKET*)lpParameter;
    char RecvBuff[MAXBUFSIZE];
    while(1){
        int n=0;
        n=recv(*cIntSock, RecvBuff, sizeof(RecvBuff), 0);
        if(n>0){
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN); //黄色
            char command=RecvBuff[0];
            switch(command){
                case 'e':
                    //服务器对请求包的反馈
                    printf("\n");
                    PrintServerPrefix();
                    printf("%s\n", (char*)(RecvBuff+1));
                    PrintUserPrefix();
                    break;
                case 't':
                    PrintServerPrefix();
                    printf("No.%d Current time: %s", cnt, (char*)(RecvBuff+1));
                    cnt++;
                    PrintUserPrefix();
                    break;
                case 'n':
                    PrintServerPrefix();
                    printf("Server's PC name: %s.\n", (char*)(RecvBuff+1));
                    PrintUserPrefix();
                    break;
                case 'l':{
                    PrintServerPrefix();
                    int numClients=RecvBuff[1];
                    printf("Number of clients connected to the server: %d\n", numClients);
                    printf("-----+\n");
                    printf("      |      User Id List      |\n");
                    printf("-----+\n");
                    int offset=2;
                    for(int i=0; i<numClients; i++){
                        char usr_id[4];
                        memcpy(usr_id, RecvBuff+offset, 3);
                        usr_id[3]='\0';
                        printf("      |      %d      |\n", atoi(usr_id));
                        offset+=3;
                    }
                    printf("-----+\n");
                    PrintUserPrefix();
                    break;
                }
                case 's':{
                    Sleep(20);
                    printf("\n");
                    PrintServerPrefix();
                    char src_id[4];
                    memcpy(src_id, RecvBuff+1, 3);
                    printf("Message from [%s]: %s\n", src_id, (char*)(RecvBuff+4));
                    PrintUserPrefix();
                    break;
                }
                case 'c':{
                    printf("\n");
                    PrintServerPrefix();
                    char id[10];
                    strcpy(id, RecvBuff+1);
                    printf("User ID: %s.\n", id);
                    user_id=atoi(id);
                    PrintUserPrefix();
                    break;
                }
                default:
                    break;
            }
        }
    }
}
```

- 服务器初始运行后显示的界面

```
[Initialize] Socket initialized!
[Initialize] Bind port 5815 successful!
[Initialize] Waiting for clients ...
```

- 服务器的主线程循环关键代码截图（描述总体，省略细节部分）


```

int main() {
    hConsole = GetStdHandle(STD_OUTPUT_HANDLE); // 用于更改控制台输出颜色
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    WSADATA wsd;
    WSAStartup(MAKEWORD(2, 2), &wsd);
    SOCKET ListenSocket = socket(AF_INET, SOCK_STREAM, 0);
    PrintInitialPrefix();
    cout << "Socket initialized!" << endl;
    SOCKADDR_IN ListenAddr;
    ListenAddr.sin_family = AF_INET;
    ListenAddr.sin_addr.S_un.S_addr = INADDR_ANY; // 表示通入本机ip
    ListenAddr.sin_port = htons(PORT);
    int n;
    n = bind(ListenSocket, (LP SOCKADDR)&ListenAddr, sizeof(ListenAddr));
    if (n == SOCKET_ERROR) {
        PrintInitialPrefix();
        cout << "Bind failed!" << endl;
        return -1;
    }
    else {
        PrintInitialPrefix();
        cout << "Bind port ";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << PORT;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << " successful!" << endl;
    }
    int l = listen(ListenSocket, 20);
    PrintInitialPrefix();
    cout << "Waiting for clients ..." << endl;

    while (1) {
        // 循环接收客户端连接请求并创建服务线程
        SOCKET* ClientSocket = new SOCKET;
        ClientSocket = (SOCKET*)malloc(sizeof(SOCKET));
        // 接收客户端连接请求
        int SockAddrLen = sizeof(sockaddr);
        *ClientSocket = accept(ListenSocket, 0, 0);
        cClients.push_back(make_pair(ClientSocket, int(*ClientSocket)));

        PrintSystemPrefix();
        cout << "Client(";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << *ClientSocket;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << ") has been connected" << endl;
        CreateThread(NULL, 0, &ServerThread, ClientSocket, 0, NULL);
    }
    closesocket(ListenSocket);
    WSACleanup();
}

```

- 服务器的客户端处理子线程循环关键代码截图（描述总体，省略细节部分）

```

DWORD WINAPI ServerThread(LPVOID lpParameter) {
    SOCKET* ClientSocket = (SOCKET*)lpParameter;
    int receByt = 0;
    char RecvBuf[MaxBufSize];
    char SendBuf[MaxBufSize];
    while (1) {
        receByt = recv(*ClientSocket, RecvBuf, sizeof(RecvBuf), 0);
        if (receByt > 0) {
            PrintSystemPrefix();
            cout << "Receive message: ";
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
            cout << RecvBuf[0];
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
            cout << " -- from client ";
            SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
            cout << *ClientSocket << endl;
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);

            switch (RecvBuf[0])
            {
            {
                case 't': // 服务端所在机器的当前时间
                {
                    time_t now = time(0); // 把 now 转换为字符串形式
                    char* dt = ctime(&now);
                    PrintSystemPrefix();
                    cout << "Local date & time: ";
                    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
                    cout << dt;
                    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
                    SendBuf[0] = 't';
                    strcpy(SendBuf + 1, dt);

                    int k = 0;
                    k = send(*ClientSocket, SendBuf, sizeof(SendBuf), 0);
                    if (k < 0) {
                        PrintSystemPrefix();
                        cout << "Send failed!" << endl;
                    }
                }

                memset(SendBuf, 0, sizeof(SendBuf));
                break;
            }
            }
        }
    }
}

```



```

case 'n': // 服务器所在机器的名称
{
    DWORD ComputerNameLength = 100;
    strcpy(SendBuf, "n");

    TCHAR t_name[100];
    GetComputerNameW((LPWSTR)t_name, &ComputerNameLength); // 获取本机名称
    char name[100];
    wcstombs(name, (wchar_t *)t_name, 100);
    strcat(SendBuf, name);

    int k = 0;
    k = send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
    if (k < 0) {
        PrintSystemPrefix();
        cout << "Send failed!" << endl;
    }
    PrintSystemPrefix();
    cout << "Server's PC name: ";
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
    cout << SendBuf + 1 << endl;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);

    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}

case '!': // 当前连接的所有客户端信息
{
    cout << "      User id list:";
    for (auto iter : clients)
    {
        cout << " " << iter.second;
    }
    cout << endl;
    SendBuf[0] = '!';
    SendBuf[1] = clients.size();
    char Buf[4] = { 0 };
    for (auto iter : clients)
    {
        char Buf[4] = { 0 };
        itoa(iter.second, Buf, 10);
        strcat(SendBuf, Buf);
    }

    int k = 0;
    k = send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
    if (k < 0) {
        PrintSystemPrefix();
        cout << "Send failed!" << endl;
    }

    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}
}

```

```

case 's':
{
    char dst_port[10];
    // 提取接收方id
    int idx = 1;
    while (RecvBuf[idx] != '&') {
        dst_port[idx - 1] = RecvBuf[idx];
        idx++;
    }

    int dst_port_id = atoi(dst_port);
    SOCKET* dst_client = NULL;

    for (auto iter : clients)
    {
        if (iter.second == dst_port_id)
        {
            dst_client = iter.first;
            break;
        }
    }
    if (!dst_client)
    {
        PrintSystemPrefix();
        cout << "Client(";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << dst_port_id;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << ") is not connected" << endl;
        // 发个反馈包
        strcpy(SendBuf, "Invalid receiver ID!");
        send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
        break;
    }
}

```

```

        memset(SendBuf, 0, sizeof(SendBuf));
        SendBuf[0] = 's';
        char Buf[4] = { 0 };
        for (auto iter : clients)
        {
            if (iter.first == ClientSocket)
            {
                itoa(iter.second, Buf, 10);
                strcat(SendBuf, Buf);
                break;
            }
        }
        strcat_s(SendBuf, RecvBuf + idx + 1);

        int k = 0;
        k = send(dst_client, SendBuf, sizeof(SendBuf), 0);
        if (k < 0) {
            PrintSystemPrefix();
            cout << "Send failed!" << endl;
        }
        PrintSystemPrefix();
        cout << "Sender(";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << "ClientSocket;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << ") -- Receiver(";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << "dst_client << ")" << endl;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << "    Message is: ";
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
        cout << SendBuf + 4 << endl;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        // 发个反馈包
        strcpy(SendBuf, "eSuccessfully sent!");
        send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
        memset(SendBuf, 0, sizeof(SendBuf));
        break;
    }
    case 'c': // 返回用户id
    {
        strcpy(SendBuf, "c");
        int id = ClientSocket;
        char usr_id[10];
        strcat(SendBuf, itoa(id, usr_id, 10));
        int k = 0;
        k = send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
        if (k < 0) {
            PrintSystemPrefix();
            cout << "Send failed!" << endl;
        }
    }

    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}
default:
    cout << "Undefined Behavior." << endl;
    break;
}
}
else
{
    PrintSystemPrefix();
    cout << "Client(";
    SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
    cout << "ClientSocket;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    cout << ") has been disconnected" << endl;

    for (auto iter = clients.begin(); iter != clients.end(); iter++) {
        if ((*iter).first == ClientSocket) {
            iter = clients.erase(iter);
            break;
        }
    }
    break;
}

memset(RecvBuf, 0, sizeof(RecvBuf));
}

closesocket(ClientSocket);
free(ClientSocket);
return 0;
}

```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

客户端：

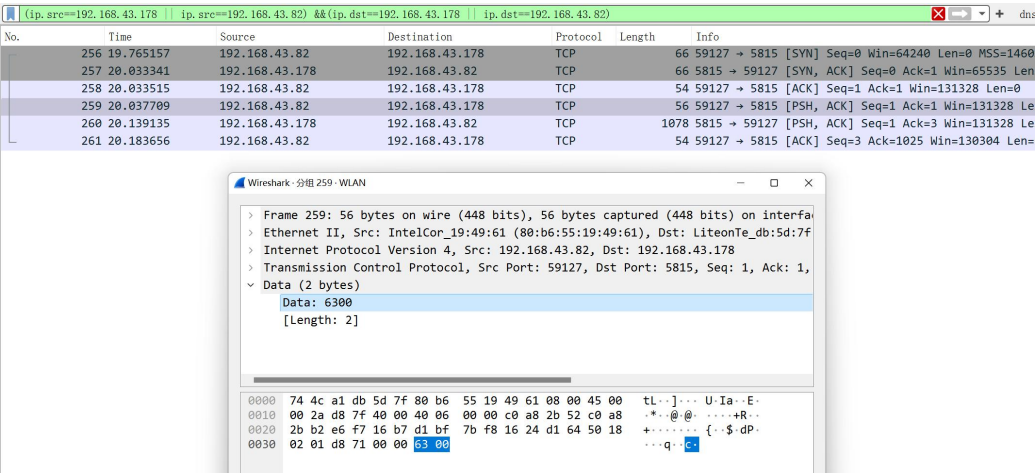
```
[abc]-c
[System]Please enter server's ip: 192.168.43.178
[System]Please enter server's port: 5815
[System]Success to connect to server 192.168.43.178.
[Server]User ID: 156.
```

服务端:

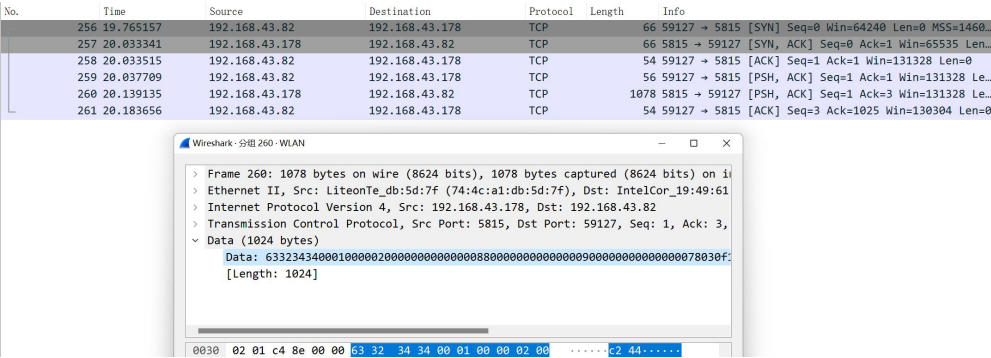
```
[System] Client(244) has been connected
[System] Receive message: c -- from client 244
```

Wireshark 抓取的数据包截图:

客户端向服务器发送“c”数据包:



服务端向客户端返回“c244”数据包(244 为 user_id):



- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

客户端:

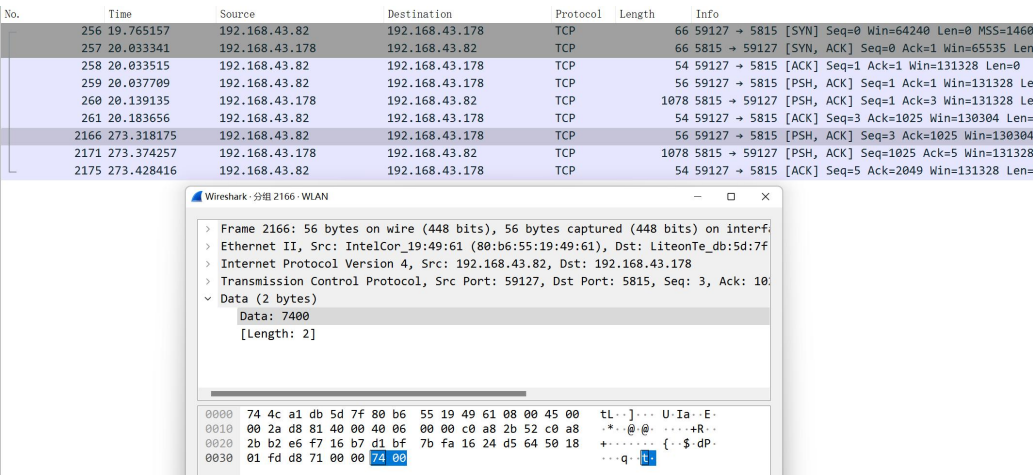
```
[abc]-t
[Server]Current time: Sat Dec 16 19:16:22 2023
```

服务端:

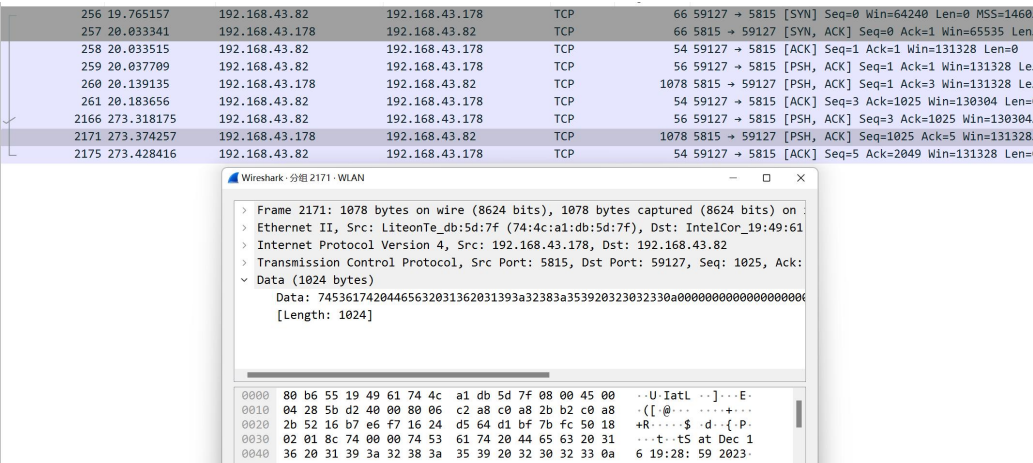
```
[System] Receive message: t -- from client 244
[System] Local date & time: Sat Dec 16 19:28:59 2023
```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的时间数据对应的）：

客户端向服务端发送“t”请求获取数据：



服务端向客户端返回“t+本地时间”：



- 客户端选择获取名字功能时，客户端和服务端显示内容截图。

客户端：

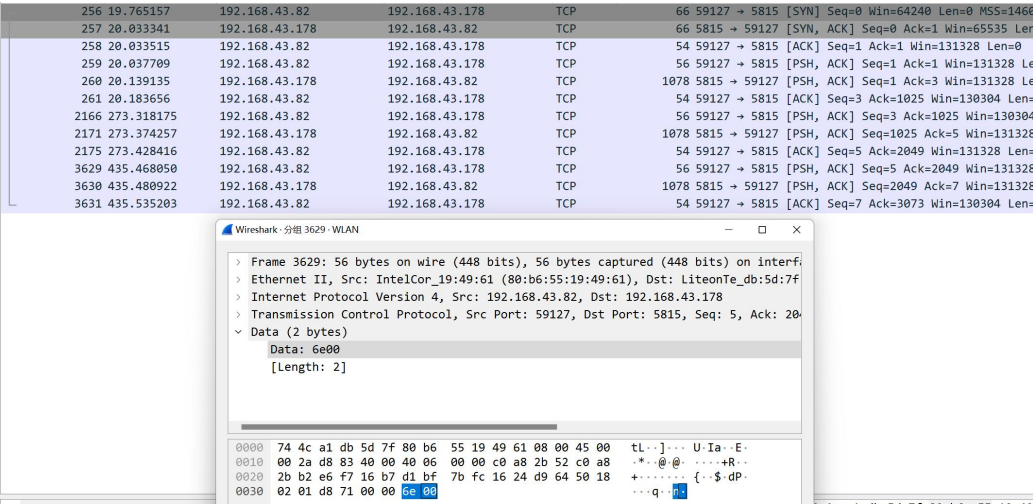
```
[abc]-n
[Server]Server's PC name: LAPTOP-HE0QJB9P.
```

服务端：

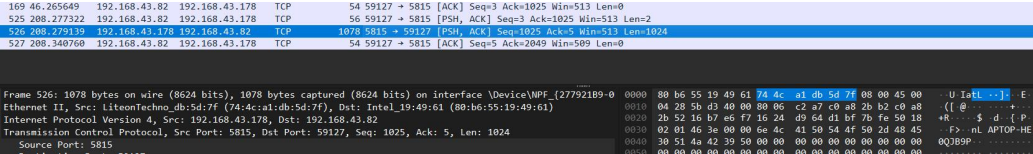

```
[System] Receive message: n -- from client 244
[System] Server's PC name: LAPTOP-HE0QJB9P
```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的名字数据对应的位置）：

客户端向服务端获取名字 “n”：



服务端向客户端返回 “n+机器名”：



相关的服务器的处理代码片段：

```
case 'n': // 服务端所在机器的名称
{
    DWORD ComputerNameLength = 100;
    strcpy(SendBuf, "n");

    TCHAR t_name[100];
    GetComputerNameW((LPWSTR)t_name, &ComputerNameLength); // 获取本机名称
    char name[100];
    wcstombs(name, (wchar_t *)t_name, 100);
    strcat(SendBuf, name);

    int k = 0;
    k = send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
    if (k < 0) {
        PrintSystemPrefix();
        cout << "Send failed!" << endl;
    }
    PrintSystemPrefix();
    cout << "Server's PC name: ";
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
    cout << SendBuf + 1 << endl;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);

    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}
```

- 客户端选择获取客户端列表功能时，客户端和服务端显示内容截图。

客户端

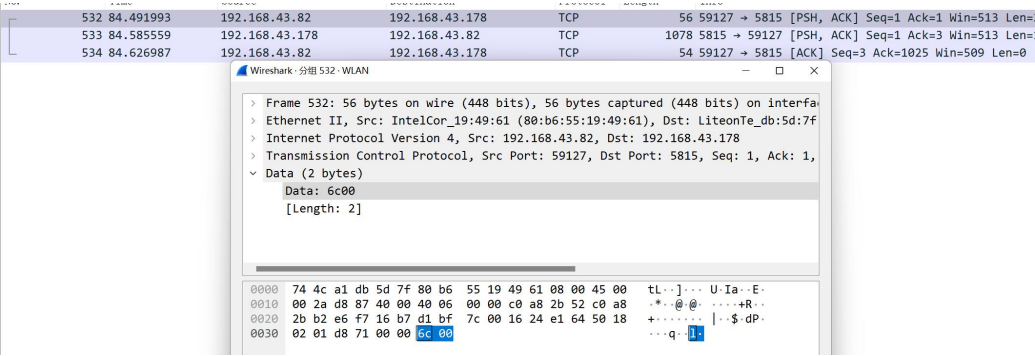
```
[abc]-l
[Server]Number of clients coonected to the server: 1
+-----+
|      User Id List      |
+-----+
|           244          |
+-----+
```

服务端：

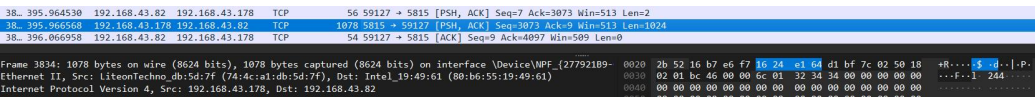
```
[System] Receive message: 1 -- from client 244
User id list: 244
```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：

客户端向服务端发送“1”：



服务器向客户端发送“1+用户列表”：



相关的服务器的处理代码片段：

```
case '1': // 当前连接的所有客户端信息
{
    cout << "      User id list:";
    for (auto iter : clients)
    {
        cout << " " << iter.second;
    }
    cout << endl;
    SendBuf[0] = '1';
    SendBuf[1] = clients.size();
    char Buf[4] = { 0 };
    for (auto iter : clients)
    {
        char Buf[4] = { 0 };
        itoa(iter.second, Buf, 10);
        strcat(SendBuf, Buf);
    }

    int k = 0;
    k = send(ClientSocket, SendBuf, sizeof(SendBuf), 0);
    if (k < 0) {
        PrintSystemPrefix();
        cout << "Send failed!" << endl;
    }

    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}
```

- 客户端选择发送消息功能时，客户端和服务端显示内容截图。

发送消息的客户端：

```
[abc]-s
[System]Please enter receiver's id: 244
[System]Please enter message to be sent: hello

[Server]Message from [244]: hello
[abc]
[Server]Successfully sent!
```

服务器：

```
[System] Receive message: s -- from client 244
[System] Sender(244) -- Receiver(244)
Message is: hello
```

接收消息的客户端：

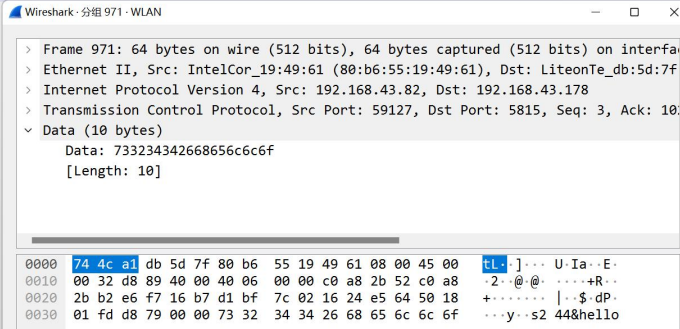
```
[abc]-s
[System]Please enter receiver's id: 244
[System]Please enter message to be sent: hello

[Server]Message from [244]: hello
[abc]
[Server]Successfully sent!
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送方向服务器发送 “s+接收方 id+&+发送内容”：

532	84.491993	192.168.43.82	192.168.43.178	TCP	56	59127 → 5815 [PSH, ACK] Seq=1 Ack=1 Win=0 Len=0
533	84.585559	192.168.43.178	192.168.43.82	TCP	1078	5815 → 59127 [PSH, ACK] Seq=1 Ack=3 Win=0 Len=0
534	84.626987	192.168.43.82	192.168.43.178	TCP	54	59127 → 5815 [ACK] Seq=3 Ack=1025 Win=0 Len=0
971	150.671831	192.168.43.82	192.168.43.178	TCP	64	59127 → 5815 [PSH, ACK] Seq=3 Ack=1025 Win=0 Len=10
972	150.735550	192.168.43.178	192.168.43.82	TCP	1078	5815 → 59127 [PSH, ACK] Seq=1025 Ack=1 Win=0 Len=0
973	150.789915	192.168.43.82	192.168.43.178	TCP	54	59127 → 5815 [ACK] Seq=13 Ack=2049 Win=0 Len=0
974	150.837300	192.168.43.178	192.168.43.82	TCP	1078	5815 → 59127 [PSH, ACK] Seq=2049 Ack=1 Win=0 Len=0
975	150.883211	192.168.43.82	192.168.43.178	TCP	54	59127 → 5815 [ACK] Seq=13 Ack=3073 Win=0 Len=0



服务器向接收方发送 “s+发送方 id+发送信息”：

42_ 462.119287	192.168.43.178	192.168.43.82	TCP	1078 5815 → 59127 [PSH, ACK] Seq=4097 Ack=19 Win=513 Len=1024	
42_ 462.219690	192.168.43.82	192.168.43.178	TCP	54 59127 → 5815 [ACK] Seq=19 Ack=5121 Win=513 Len=0	
42_ 462.219800	192.168.43.178	192.168.43.82	TCP	1078 5815 → 59127 [PSH, ACK] Seq=5121 Ack=19 Win=513 Len=1024	
42_ 462.321594	192.168.43.82	192.168.43.178	TCP	54 59127 → 5815 [ACK] Seq=19 Ack=6145 Win=509 Len=0	

Frame 4249: 1078 bytes on wire (8624 bits), 1078 bytes captured (8624 bits) on interface \Device\NPF_{277921B...}	0020	2b 52 16 b7 e6 f7 16 24 e5 64 d1 bf 7c 0c 50 18	*R... 5 d -[-P-
Ethernet II, Src: LiteonTechno_db:5d:7f (74:4c:a1:db:5d:7f), Dst: Intel_19:49:61 (80:b6:55:19:49:61)	0030	02 01 5f 39 00 00 73 32 34 34 68 05 6c 6c 6f 00	...9..s2 44hello
	0040	6c 6c 79 20 73 05 6e 74 21 00 00 00 00 00 00	

服务端向发送方发送“e+反馈信息”：

42_ 462.219800	192.168.43.178	192.168.43.82	TCP	1078 5815 → 59127 [PSH, ACK] Seq=5121 Ack=19 Win=513 Len=1024	
42_ 462.321594	192.168.43.82	192.168.43.178	TCP	54 59127 → 5815 [ACK] Seq=19 Ack=6145 Win=509 Len=0	

Frame 4251: 1078 bytes on wire (8624 bits), 1078 bytes captured (8624 bits) on interface \Device\NPF_{277921B...}	0020	2b 52 16 b7 e6 f7 16 24 e9 64 d1 bf 7c 0c 50 18	*R... 5 d -[-P-
Ethernet II, Src: LiteonTechno_db:5d:7f (74:4c:a1:db:5d:7f), Dst: Intel_19:49:61 (80:b6:55:19:49:61)	0030	02 01 86 06 00 00 65 53 75 63 63 65 73 73 66 75e5 successfu
Internet Protocol Version 4, Src: 192.168.43.178, Dst: 192.168.43.82	0040	6c 6c 79 20 73 05 6e 74 21 00 00 00 00 00 00	lly sent !

相关的服务器的处理代码片段：

```

case 's':
{
    char dst_port[10];
    // 提取接收方id
    int idx = 1;
    while (RecvBuf[idx] != '&') {
        dst_port[idx - 1] = RecvBuf[idx];
        idx++;
    }

    int dst_port_id = atoi(dst_port);
    SOCKET* dst_client = NULL;

    for (auto iter : clients)
    {
        if (iter.second == dst_port_id)
        {
            dst_client = iter.first;
            break;
        }
    }
    if (!dst_client)
    {
        PrintSystemPrefix();
        cout << "Client(";
        SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
        cout << dst_port_id;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
        cout << ") is not connected" << endl;
        // 发个反馈包
        strcpy(SendBuf, "eInvalid receiver ID!");
        send(*ClientSocket, SendBuf, sizeof(SendBuf), 0);
        break;
    }

    memset(SendBuf, 0, sizeof(SendBuf));
    SendBuf[0] = 's';
    char Buf[4] = { 0 };
    for (auto iter : clients)
    {
        if (iter.first == ClientSocket)
        {
            itoa(iter.second, Buf, 10);
            strcat(SendBuf, Buf);
            break;
        }
    }
    strcat_s(SendBuf, RecvBuf + idx + 1);

    int k = 0;
    k = send(*dst_client, SendBuf, sizeof(SendBuf), 0);
    if (k < 0) {
        PrintSystemPrefix();
        cout << "Send failed!" << endl;
    }
    PrintSystemPrefix();
    cout << "Sender(";
    SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
    cout << *ClientSocket;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    cout << ") -- Receiver(";
    SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
    cout << "dst_client << ")" << endl;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    cout << "    Message is: ";
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
    cout << SendBuf + 4 << endl;
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    // 发个反馈包
    strcpy(SendBuf, "eSuccessfully sent!");
    send(*ClientSocket, SendBuf, sizeof(SendBuf), 0);
    memset(SendBuf, 0, sizeof(SendBuf));
    break;
}

```

相关的客户端（发送和接收消息）处理代码片段：

发送：

```
void Send()
{
    if(connect_status){
        int _ReceiverId;
        char Message[MAXBUFFSIZE];
        char SendBuff[MAXBUFFSIZE+12]="s";
        char ReceiverId[10];
        //读取receiver id
        PrintSystemPrefix();
        printf("Please enter receiver's id: ");
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN); //黄色
        scanf("%d", &_ReceiverId);
        getchar();
        _itoa(_ReceiverId, ReceiverId, 10);
        //拼接"&"来区分id和message
        strcat(ReceiverId, "&");
        //读取发送信息
        PrintSystemPrefix();
        printf("Please enter message to be sent: ");
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN); //黄色
        //要忽略空格，否则读不全
        CustomizeRead(Message, MAXBUFFSIZE);
        strcat(SendBuff, ReceiverId);
        strcat(SendBuff, Message);
        int status=send(servSock, SendBuff, strlen(SendBuff), 0);
        if(status<0){
            PrintSystemPrefix();
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
            printf("Send failed");
            SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
            printf(".\n");
            return;
        }
    }
    else{
        //尚未连接
        PrintSystemPrefix();
        printf("No server connected.\n");
        PrintUserPrefix();
    }
}
```

接收：

```
case 's':{
    Sleep(20);
    printf("\n");
    PrintServerPrefix();
    char src_id[4];
    memcpy(src_id, RecvBuff+1, 3);
    printf("Message from [%s]: %s\n", src_id, (char*)(RecvBuff+4));
    PrintUserPrefix();
    break;
}
```

- 拔掉客户端的网线，然后退出客户端程序。观察客户端的 TCP 连接状态，并使用 Wireshark 观察客户端是否发出了 TCP 连接释放的消息。同时观察服务端的 TCP 连接状态在较长时间内（10 分钟以上）是否发生变化。

在设备上运行用户 id 为 244 的客户端，连接 IP 地址为 192.168.43.178，端口号为 5815 的服务端。两者成功建立联系如下：

TCP	192.168.43.178:139	0.0.0.0:0	LISTENING
TCP	192.168.43.178:5815	192.168.43.82:59127	ESTABLISHED
TCP	192.168.43.178:49163	20.187.186.89:443	ESTABLISHED

关闭客户端网络连接等待 10min 后，通过 netstat -an 指令查看发现客户端的 TCP 连接状态仍为 ESTABLISHED。且服务端并未显示该用户断开连接的提示。

- 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

客户端查询用户列表，发现异常连接的客户端(244)仍然在用户列表中：

```
[abc]-c
[System]Please enter server's ip: 192.168.43.178
[System]Please enter server's port: 5815
[System]Success to connect to server 192.168.43.178.
[Server]User ID: 232.
[abc]-l
[Server]Number of clients coonected to the server: 2

+-----+
|      User Id List      |
+-----+
|          244           |
|          232           |
+-----+
```

用新的客户端(232)向异常客户端(244)发送信息，发现显示异常用户 id:

```
[abc]-s
[System]Please enter receiver's id: 244
[System]Please enter message to be sent: hello
[Server]Invalid receiver ID!
```

再次请求用户列表，发现异常客户端(244)已经断开连接:

```
[abc]-l
[Server]Number of clients coonected to the server: 1

+-----+
|      User Id List      |
+-----+
|          232           |
+-----+
```

Wireshark 抓包记录:

```
10_ 1341.052806 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1341.506374 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1342.406056 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1344.204830 192.168.43.178 192.168.43.82 TCP 590 [TCP Retransmission] 5815 → 59127 [ACK] Seq=10241 Ack=39 Win=513 Len=536
10_ 1346.005012 192.168.43.178 192.168.43.82 TCP 590 [TCP Retransmission] 5815 → 59127 [ACK] Seq=10241 Ack=39 Win=513 Len=536
10_ 1347.810128 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1351.400340 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1358.569000 192.168.43.178 192.168.43.82 TCP 1078 [TCP Retransmission] 5815 → 59127 [PSH, ACK] Seq=10241 Ack=39 Win=513 Len=1024
10_ 1372.917952 192.168.43.178 192.168.43.82 TCP 54 5815 → 59127 [RST, ACK] Seq=11265 Ack=39 Win=0 Len=0
```

发现服务端连续向异常客户端重传多个[PSH,ACK]包均失败随后服务端释放线程。此后再次在新客户端(232)上获取用户连接列表，发送信息等功能都可以正常返回。

- 修改获取时间功能，改为用户选择 1 次，程序内自动发送 100 次请求。服务器是否正常处理了

100 次请求，截取客户端收到的响应（通过程序计数一下是否有 100 个响应回来），并使用 Wireshark 抓取数据包，观察实际发出的数据包个数。

```
void GetTime()
{
    if(connect_status){
        char SendBuff[2]="t";
        for(int i=0; i<100; i++){
            send(servSock, SendBuff, sizeof(SendBuff), 0);
            Sleep(5);
        }
    }
    else{
        //尚未连接
        PrintSystemPrefix();
        printf("No server connected.\n");
        PrintUserPrefix();
    }
}
```

[illegible]

```
[abc][Server]No.34 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.35 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.36 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.37 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.38 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.39 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.40 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.41 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.42 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.43 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.44 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.45 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.46 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.47 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.48 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.49 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.50 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.51 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.52 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.53 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.54 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.55 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.56 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.57 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.58 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.59 Current time: Sat Dec 16 20:17:42 2023
[abc][Server]No.60 Current time: Sat Dec 16 20:17:43 2023
```

发现客户端发送了 60 次时间请求

——在这里我们还测试了 `Sleep(1)` 的情况，发现客户端收到了 2 次响应，同时在 wireshark 上抓包也显示实际发出了 2 个数据包

- 多个客户端同时连接服务器，同时发送时间请求（程序内自动连续调用 100 次 `send`），服务器和客户端的运行截图

客户端 1:

```
[abc]-t
[Server]No.0 Current time: Sat Dec 16 20:16:06 2023
[abc][Server]No.1 Current time: Sat Dec 16 20:16:06 2023
[abc][Server]No.2 Current time: Sat Dec 16 20:16:06 2023
[abc][Server]No.3 Current time: Sat Dec 16 20:16:06 2023
[abc][Server]No.4 Current time: Sat Dec 16 20:16:07 2023
[abc][Server]No.5 Current time: Sat Dec 16 20:16:07 2023
[abc][Server]No.6 Current time: Sat Dec 16 20:16:07 2023
[abc][Server]No.7 Current time: Sat Dec 16 20:16:08 2023
[abc][Server]No.8 Current time: Sat Dec 16 20:16:08 2023
```

客户端 2:


```

[sks] [Server] No. 89 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 90 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 91 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 92 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 93 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 94 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 95 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 96 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 97 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 98 Current time: Sat Dec 16 20:16:07 2023
[sks] [Server] No. 99 Current time: Sat Dec 16 20:16:07 2023
[sks]

```

服务端：

```

[System] Receive message: t -- from client 236
[System] Num = 103 Local date & time: Sat Dec 16 20:16:07 2023
[System] Receive message: t -- from client 236
[System] Num = 104 Local date & time: Sat Dec 16 20:16:07 2023
[System] Receive message: t -- from client 236
[System] Num = 105 Local date & time: Sat Dec 16 20:16:07 2023
[System] Receive message: t -- from client 236
[System] Num = 106 Local date & time: Sat Dec 16 20:16:07 2023
[System] Receive message: t -- from client 148
[System] Num = 107 Local date & time: Sat Dec 16 20:16:08 2023
[System] Receive message: t -- from client 148
[System] Num = 108 Local date & time: Sat Dec 16 20:16:08 2023

```

这里我们均是从 0 开始计数，因此客户端 1 收到的响应包为 9 个，客户端 2 收到的响应包为 100 个，服务端发出的响应包为 109 个，这是对应的。

六、实验结果与分析

- 客户端是否需要调用 bind 操作？它的源端口是如何产生的？每一次调用 connect 时客户端的端口是否都保持不变？

客户端不需要调用 bind 操作，源端口在 connect 的时候由操作系统分配，每次调用 connect 时客户端的端口会发生变化

- 假设在服务端调用 listen 和调用 accept 之间设了一个调试断点，暂停在此断点时，此时客户端调用 connect 后是否马上能连接成功？

客户端可以马上连接成功

- 连续快速 send 多次数据后，通过 Wireshark 抓包看到的发送的 Tcp Segment 次数

是否和 send 的次数完全一致？

并不完全一致，连续快速发送包会导致多个包合并发送，实际发送的数量小于 send 的数量

- 服务器在同一个端口接收多个客户端的数据，如何能区分数据包是属于哪个客户端的？

可以依靠客户端 IP 或者 FP 区分

- 客户端主动断开连接后，当时的 TCP 连接状态是什么？这个状态保持了多久？（可以使用 `netstat -an` 查看）

保持 CLOSE_WAIT 或者 FIN_WAIT_2 约 1min

- 客户端断网后异常退出，服务器的 TCP 连接状态有什么变化吗？服务器该如何检测连接是否继续有效？

不会立刻有变化，但是服务器可以每过一段时间向客户端发送数据包进行探测，观察其是否在设定的超时时间内进行回应。如果客户端没有在指定时间内进行回应，则服务端会主动断开 TCP 连接。

七、 讨论、心得

本次实验由我和队友分工完成，我负责客户端代码的撰写，客户端内容相对更丰富一些，但是一个写还是相对顺利的。通过查找相关参考资料并加以借鉴自己也逐渐掌握了一些 socket API 的功能以及使用方法。