

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: 陈希典

学 院: 计算机学院

系:

专 业: 计科

学 号: 3210102362

指导教师: 郑扣根

2023 年 10 月 24 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: 无 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

以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- 描述请求数据包的格式（画图说明），请求类型的定义

请求数据包格式



总共256Byte 定长，
对于非消息请求,data段=len=0
对于消息请求MESG_REQUEST
DATA段
data[0] = 目标client的序号
data[1]-data[254] = 发送信息

请求类型的定义：

```
#define TIME_REQUEST 1
#define NAME_REQUEST 2
#define LIST_REQUEST 3
#define MESG_REQUEST 4 // 发送消息请求
#define DISC_REQUEST 5 // 断连请求
```

分为五种, 分别是时间请求, 服务器名字请求, 服务器连接列表请求, 服务器发送信息请求, 服务器断开连接请求

- 描述响应数据包的格式（画图说明），响应类型的定义

响应数据包格式



相应类型

```
#define TIME_RESPONSE 6
#define NAME_RESPONSE 7
#define LIST_RESPONSE 8
#define MESG_RESPONSE 9 // 发送一条消息
#define DISC_RESPONSE 10
```

TIME 时间相应

NAME 服务器名称响应

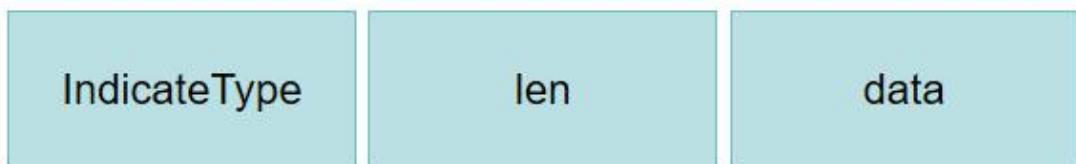
LIST 连接列表信息响应

MESG 转发消息响应

DISC 断连响应

- 描述指示数据包的格式（画图说明），指示类型的定义

指示数据包格式



总共256Byte 定长，
DATA段保存发送消息

只有一种，用于发送目标客户端转发消息

- 客户端初始运行后显示的菜单选项

```
PSU-LESSONS:team02@PSU-LESSONS:~$ ./client
You have not connect to any server
menu:
1. connect to server
2. exit
```

- 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）

```
while(ClientRunning){  
    // main线程检查消息队列是否有新的消息到达  
    if(msgIsArrived){  
        while(!messageQueue.empty()){  
            DecodeMsg(messageQueue.front());  
            messageQueue.pop();  
        }  
        msgIsArrived = false;  
    }  
    IOHandle();  
}
```

```
void IOHandle(){  
    // 获取用户输入,选择发送请求包  
    if(isConnect) ConnectMenu();  
    else disConnectMenu();  
}
```

```
void disConnectMenu(){
    cout << "You have not connect to any server\n" \
        "menu:\n" \
        "1. connect to server\n" \
        "2. exit\n";
    int choice;
    cin >> choice;
    switch(choice){
        case 1:{
            Createsocket();
            break;
        }
        case 2:{
            cout << "Bye\n";
            exit(0);
        }
        default:{
            cout << "wrong choice\n";
            break;
        }
    }
}
```

```
void ConnectMenu(){
    cout << "You have connected to server:\n" \
        "1. get time\n" \
        "2. get name\n" \
        "3. get list\n" \
        "4. send message\n" \
        "5. disconnect\n" \
        "6. exit\n";
    int choice;
    cin >> choice;
    char sendPacket[PACKETSIZE] = {0};
    switch(choice){
        case 1:{
            MakeRequestPacket(TIME_REQUEST,sendPacket);
            break;
        }
        case 2:{
            MakeRequestPacket(NAME_REQUEST,sendPacket);
            break;
        }
        case 3:{
            MakeRequestPacket(LIST_REQUEST,sendPacket);
            break;
        }
        case 4:{
            cout << "please input index of destination client\n";
            int destIndex;
            cin >> destIndex;
            sendPacket[2] = (char)destIndex;
            cout << "please input message\n";
            string message;
            cin >> message;
            memcpy(sendPacket + 3, message.c_str(), message.length());
            MakeRequestPacket(MESG_REQUEST,sendPacket);
        }
    }
}
```

```
        MakeRequestPacket(MESG_REQUEST,sendPacket);
        break;
    }
    case 5:{
        MakeRequestPacket(DISC_REQUEST,sendPacket);
        closesocket(nowSOCKETptr);
        // 加入通知逻辑
        break;
    }
    case 6:{
        if(isConnect){
            cout << "Now disconnect to server\n";
            MakeRequestPacket(DISC_REQUEST,sendPacket);
        }
        break;
    }
    default:{
        cout << "wrong choice\n";
        break;
    }
}
int handler = send(nowSOCKETptr,sendPacket,PACKETSIZE,0);
if (handler == SOCKET_ERROR) {
    cout << "send error\n";
}
```

主要可以概括为：

查看接受信息缓冲区，有则打印来自服务器的信息：

显示菜单

接受用户输入

非连接情况：创建对应请求包，发包

连接情况：建立连接，启用一个监听线程

- 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）

```
// 接收数据 发送给主线程
void RecvHandleThread(){
    char recvBuff[PACKETSIZE] = {0};
    while(ClientRunning){
        // 一次接受一个包
        int handler = recv(nowSOCKETptr,recvBuff,PACKETSIZE,0);
        if(handler > 0){
            msgIsArrived = true ;
            strcpy(msgRcvBuffer,recvBuff);
        }
    }
}
```

接受缓冲区一次接受一个包, 包是定长的 256byte

发送到全局 msg 寄存区,

设置 msg 到达

- 服务器初始运行后显示的界面

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
```

- 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）

```

    void Init(){
        WSADATA wsaData;      // 用于初始化 Winsock 库
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) { // 初始化 winsock 库
            std::cerr << "Failed to initialize Winsock." << std::endl;
            return;
        }
        serverSocket = socket(AF_INET, SOCK_STREAM, 0);
        serverAddress.sin_family = AF_INET;
        serverAddress.sin_addr.s_addr = INADDR_ANY;
        serverAddress.sin_port = htons(2362); // 指定监听的端口

        // 绑定套接字到指定地址和端口
        if (bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == SOCKET_ERROR) {
            std::cerr << "Failed to bind socket. Error code: " << WSAGetLastError() << std::endl;
            closesocket(serverSocket);
            WSACleanup();
            return ;
        }

        if (listen(serverSocket, SOMAXCONN) == SOCKET_ERROR) {
            std::cerr << "Failed to listen on socket. Error code: " << WSAGetLastError() << std::endl;
            closesocket(serverSocket);
            WSACleanup();
            return ;
        }
        cout << "Server start!" << endl ;
    }
}

```

```

int main(){
    std::signal(SIGINT, signalHandler); // 登记一个信号
    Init(); // 套接字初始化

    while(serverRunning){}

        SOCKET* clientSocket = new(SOCKET);
        struct sockaddr_in clientAddr;
        int clientAddrLength = sizeof(clientAddr);
        // 主线程循环调用 accept 直到返回一个有效的socket句柄
        *clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &clientAddrLength);
        // 获取客户端地址信息,记录下该客户端句柄和连接状态、端口。
        char* clientIP = inet_ntoa(clientAddr.sin_addr);
        int clientPort = ntohs(clientAddr.sin_port);
        int temidx = clients_map.size();
        clients_map[temidx] = ClientInfo(clientSocket, clientIP, clientPort);
        std::cout << "Client connected. IP: " << clientIP << ", Port: " << clientPort << std::endl;

        // 然后创建一个子线程后继续调用accept()
        thread clientThread(clientHandle,clientSocket,temidx);
        clientThread.detach();
    }
    cout << "finish ";
    // 关闭服务器套接字
    closesocket(serverSocket);
    WSACleanup();
}

```

绑定端口后,主线程循环调用 accept 函数,直到接收到一个连接信号

登记信息,保存在连接 MAP 中

开辟一个处理线程 ClientThread,用于运行子线程处理函数

继续监听

- 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）

```

/*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*/
void clientHandle(SOCKET* clientSocketptr, int clientIdx){
    char sendPacket[PACKETSIZE+10] = {0};
    char recvBuffer[PACKETSIZE+10] = {0};
    //发送一个hello
    strcpy(sendPacket, "hello");
    MakeResponsePacket(MESG_RESPONSE, sendPacket);
    send(*clientSocketptr, sendPacket, PACKETSIZE, 0);

    while(serverRunning){
        //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
        int recvFlag = recv(*clientSocketptr, recvBuffer, PACKETSIZE, 0);
        if(recvFlag > 0){
            char requestType = recvBuffer[0];
            // cout << "收到 " << recvBuffer << endl;
            switch (requestType) {
                case TIME_REQUEST: { // 获取时间
                    time_t now = time(0);
                    char *time = ctime(&now);
                    strcpy(sendPacket, time);
                    MakeResponsePacket(TIME_RESPONSE, sendPacket);
                    send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
                    break;
                }
                case NAME_REQUEST: {
                    char hostname[MAX_COMPUTERNAME_LENGTH + 1];
                    DWORD size = sizeof(hostname);
                    GetComputerNameA(hostname, &size);
                    strcpy(sendPacket, hostname);
                    MakeResponsePacket(NAME_RESPONSE, sendPacket);
                    send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
                    break;
                }
                case LIST_REQUEST: {
                    // 可能会遇到包长不够的情况，此时需要分多批发，目前不考虑这个问题
                    for (auto it: clients_map) {
                        int idx = it.first;
                        char idxstr[10];
                        sprintf(idxstr, "[%d]", idx);
                        strcat(sendPacket, idxstr);
                        strcat(sendPacket, it.second.IP.c_str()); // 连接IP字符串
                        char portstr[10];
                        sprintf(portstr, ":%d\n", it.second.port); // 将端口值转换为字符串
                        strcat(sendPacket, portstr);
                    }
                }
            }
        }
    }
}


```

```

        break;
    }
    case MESG_REQUEST: {
        int destIdx = recvBuffer[2];
        if (clients_map.find(destIdx) == clients_map.end()) {
            // 不存在 发送一个错误响应
            strcpy(sendPacket, "destination Client not exist!");
            MakeResponsePacket(MESG_RESPONSE, sendPacket);
        } else {
            SOCKET *destSocket = clients_map[destIdx].clientSocket;
            strcpy(sendPacket, "message send successfully!");
            MakeResponsePacket(MESG_RESPONSE, sendPacket);
            // 还要额外发送一个指示包
            char msgPack[256];
            memcpy(msgPack, recvBuffer + 2, (size_t)(recvBuffer[1]));
            MakeIndicatePacket(INDICATE, msgPack);
            send(*destSocket, sendPacket, PACKETSIZE, 0);
        }
        send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
        break;
    }
    case DISC_REQUEST: { // 收到断连请示 子线程结束
        int Del_idx = (int)recvBuffer[2];
        clients_map.erase(Del_idx);
        MakeResponsePacket(DISC_RESPONSE, sendPacket);
        send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
        goto final;
        break;
    }
}
memset(sendPacket, 0, PACKETSIZE+10);
}
}
final:
// 结束时释放
closesocket(*clientSocketptr);
clients_map.erase(clientIdx);
}

```

概括为：

客户端建立监听线程, 发送一个 hello

然后循环调用 receive 函数, 直至收到一个完整的包

拆包,

根据请求类型, 组装对应的相应包

发送相应包

- 客户端选择连接功能时, 客户端和服务端显示内容截图。

客户端显示：

```
You have not connect to any server
menu:
1. connect to server
2. exit
1
please input server IP
127.0.0.1
please input server port
2362
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

服务端显示：

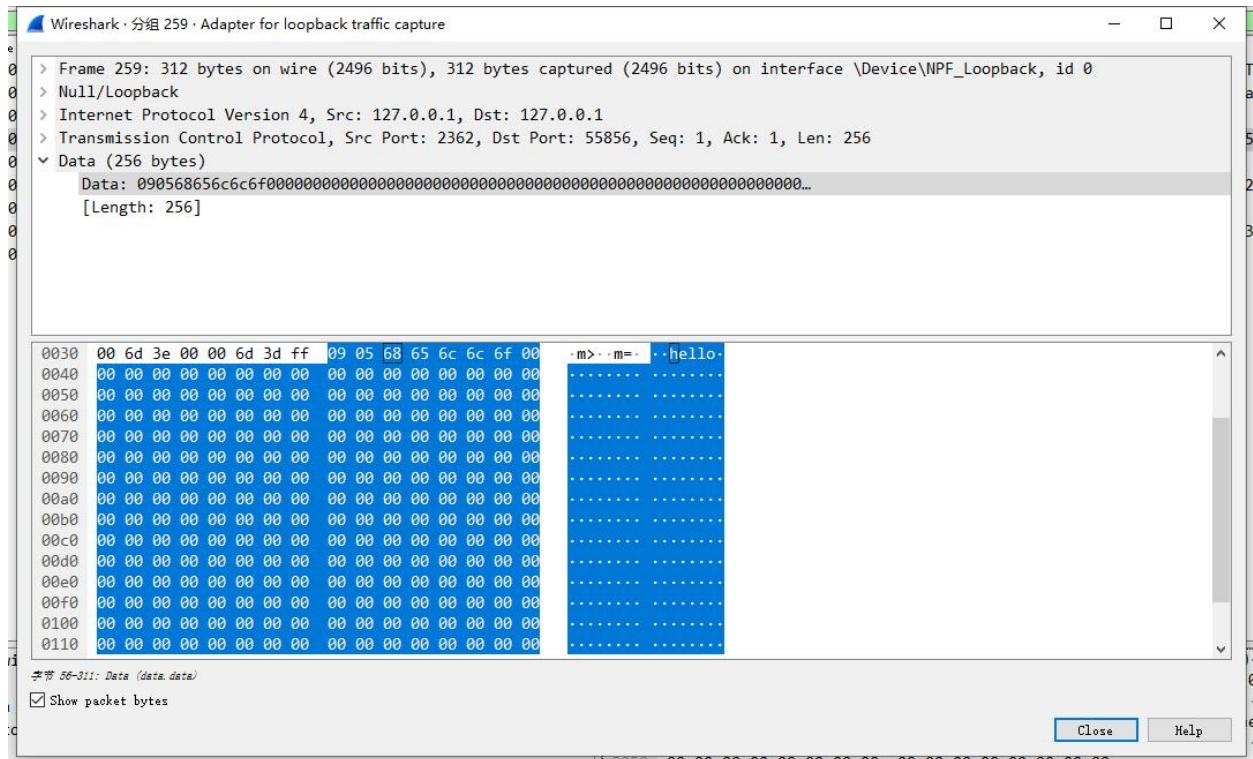
```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
Client connected. IP: 127.0.0.1, Port: 55856
```

Wireshark 抓取的数据包截图：

建立连接

No.	Time	Source	Destination	Protocol	Length	Info
256	55.699167	127.0.0.1	127.0.0.1	TCP	64	55856 → 2362 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=7159295 TSecr=0
257	55.699231	127.0.0.1	127.0.0.1	TCP	64	2362 → 55856 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=7159295 TSecr=7159296
258	55.699307	127.0.0.1	127.0.0.1	TCP	56	55856 → 2362 [ACK] Seq=1 Ack=0 Win=2619136 Len=0 TSval=7159295 TSecr=7159295
259	55.700147	127.0.0.1	127.0.0.1	TCP	312	2362 → 55856 [PSH, ACK] Seq=1 Ack=1 Win=2619136 Len=256 TSval=7159295 TSecr=7159295
260	55.700188	127.0.0.1	127.0.0.1	TCP	56	55856 → 2362 [ACK] Seq=1 Ack=257 Win=2618880 Len=0 TSval=7159296 TSecr=7159296
663	133.277078	127.0.0.1	127.0.0.1	TCP	312	55856 → 2362 [PSH, ACK] Seq=1 Ack=257 Win=2618880 Len=256 TSval=7236873 TSecr=7159296
664	133.277169	127.0.0.1	127.0.0.1	TCP	56	2362 → 55856 [ACK] Seq=257 Ack=257 Win=2618880 Len=0 TSval=7236873 TSecr=7236873
665	133.280467	127.0.0.1	127.0.0.1	TCP	312	2362 → 55856 [PSH, ACK] Seq=257 Ack=257 Win=2618880 Len=256 TSval=7236877 TSecr=7236873
666	133.280501	127.0.0.1	127.0.0.1	TCP	56	55856 → 2362 [ACK] Seq=257 Ack=513 Win=2618624 Len=0 TSval=7236877 TSecr=7236877

同时服务器给客户端发了一个 hello



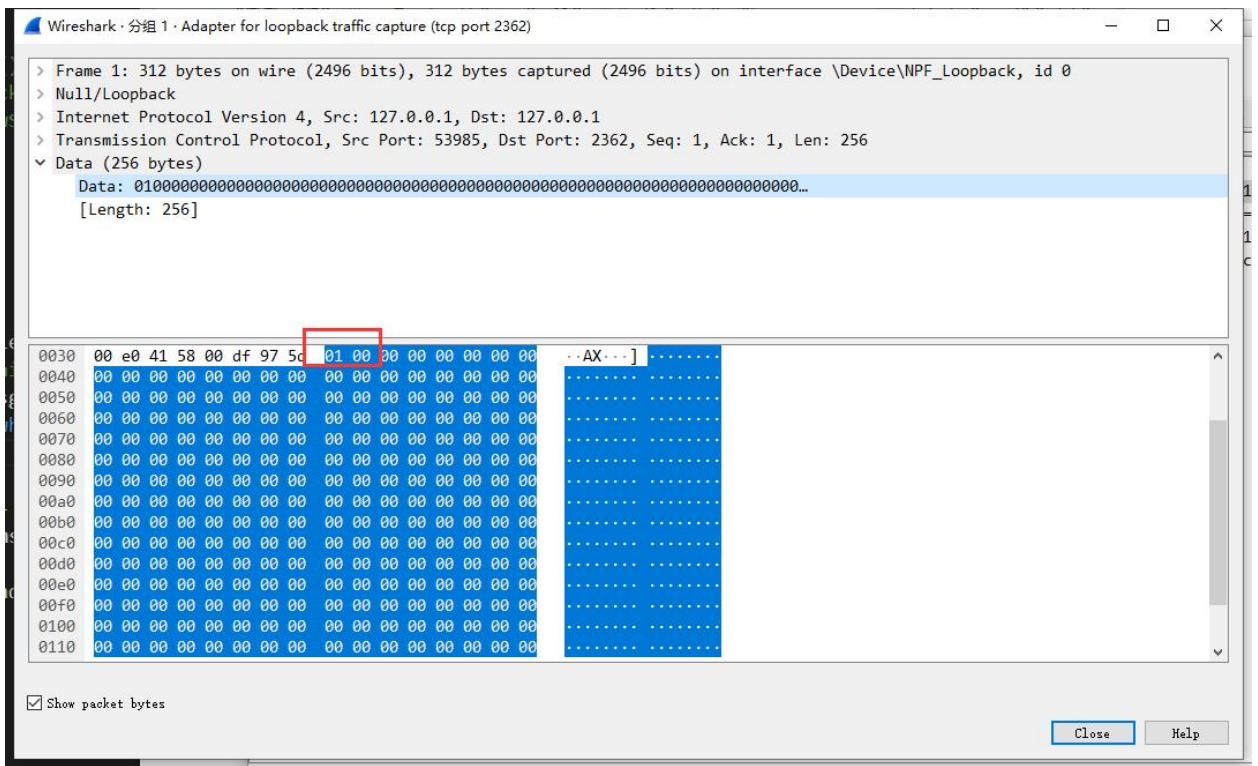
- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

```
1
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
1
[Server]
Time is Tue Oct 24 12:26:06 2023

You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
Server start!
Client connected. IP: 127.0.0.1, Port: 53985
Time Request from Client0
```

客户端发送一个时间请求

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位置）：



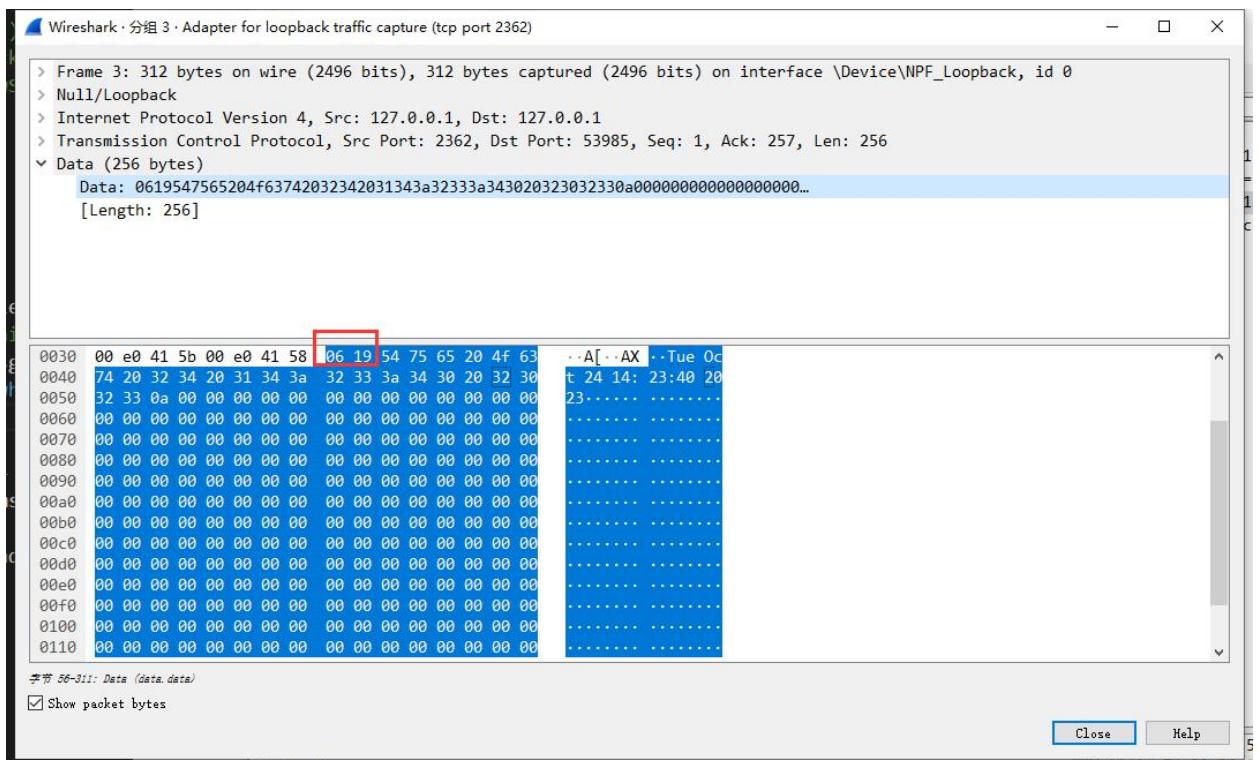
客户端发送请求包，第一个字段表示请求类型

1 代表时间请求

00 代表数据段长度

后面都是 0

随即服务端发送一个响应包：



06 代表包类型为相应包, 响应类型为时间相应

19 是 len, 代表 data 段信息长度

后续是发送的时间内容

相应的服务器处理代码片段:

```

switch (requestType) {
    case TIME_REQUEST: { // 获取时间
        time_t now = time(0);
        char *time = ctime(&now);
        strcpy(sendPacket, time);
        MakeResponsePacket(TIME_RESPONSE, sendPacket);
        send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
        break;
    }
}

```

- 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。

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
Client connected. IP: 127.0.0.1, Port: 53985
Time Request from Client0
Name Request from Client0
□
```

You have connected to server:

1. get time
 2. get name
 3. get list
 4. send message
 5. disconnect
 6. exit

wrong choice

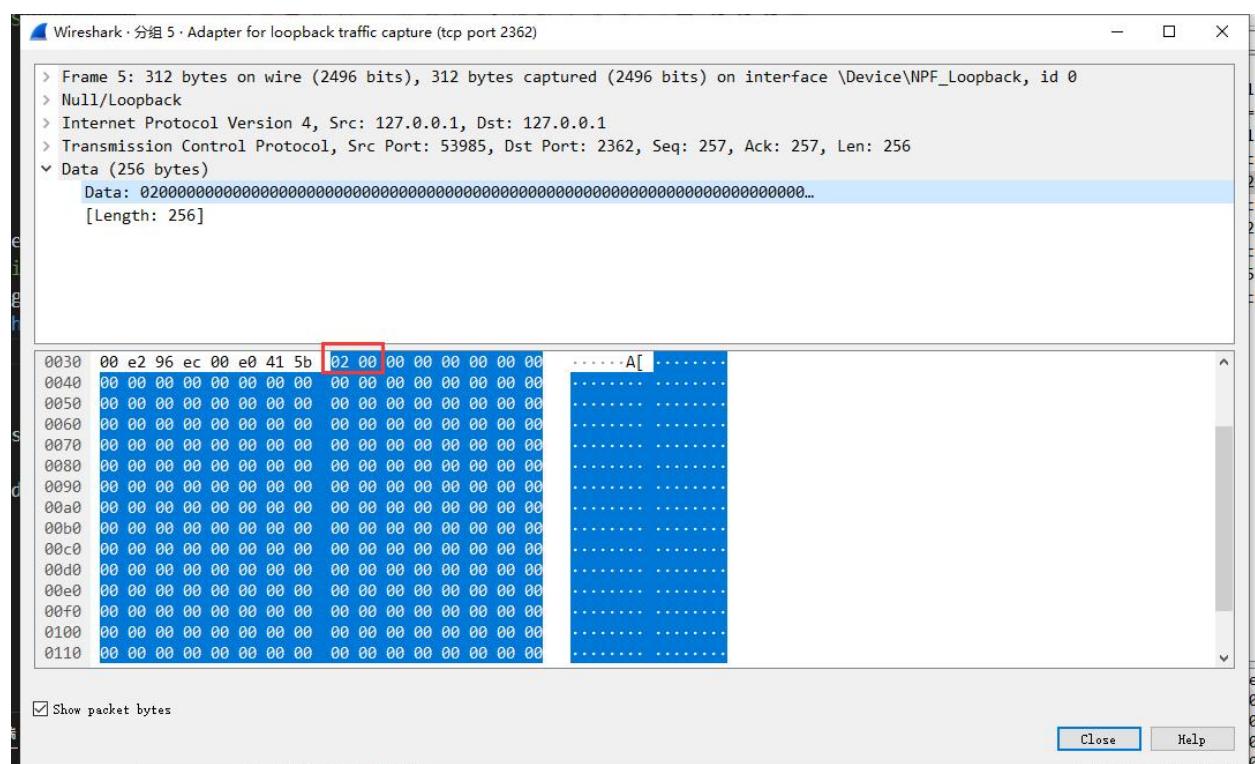
Wrong Client
[Server]

[Server] Name is LAPTOP-3DM1P3RI

You have connected to server:

1. get time
 2. get name
 3. get list
 4. send message
 5. disconnect
 6. exit

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）：



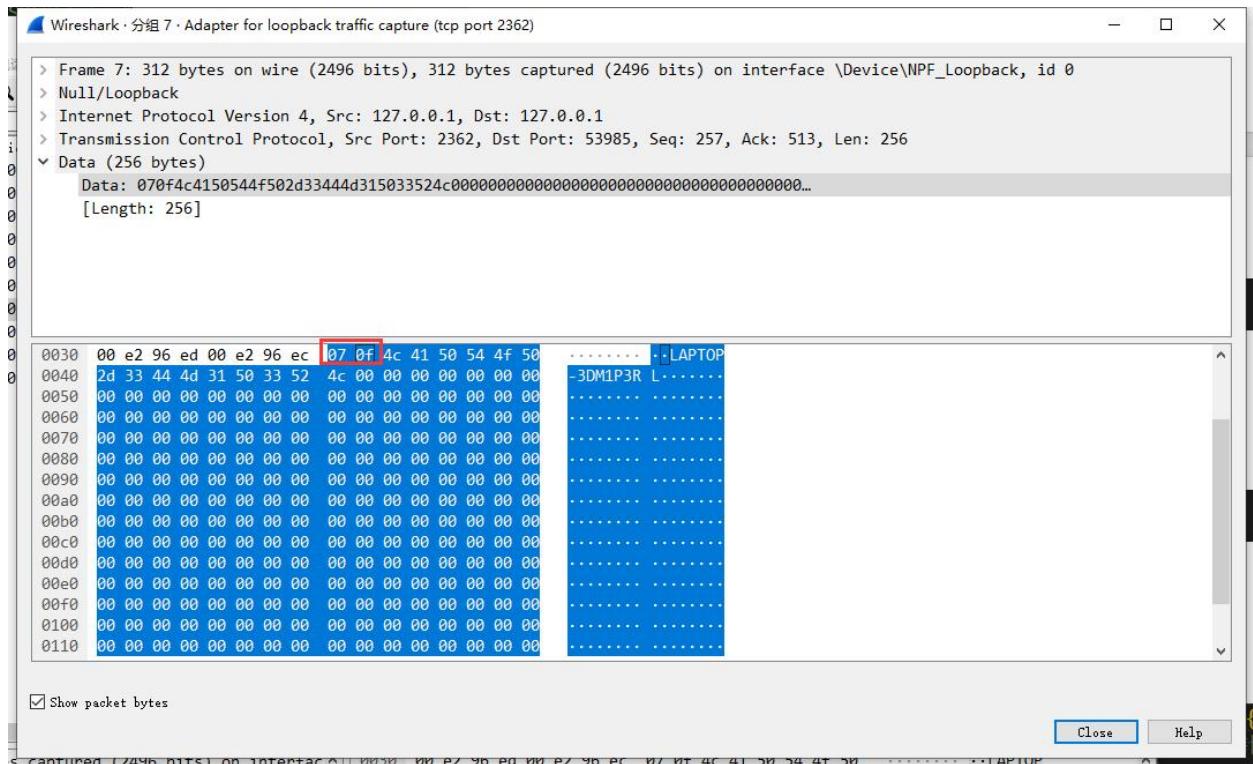
客户端发送一个请求类型为 2, 名称请求包, 长度为 0

```
#define NAME_REQUEST 2
```

服务器收到内容后回一个响应包

```
#define NAME_RESPONSE 7
```

07 表示相应类型为名称响应, len 表示处理长度, 后续 data 段是主机名



相关的服务器的处理代码片段:

```
void MakeResponsePacket(char HeadType,char* buffer){
    //输入一个包的类型,组装出一个响应包,并通过buffer指针来显式返回;
    // 对于MSG请求,需要事先将内容放在buffer之中
    int Len = strlen(buffer);
    memmove(buffer+2,buffer,Len);
    buffer[1] = (char)len;
    buffer[0] = HeadType;
}
```

```
case NAME_REQUEST: {
    char hostname[MAX_COMPUTERNAME_LENGTH + 1];
    DWORD size = sizeof(hostname);
    GetComputerNameA(hostname, &size);
    strcpy(sendPacket, hostname);
    MakeResponsePacket(NAME_RESPONSE, sendPacket);
    send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
    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。

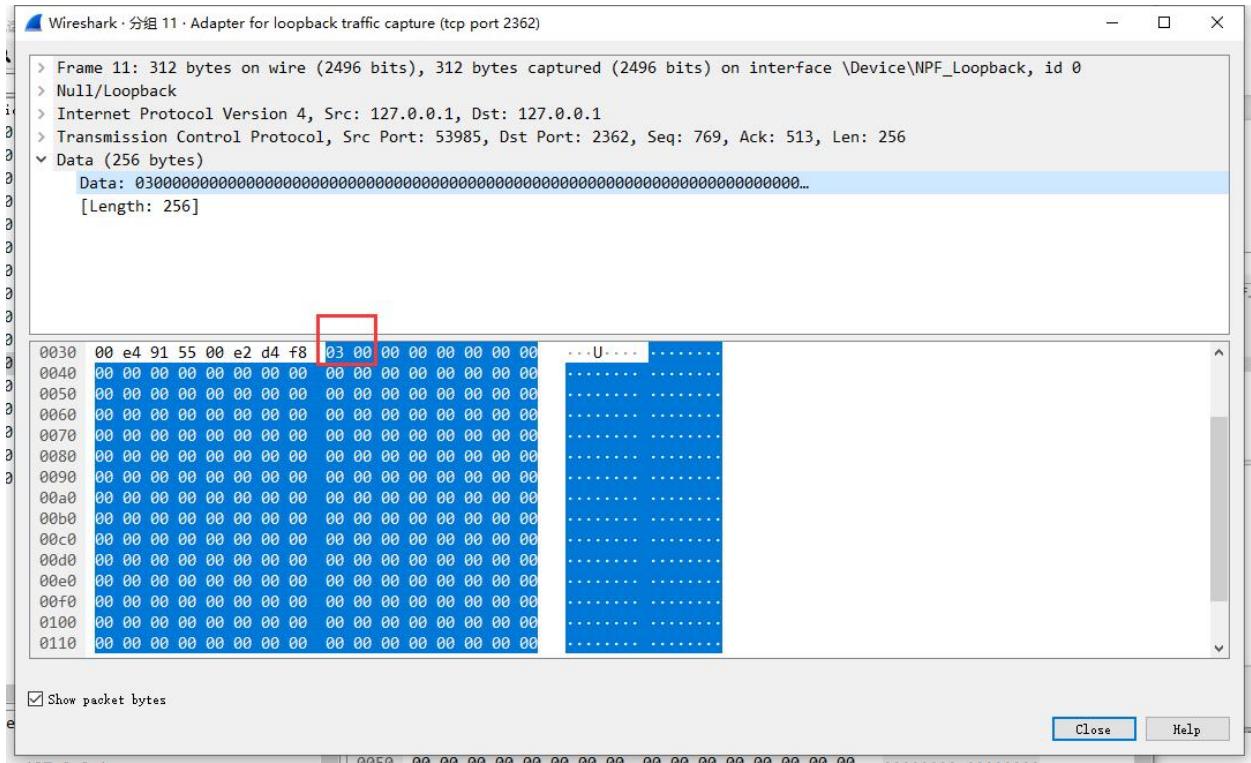
```
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
0
wrong choice
[Server]
Client list is [0]127.0.0.1:53985

You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
Client connected. IP: 127.0.0.1, Port: 53985
Time Request from Client0
Name Request from Client0
List Request from Client0
```

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

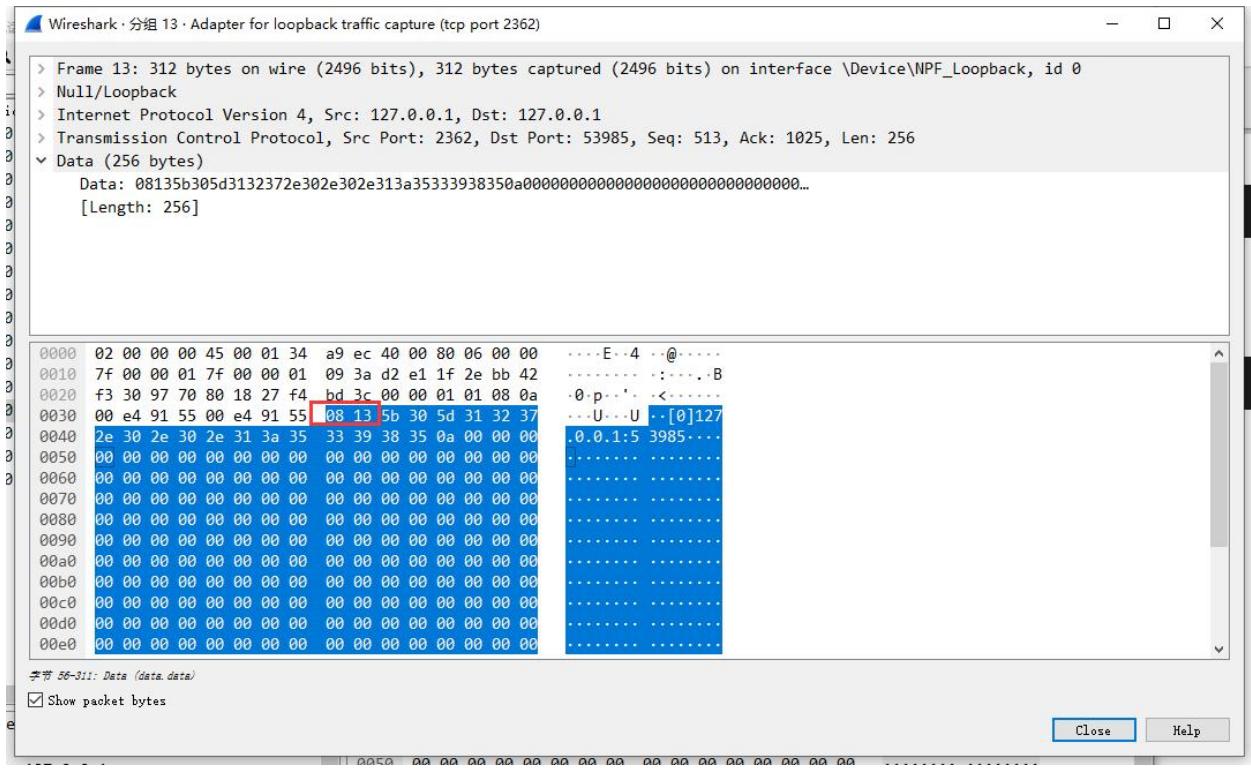
客户端列表数据对应的位置)：



请求包, 请求类型是 03, 列表请求, len=data=0

```
#define LIST_REQUEST 3
```

响应包: 相应类型是 08, 数据长度 len = 13, 数据内容则是服务器 LIST 信息



```
#define LIST_RESPONSE 8
```

相关的服务器的处理代码片段：

```
case LIST_REQUEST: {
    // 可能会遇到包长不够的情况,此时需要分多批发,目前不考虑这个问题
    for (auto it: clients_map) {
        int idx = it.first;
        char idxstr[10];
        sprintf(idxstr, "[%d]", idx);
        strcat(sendPacket, idxstr);
        strcat(sendPacket, it.second.IP.c_str()); // 连接IP字符串
        char portstr[10];
        sprintf(portstr, ":%d\n", it.second.port); // 将端口值转换为字符串
        strcat(sendPacket, portstr);
    }
    MakeResponsePacket(LIST_RESPONSE, sendPacket);
    send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
    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：

```
You have connected to server:  
1. get time  
2. get name  
3. get list  
4. send message  
5. disconnect  
6. exit  
4  
please input index of destination client  
0  
please input message  
Ni hao
```

收到来自服务器的信息

```
You have connected to server:  
1. get time  
2. get name  
3. get list  
4. send message  
5. disconnect  
6. exit  
0  
wrong choice  
[Server]  
message send successfully!  
You have connected to server:  
1. get time  
2. get name  
3. get list  
4. send message  
5. disconnect  
6. exit  
|
```

服务器：

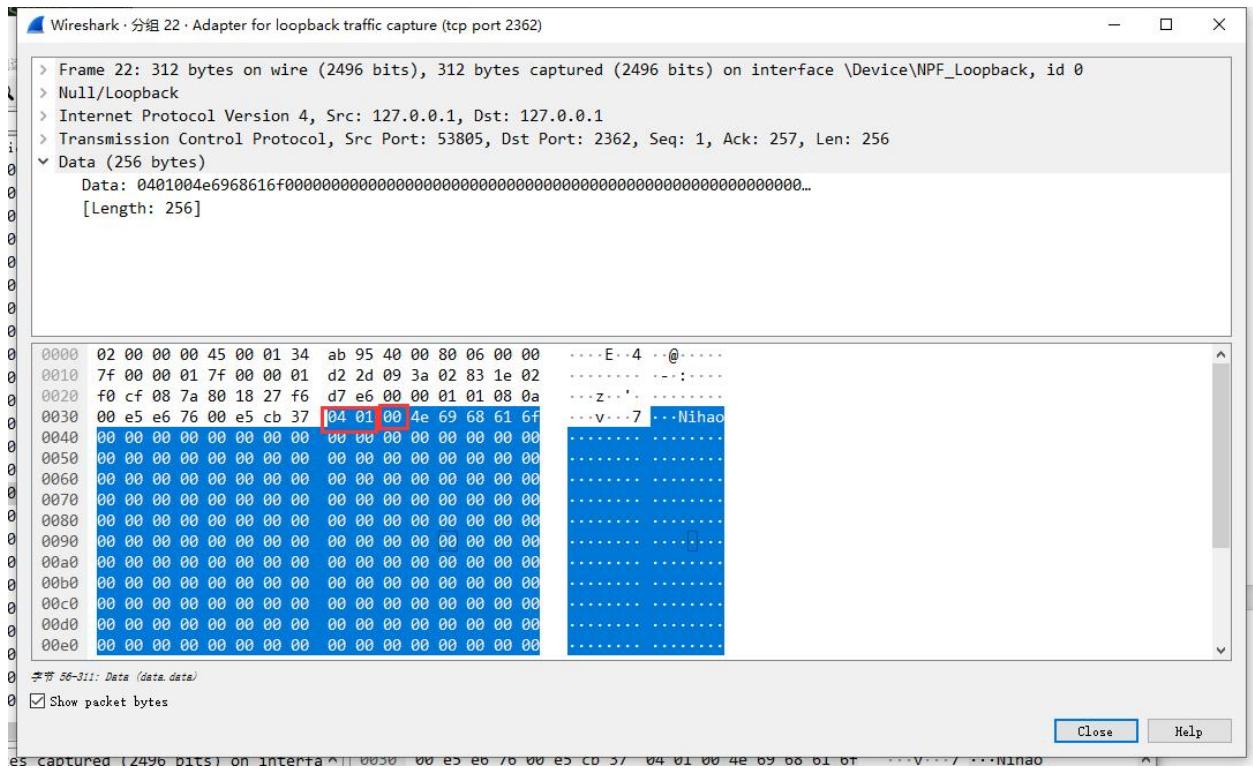
```
SERVER STARTED  
Client connected. IP: 127.0.0.1, Port: 53985  
Time Request from Client0  
Name Request from Client0  
List Request from Client0  
Client connected. IP: 127.0.0.1, Port: 53805  
Message Request from Client1  
|
```

接收消息的客户端：

```
wrong choice
[Server]
Receive Nihao
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送的消息请求包



第一个 char 表示类型为 MESG_REQUEST 这是一个发送信息请求包

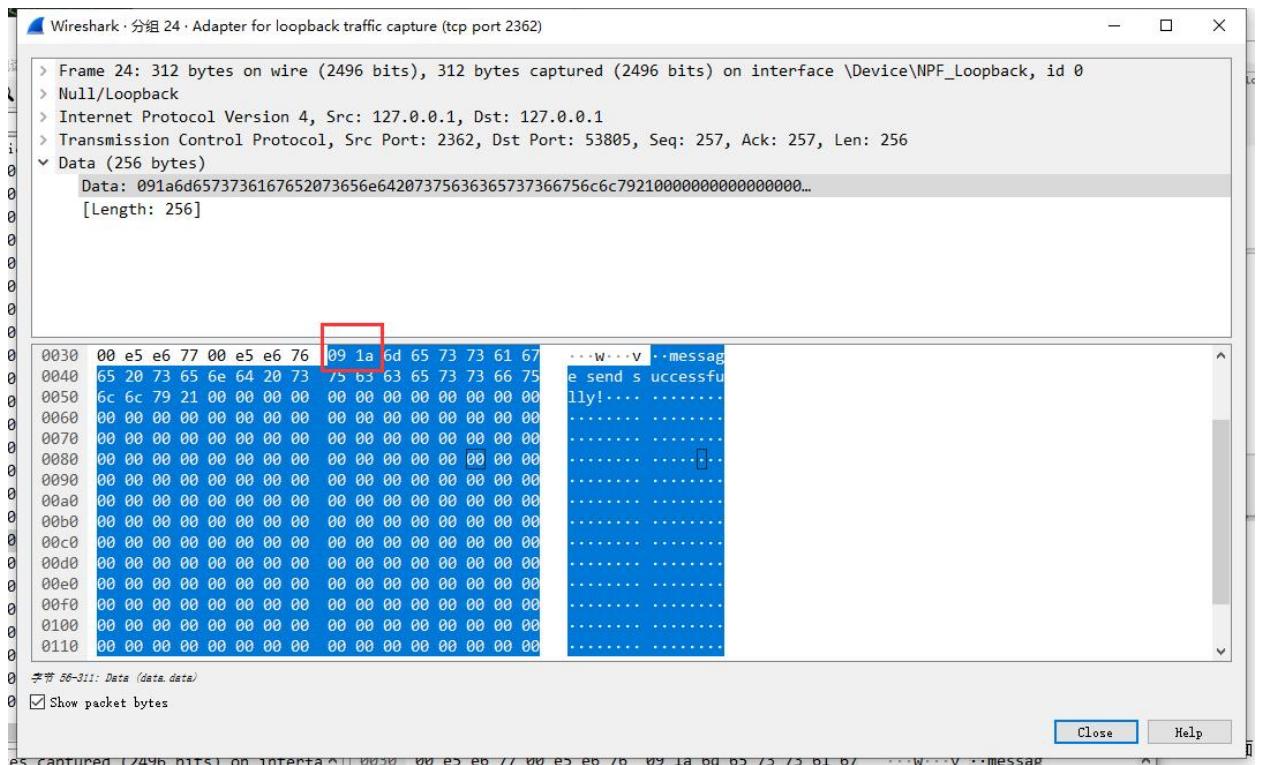
第二个 char 1 表示这个消息有一个包(考虑到消息可能很长, 需要多个包来实现,
但是本 lab 只实现了消息小于包长的情况)

第三个 char 表示发送给的目标 Client 的序号

后续的 data 段表示信息, 可以看到发了一个 Nihao

相应的 服务器会查找对应的序号, 如果找到给目标 Client 发送一个指示包, 里面包含转发信息, 并且给源 Client 发送一个响应包, 这个包用于通知发送状态

给 源 Client 的响应包如下;



第一个 char 09 代表发包类型 是 MESG_RESPONSE 相应

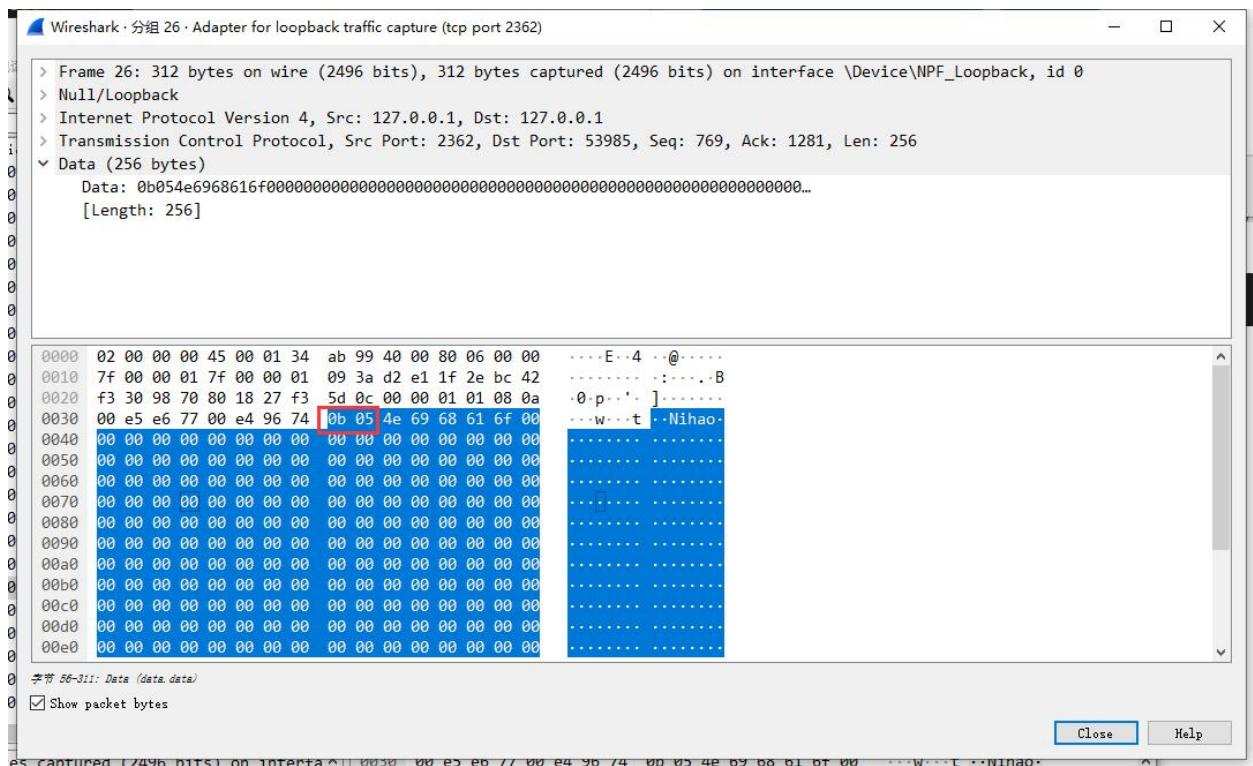
```
#define MESG_RESPONSE 9 // 发送一条消息
```

后面的 1a 表示长度

后续 data 段是响应源 Client 的消息 , 表示成功发送

给 目标 Client 的包如下:

```
#define DISC_RESPONSE 11
```



第一个 char 表示类型为 INDICATE 指示包

第二个 char 表示长度

后续 char 是发送内容, 可以看到服务器确实转发了源 Client 想要发送的消息

相关的服务器的处理代码片段:

```
        }
        case MESG_REQUEST: {
            cout << "Message Request from Client" << clientIdx << endl;
            int destIdx = recvBuffer[2];
            bool destNotValid = (clients_map.find(destIdx) == clients_map.end());
            if (destNotValid) {
                // 不存在 发送一个错误响应
                strcpy(sendPacket, "destination Client not exist!");
                MakeResponsePacket(MESG_RESPONSE, sendPacket);
                send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
            } else {
                strcpy(sendPacket, "message send successfully!");
                MakeResponsePacket(MESG_RESPONSE, sendPacket);

                send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
                // 还要额外发送一个指示包
            }
            if(!destNotValid){
                SOCKET *destSocket = clients_map[destIdx].clientSocket;
                char msgPack[256];
                memcpy(msgPack, recvBuffer + 3, PACKETSIZE);
                MakeIndicatePacket(INDICATE, msgPack);
                send(*destSocket, msgPack, PACKETSIZE, 0);
            }
            break;
        }
    }
```

相关的客户端（发送和接收消息）处理代码片段：

发送请求包：

```
}

case 4:{  
    cout << "please input index of destination client\n";  
    int destIndex;  
    cin >> destIndex;  
    sendPacket[2] = (char)destIndex;  
    cout << "please input message\n";  
    string message;  
    cin >> message;  
    memcpy(sendPacket+3, message.c_str(), message.length());  
    MakeRequestPacket(MESG_REQUEST,sendPacket);  
    cout << sendPacket+3;  
    break;  
}
```

客户端 接受 Mesg 响应包

```
void Decodemsg(char* msg){
    char type = msg[0];
    char len = msg[1];
    char* data = msg+2;
    cout << "[Server]\n";
    switch(type){
        case TIME_RESPONSE:{
            cout << "Time is " << data << endl;
            break;
        }
        case NAME_RESPONSE:{
            cout << "Name is " << data << endl;
            break;
        }
        case LIST_RESPONSE:{
            cout << "Client list is " << data << endl;
            break;
        }
        case MESG_RESPONSE:{
            cout << data << endl;
            break;
        }
        case DISC_RESPONSE:{
            cout << "Disconnect success\n";
            isConnect = false;
            break;
        }
        case INDICATE:{
            cout << "Receive " << data << endl;
            break;
        }
        default:{
            cout << "wrong type\n";
            break;
        }
    }
}
```

客户端接受 Mesg 指示包

```
case INDICATE:{
    cout << "Receive " << data << endl;
    break;
}
```

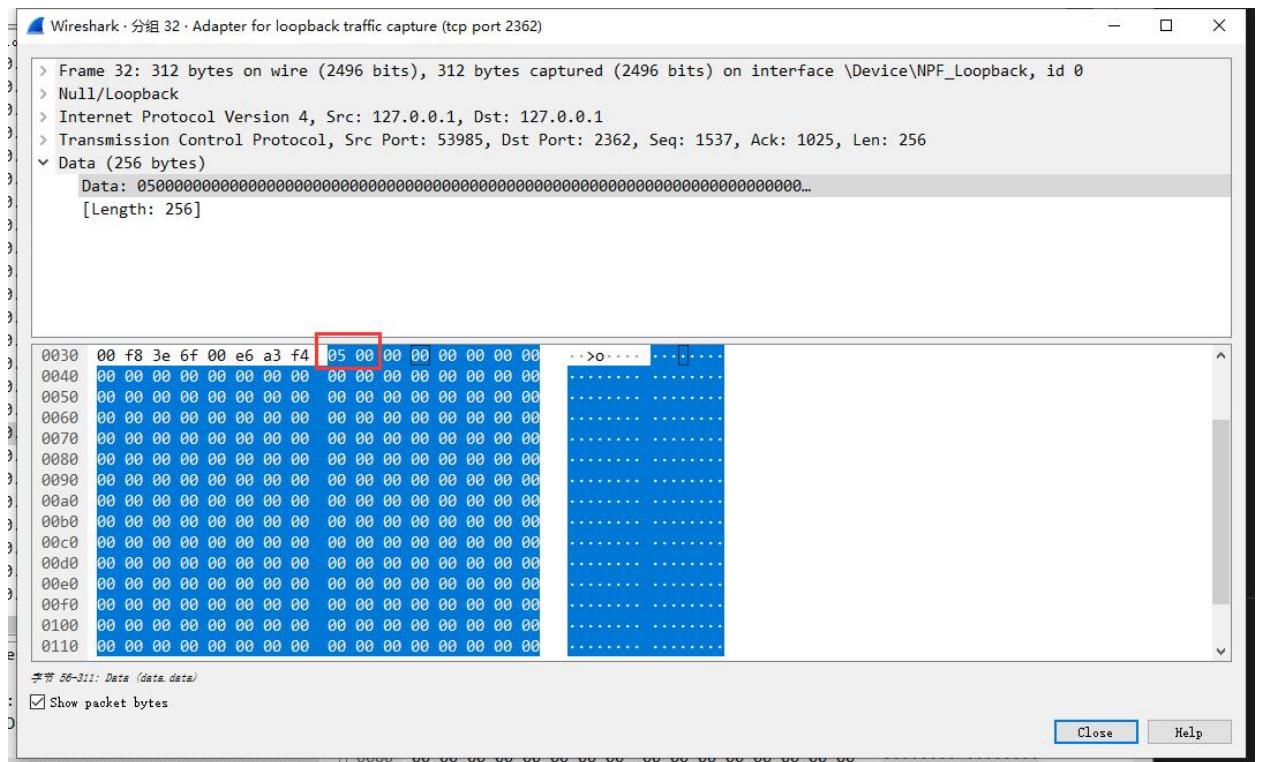
- 拔掉客户端的网线，然后退出客户端程序。观察客户端的 TCP 连接状态，并使用 Wireshark 观察客户端是否发出了 TCP 连接释放的消息。同时观察服务端的 TCP 连

接状态在较长时间内（10分钟以上）是否发生变化。

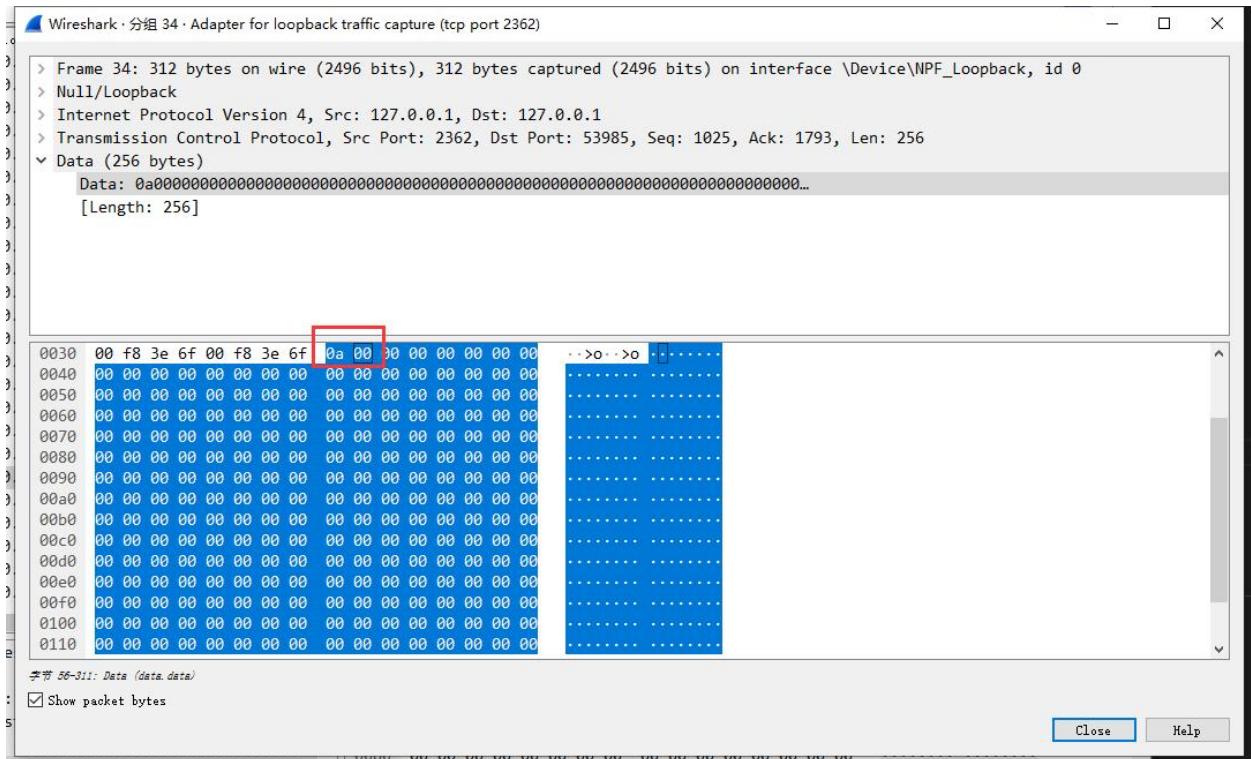
先模拟客户端正常断开，为此我自己添加了一组断连相应和请求包

```
send error
[Server]
Disconnect success
You have not connect to any server
menu:
1. connect to server
2. exit
2
Bye
```

此时发送了一个 DISC_REQUEST 信号，客户端告诉服务器删除自己



服务器收到 DISC 请求，断开连接，删除对应 Client 记录，同时回应一个 DISC_RESPONSE 响应包



客户端收到响应包之后关闭套接字, 此时 TCP 连接断开了

```
命令提示符 - NETSTAT -a
TCP      10.162.117.167:57735    ecs-120-46-45-120:https  CLOSE_WAIT
C
C:\Users\Lenovo>NETSTAT -a

活动连接

协议 本地地址        外部地址        状态
TCP    0.0.0.0:135      LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:445      LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:902      LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:912      LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:2362     LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:3306     LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:5040     LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:5091     LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:7890     LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:10001    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:30307    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:33060    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49664    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49665    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49667    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49668    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49671    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49687    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:49706    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:54740    LAPTOP-3DM1P3RL:0      LISTENING
TCP    0.0.0.0:61242    LAPTOP-3DM1P3RL:0      LISTENING
TCP    10.162.117.167:139  LAPTOP-3DM1P3RL:0      LISTENING
TCP    10.162.117.167:52070  180.163.235.169:http  ESTABLISHED
```

相应的客户端发出了 TCP 连接释放的消息

35	1572.119720	127.0.0.1	127.0.0.1	TCP	56	[53985 → 2362 [ACK] Seq=1793 Ack=1281 Win=10225 Len=0 TStamp=162689112 TSectr=16268911]
36	1572.119750	127.0.0.1	127.0.0.1	TCP	56	[2362 → 53985 [FIN, ACK] Seq=1281 Ack=1793 Win=10225 Len=0 TStamp=16268912 TSectr=16268912]
37	1572.119773	127.0.0.1	127.0.0.1	TCP	56	[53985 → 2362 [ACK] Seq=1793 Ack=1282 Win=10225 Len=0 TStamp=16268912 TSectr=16268912]
38	1581.454345	127.0.0.1	127.0.0.1	TCP	56	[53985 → 2362 [FIN, ACK] Seq=1793 Ack=1282 Win=10225 Len=0 TStamp=16278246 TSectr=16268912]
39	1581.454441	127.0.0.1	127.0.0.1	TCP	56	[2362 → 53985 [ACK] Seq=1282 Ack=1794 Win=10225 Len=0 TStamp=16278246 TSectr=16278246]

由于连接已经断开, 所以较长时间后仍然不会 TCP 连接

意外中断的 CASE

使用 CtrlC 杀掉 client 进程

```
2. exit
1
please input server IP
127.0.0.1
please input server port
2362
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
wrong choice
PS F:\lessons\ComputerNetwork\lab\lab1> 
```

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
Client connected. IP: 127.0.0.1, Port: 60795
Client connected. IP: 127.0.0.1, Port: 60796
[]
```

这个时候 server 是不能立刻知道有客户端断开连接的, 所以 list 还会残余已经结束的 client 信息

根据抓包显示，异常断开的 client 向服务器发出连接释放的消息：

断开连接的 client 是 60795 端口

可以看到在中断后，TCP 连接释放了，只是 list 列表还没更新

```
0
wrong choice
[Server]
Client list is hello[0]127.0.0.1:60795
[1]127.0.0.1:60796

You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

- 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？

重新连接

```
wrong choice
PS F:\lessons\ComputerNetwork\lab\lab1> .\c.exe
You have not connect to any server
menu:
1. connect to server
2. exit
1
please input server IP
127.0.0.1
please input server port
2362
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

```
0
wrong choice
[Server]
Client list is hello[0]127.0.0.1:60795
[1]127.0.0.1:60796
[2]127.0.0.1:61028

You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

发现之前本应该结束的 60795 端口还在

尝试发送消息能够成功,

```
NihaoYou have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
0
wrong choice
[Server]
message send successfully!
You have connected to server:
1. get time
2. get name
3. get list
4. send message
5. disconnect
6. exit
```

- 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。

```
cout << "[Server] " << endl;
switch(type){
    case TIME_RESPONSE:{
        cout << "Time is " << data << endl;
        cout << "timecnt = " << timecnt++ << endl;
        break;
    }
}
```

```
switch (requestType) {
    case TIME_REQUEST: { // 获取时间
        cout << "Time Request from Client" << clientIdx << endl;
        time_t now = time(0);
        char *time = ctime(&now);
        strcpy(sendPacket, time);
        MakeResponsePacket(TIME_RESPONSE, sendPacket);

        for(int i = 0 ; i < 100 ; i++) // for test
            send(*clientSocketptr, sendPacket, PACKETSIZE, 0);
        break;
    }
    case NAME_REQUEST: {
```

```
wrong choice
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 0
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 1
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 2
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 3
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 4
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 5
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 6
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 7
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 8
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 9
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 10
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 11
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 12
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 13
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 14
[Server]
Time is Tue Oct 24 15:27:47 2023
```

```
Time is Tue Oct 24 15:27:47 2023

timecnt = 87
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 88
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 89
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 90
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 91
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 92
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 93
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 94
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 95
[Server]
Time is Tue Oct 24 15:27:47 2023

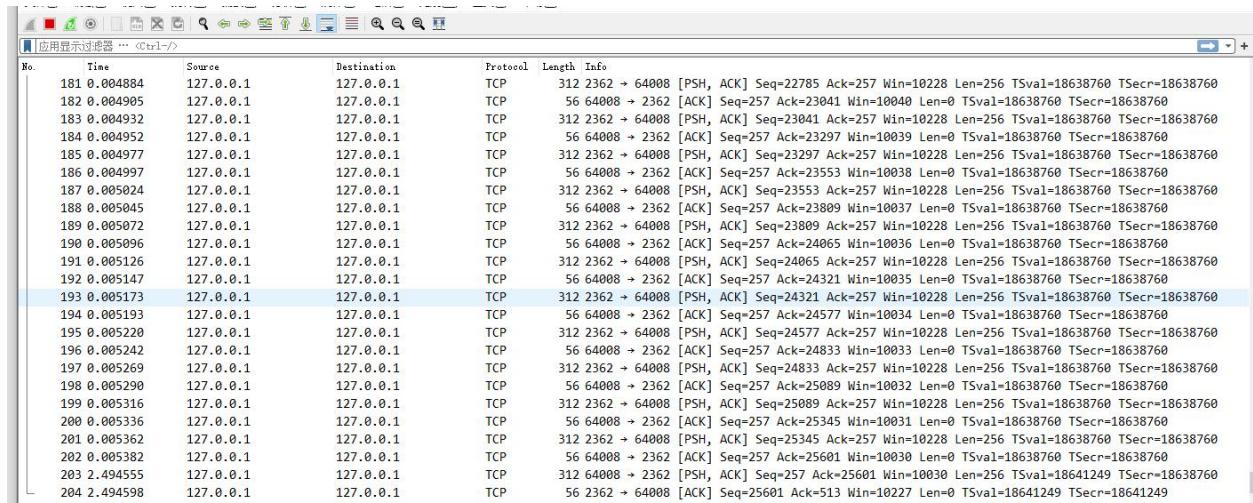
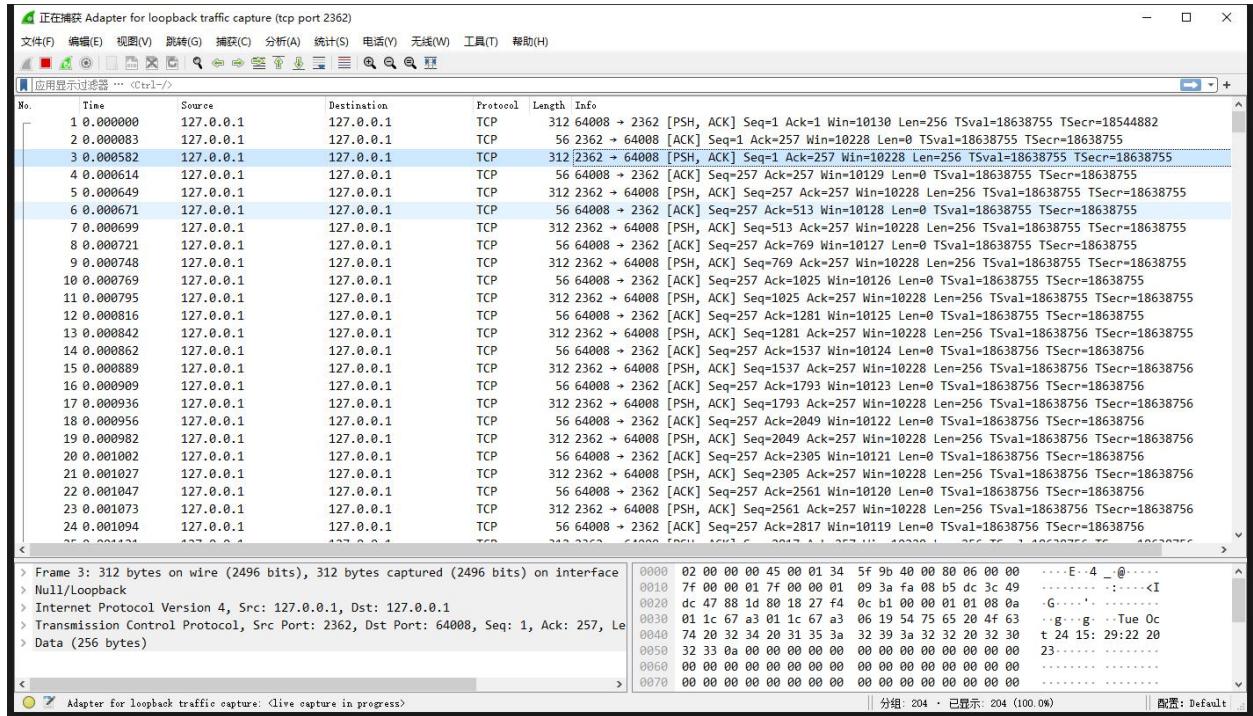
timecnt = 96
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 97
[Server]
Time is Tue Oct 24 15:27:47 2023

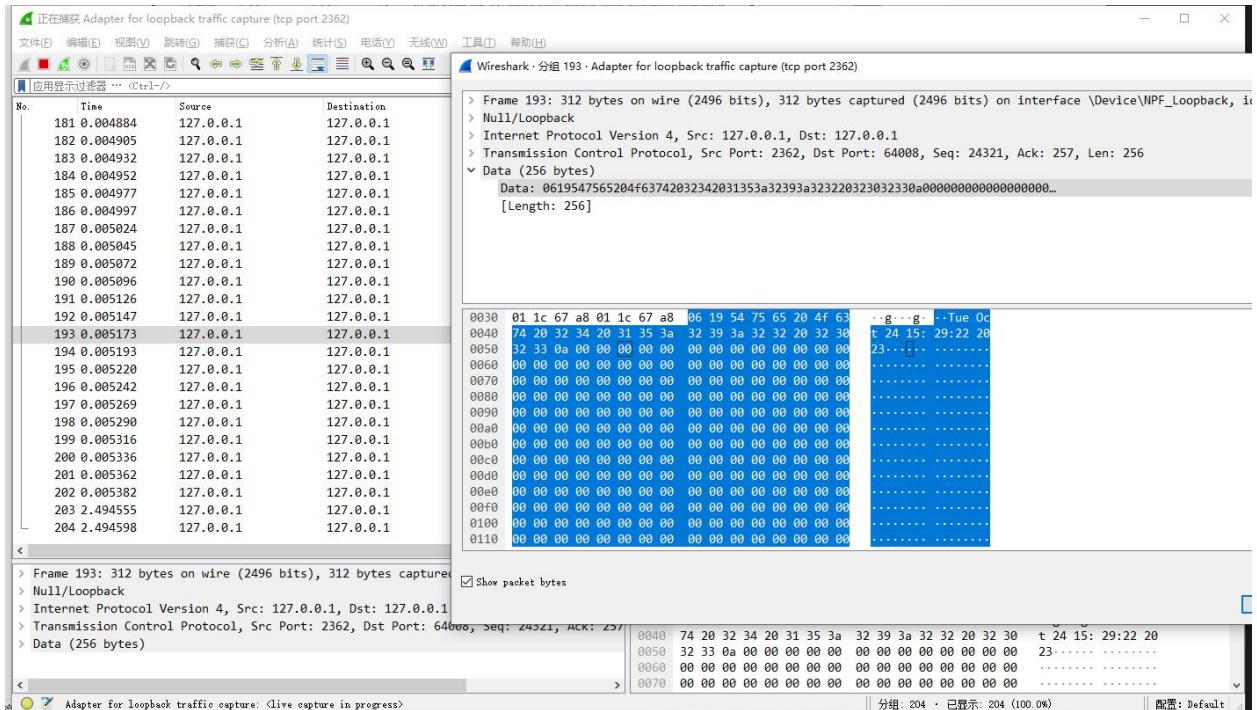
timecnt = 98
[Server]
Time is Tue Oct 24 15:27:47 2023

timecnt = 99
```

下面是 Wireshark 的截图，是从连接成功后还未选择 time 请求开始



通过最左侧的包计数可以看到，一次请求对应了两个包，其中 2362->64008 的 PSH 内容就是时间



由于只运行了一次请求,wireshark 抓到了 200 个包,这表示实际的的确完成了 100 个包的发送,追踪 cnt 信息也可以看到客户端收到了 100 个包,这表明消息队列是有效的,的确起到了消息缓冲的作用

- 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

由于做到同时发送比较困难, 需要借助线程的力量

```

}
cout << "timecnt = " << timecnt++ << endl;
for(int i = 0 ; i < 100 ; i++) // for test
    int handler = send(nowSOCKETptr,sendPacket,PACKETSIZE,0);
}

```

```
}

void test2(){
    //连接服务器
    struct sockaddr_in serverAddress;
    SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1");
    serverAddress.sin_port = htons(2362);
    connect(clientSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress));
    char sendPacket[PACKETSIZE] = {0};
    MakeRequestPacket(TIME_REQUEST,sendPacket);
    for(int i = 0 ; i < 100 ;i++){
        send(clientSocket,sendPacket,PACKETSIZE,0);
        cout << "[2]send time packet" << i << endl;
    }
}

int main(){

    Init();

    thread t1(test1);
    thread t2(test2);
    t1.join();
    t2.join();

    return 0;
}
```

写一个测试程序, 内容就是连接服务器, 并发送 100 个时间请求包. 让他们同时运行.

客户端部分, 可以看到两个线程交替执行, 轮流发包, 直到各自发完一百个包

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\c.exe
[1]send time packet[2]send time packet00

[1]send time packet1
[2]send time packet1
[1]send time packet2
[2]send time packet2
[1]send time packet3
[2]send time packet3
[1]send time packet4
[2]send time packet4
[1]send time packet5
[2]send time packet5
[1]send time packet6
[2]send time packet6
[1]send time packet7
[2]send time packet7
[1]send time packet8
[2]send time packet8
[1]send time packet9
[2]send time packet9
[1]send time packet10
[2]send time packet10
[1]send time packet11
[2]send time packet11
```

```
PS F:\lessons\ComputerNetwork\lab\lab1> .\s.exe
Server start!
Client connected. IP: 127.0.0.1, Port: 60123
Client connected. IP: Time Request from Client0
127.0.0.1, Port: 60122
Time Request from Client1
Time Request from ClientTime Request from Client0
1
Time Request from Client0
Time Request from Client1
```

客户端依次进行响应

No.	Time	Source	Destination	Protocol	Length	Info
775	0.067151	127.0.0.1	127.0.0.1	TCP	312	60158 → 2362 [PSH, ACK] Seq=24833 Ack=25089 Win=2594048 Len=256 TStamp=19762497 TSect=19762497
776	0.067173	127.0.0.1	127.0.0.1	TCP	312	2362 → 60159 [PSH, ACK] Seq=25089 Ack=25089 Win=2594304 Len=256 TStamp=19762497 TSect=19762497
777	0.067205	127.0.0.1	127.0.0.1	TCP	56	60159 → 2362 [ACK] Seq=25089 Ack=25345 Win=2593792 Len=0 TStamp=19762497 TSect=19762497
778	0.067219	127.0.0.1	127.0.0.1	TCP	56	2362 → 60158 [ACK] Seq=25089 Ack=25089 Win=2594304 Len=0 TStamp=19762497 TSect=19762497
779	0.067490	127.0.0.1	127.0.0.1	TCP	312	60159 → 2362 [PSH, ACK] Seq=25089 Ack=25089 Win=2593792 Len=0 TStamp=19762497 TSect=19762497
780	0.067497	127.0.0.1	127.0.0.1	TCP	312	2362 → 60158 [PSH, ACK] Seq=25089 Ack=25089 Win=2594304 Len=256 TStamp=19762497 TSect=19762497
781	0.067532	127.0.0.1	127.0.0.1	TCP	56	60159 → 2362 [ACK] Seq=25089 Ack=25345 Win=2593792 Len=0 TStamp=19762497 TSect=19762497
782	0.067544	127.0.0.1	127.0.0.1	TCP	56	2362 → 60159 [ACK] Seq=25345 Ack=25345 Win=2594048 Len=0 TStamp=19762498 TSect=19762498
783	0.067849	127.0.0.1	127.0.0.1	TCP	312	60158 → 2362 [PSH, ACK] Seq=25089 Ack=25345 Win=2593792 Len=256 TStamp=19762498 TSect=19762498
784	0.067865	127.0.0.1	127.0.0.1	TCP	312	2362 → 60159 [PSH, ACK] Seq=25345 Ack=25089 Win=2594048 Len=256 TStamp=19762498 TSect=19762498
785	0.067907	127.0.0.1	127.0.0.1	TCP	56	60159 → 2362 [ACK] Seq=25345 Ack=25601 Win=2593536 Len=0 TStamp=19762498 TSect=19762498
786	0.067923	127.0.0.1	127.0.0.1	TCP	56	2362 → 60158 [ACK] Seq=25345 Ack=25345 Win=2594048 Len=0 TStamp=19762498 TSect=19762498
787	0.068189	127.0.0.1	127.0.0.1	TCP	312	60159 → 2362 [PSH, ACK] Seq=25345 Ack=25601 Win=2593536 Len=256 TStamp=19762498 TSect=19762498
788	0.068197	127.0.0.1	127.0.0.1	TCP	312	2362 → 60158 [PSH, ACK] Seq=25345 Ack=25345 Win=2594048 Len=256 TStamp=19762498 TSect=19762498
789	0.068231	127.0.0.1	127.0.0.1	TCP	56	60159 → 2362 [ACK] Seq=25345 Ack=25601 Win=2593536 Len=0 TStamp=19762498 TSect=19762498
790	0.068246	127.0.0.1	127.0.0.1	TCP	56	2362 → 60159 [ACK] Seq=25601 Ack=25601 Win=2593792 Len=0 TStamp=19762498 TSect=19762498
791	0.068526	127.0.0.1	127.0.0.1	TCP	312	60158 → 2362 [PSH, ACK] Seq=25345 Ack=25601 Win=2593536 Len=256 TStamp=19762498 TSect=19762498
792	0.068535	127.0.0.1	127.0.0.1	TCP	312	2362 → 60159 [PSH, ACK] Seq=25601 Ack=25601 Win=2593792 Len=256 TStamp=19762498 TSect=19762498
793	0.068567	127.0.0.1	127.0.0.1	TCP	56	60159 → 2362 [ACK] Seq=25601 Ack=25857 Win=2593280 Len=0 TStamp=19762499 TSect=19762498
794	0.068579	127.0.0.1	127.0.0.1	TCP	56	2362 → 60158 [ACK] Seq=25601 Ack=25601 Win=2593792 Len=0 TStamp=19762499 TSect=19762499
795	0.068919	127.0.0.1	127.0.0.1	TCP	312	2362 → 60158 [PSH, ACK] Seq=25601 Ack=25601 Win=2593792 Len=256 TStamp=19762499 TSect=19762499
796	0.068958	127.0.0.1	127.0.0.1	TCP	56	60158 → 2362 [ACK] Seq=25601 Ack=25857 Win=2593280 Len=0 TStamp=19762499 TSect=19762499
797	0.071212	127.0.0.1	127.0.0.1	TCP	44	60158 → 2362 [RST, ACK] Seq=25601 Ack=25857 Win=0 Len=0
798	0.071404	127.0.0.1	127.0.0.1	TCP	44	60159 → 2362 [RST, ACK] Seq=25601 Ack=25857 Win=0 Len=0

使用 wireshark 抓包, 可以看到 60158 和 60159 轮流向服务器发包, 服务器轮流回应

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 bind, 服务端才需要

其源端口是系统自动找到一个空闲端口的, 一般是按照递增的顺序查找

一般来说 每次调用 connect 端口不会保持不变, 而是会自增 1, 除非强制指定

- 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?

不会, 因为 TCP 通信里面, accept 会一直阻塞到客户端的一个连接请求到达. 假如断点, 那么客户端的 connect 调用将会尝试在连接到服务器的时候被阻塞掉, 除非服务端断点解除. 只有当服务端先 accept, 客户端的 connect 才会完成, 题目说的情况会让客户端的 connect 连接也被阻塞

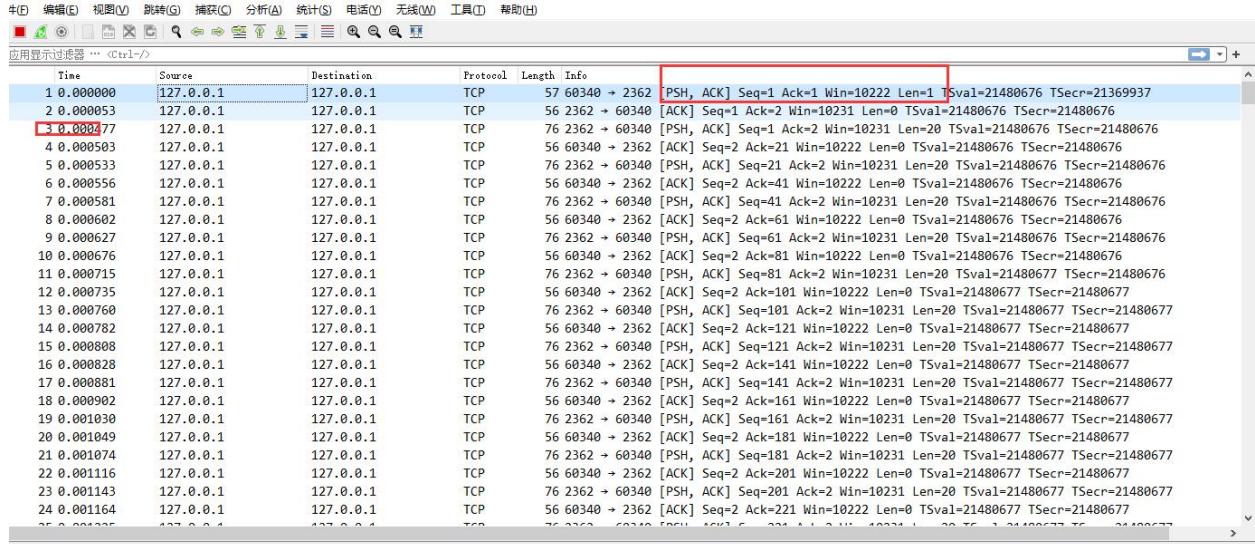
- 连续快速 send 多次数据后, 通过 Wireshark 抓包看到的发送的 Tcp Segment 次数是否和 send 的次数完全一致?
- 从 No8 是第一个 Segment 到 206 是最后一次 Segment, 总共是 $(206-8)/2 + 1$ 正好

是 100 次 Segment

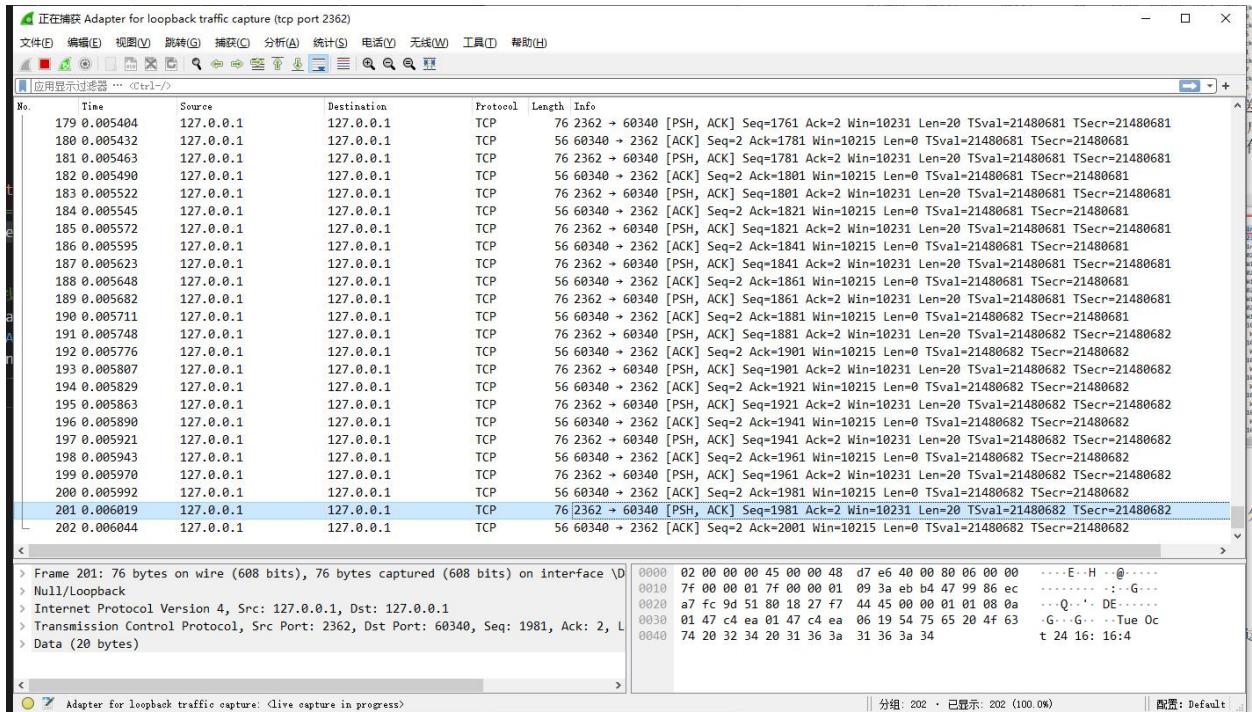
8 1.669890	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=257 Ack=257 Win=256 Tsva=20964861 TSecr=20964857
9 1.669928	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=513 Win=2618624 Len=0 Tsva=20964861 TSecr=20964861
10 1.669965	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=513 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
11 1.669988	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=769 Win=2618368 Len=0 Tsva=20964861 TSecr=20964861
12 1.670018	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=769 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
13 1.670039	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=1025 Win=2618112 Len=0 Tsva=20964861 TSecr=20964861
14 1.670067	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=1025 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
15 1.670089	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=1281 Win=2617856 Len=0 Tsva=20964861 TSecr=20964861
16 1.670116	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=1281 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
17 1.670137	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=1537 Win=2617600 Len=0 Tsva=20964861 TSecr=20964861
18 1.670166	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=1537 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
19 1.670188	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=1793 Win=2617344 Len=0 Tsva=20964861 TSecr=20964861
20 1.670216	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=1793 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
21 1.670237	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=2045 Win=2617088 Len=0 Tsva=20964861 TSecr=20964861
22 1.670265	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=2049 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
23 1.670286	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=2308 Win=2616832 Len=0 Tsva=20964861 TSecr=20964861
24 1.670313	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=2305 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
25 1.670334	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=2561 Win=2616576 Len=0 Tsva=20964861 TSecr=20964861
26 1.670363	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=2561 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
27 1.670385	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=2817 Win=2616320 Len=0 Tsva=20964861 TSecr=20964861
28 1.670413	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=2817 Ack=257 Win=2618880 Len=256 Tsva=20964861 TSecr=20964861
29 1.670435	127.0.0.1	127.0.0.1	TCP	56 60199 → 2362 [ACK] Seq=257 Ack=3073 Win=2616064 Len=0 Tsva=20964861 TSecr=20964861
30 1.670456	127.0.0.1	127.0.0.1	TCP	312 2362 → 60199 [PSH, ACK] Seq=257 Ack=3073 Win=2616064 Len=0 Tsva=20964861 TSecr=20964861

事实上当包比较小的时候，TCP 协议通常会对应用程序发送的数据进行优化，以减少网络上的开销，比如滑动窗口，将包放入缓冲区，使得多个小包合成一起发送

但是本 lab 实现的方式是定长 256B 的固定包，由于本身具有一定的大小，所以发送的时间没有那么快，于是我降低包的大小再次实验



即便是 20B 仍然没有出现问题 还是抓到了 100 个包



- 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？

对于每个客户端来说，都有分配一个专门的线程，每个线程中维护了一个已连接客户端的 SOCKET，由于服务器是以不同套接字来处理的接受包的，所以通过这个实现区分。而且服务器是多线程的，对于每个线程而言，客户端的处理也是独立并发进行的。

- 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）

TCP 连接的状态通常会变为“TIME_WAIT”状态。在 TIME_WAIT 状态下，连接会保持一段时间，以确保所有的传输数据已经成功传送并丢弃。这是 TCP 协议的一部分，用于处理可能存在的延迟数据包，以确保数据的可靠传输和可靠关闭。

TIME_WAIT 状态持续大概是四分钟

- 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？

在客户端拔掉网线后，TCP 没有变化，连接并没有释放，所以当客户端插回了网线，

那么双方原本的 TCP 连接还是能正常存在. 重发数据依然可以接收到.

检测方式:服务器可以设置一个容许时间,在容许时间内如果没有收到数据,就关闭与该客户端的连接.

服务器也可以维护一个计数器,定期向所有客户端发送,如果客户端用完了所有计数还是没有响应认为是断连.

客户端可以捕获异常,并且在意外结束时发送一个信号,这通常适用于用户自主的中断,比如亲自关闭窗口.这个时候服务器可以自动的接受来自客户端的提示信号,本实验的 DISC 用的就是这个机制

此外本实验实现了一个简单的心跳检测,服务器周期性的向客户端发送小包,发送失败就关闭连接.

具体实现如下

```
// 定期给全体客户端发包
void sendHeartbeat() {
    while (serverRunning) {

        for (auto& client : clients_map) {
            SOCKET* clientSocket = client.second.clientSocket;
            char heartbeatPacket[PACKETSIZE + 10] = {0};
            MakeResponsePacket(INDICATE, heartbeatPacket);

            // 发送一个空指示包
            if (send(*clientSocket, heartbeatPacket, PACKETSIZE, 0) == SOCKET_ERROR) {
                // Failed to send heartbeat, update clients_map
                int clientIdx = client.first;
                std::cout << "Heartbeat failed for Client " << clientIdx << std::endl;

                // 如果发送失败就清理掉
                closesocket(*clientSocket);
                clients_map.erase(clientIdx);
            }
        }
        // Wait for a specified interval before sending the next heartbeat
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }
}

Server start:
Client connected. IP: 127.0.0.1, Port: 61369
Client connected. IP: 127.0.0.1, Port: 61371
Heartbeat failed for Client 1
```

这样服务器就能定期知道哪些客户端断掉了,然后进行垃圾清理;

由于新的客户端再建立套接字会开辟新空间,所以这里不会妨碍重连的正确性

七、讨论、心得

问题 1: 数据包的定义:

TCP 使用 Socket 所导致的粘包现象:发送方可能为了效率,将所有数据合成一包同时发出去:所以需要长度来定义. 本实验采取定长的包, 内容使用 len 来维护, 所以避免了这个问题, 因为无论粘包是否发生, 都只会一次读取一个包的大小, 从而避开了变长包无法界定边界的问题,

如果根据读字头来判断是什么包: 问题在于, 可能 data 段中有包头的字符, 导致误判. 所以在非定长的包里面必须考虑起始位置的界定问题, 本实验定长的包同样由于一次是按照 256 的 size 存读的, 所以不会出现界定问题导致的解析错误

定义数据包的长度字段或者结尾标记: 长度表示了有效信息的部分, 如果包是变长的, 那么长度还可以作为检查完整性的单元.

问题在于粘包, 这是指 TCP 协议将多个小包

定义数据包的请求、指示、响应类型字段. 这个放在字头用一个 char 来表示, 所以最多可以支持 256 个状态, 而且如果需要增加再添加更多的包头字节即可, 扩展性比较好.

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)

本实验直接在服务端将信息转化通过包发送过来, 在对应的地方加入了换行符, 所以客户端列表数据只需要读到换行符换行即可.

对于较长的数据, 比如 MSEG 段超过了一个包的大小, 我的做法是, 在第一个包中声明该 MSEG 总共占用的包数, 然后后续包可以维护一个序号实现依次传递数据.

另一种做法是保留位, 默认正常发包是不会占用最后一位的, 所以如果占用了最后一位, 就说明该包没有发完, 这个方法也可以保证连续发包界定消息终止的范围.

问题 2: 如何实现 Client 和 Sever 的退出

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。

使用 join 可以让 main 强制等待所有创建的子线程退出后才结束, 而如何在一直 accept 的主循环中检测到用户的中断信号从而终止服务器呢?

使用 SIGINT 在 main 开始时设置一个信号检测到中断之后, 设置全局变量 ServerRunning=false, 这样就实现了 CtrlC 结束服务器的功能, 而且使用全局变量来提醒所有子进程退出

但是还是有点问题, 最后我了解到使用全局变量进行通信存在问题, 应该使用 CPP 的 atomic 量来实现这一点, 所以无论是用户的手动中断还是来自系统的中断, 只要不是断电这种没有后续处理事件的故障, 都可以让程序检测到当前处理结束, 修改 atomic 量, 终止主循环, 完成后续的收尾工作.

这一点通过本实验断连部分可以体现, 确实套接字消失了.

问题 3:

客户端接收消息是产销模型, 服务器发一个包 对应生产者, Client 接受并处理对应消费一次.

我现在有这样一个模型, 客户端发送信息给服务器, 主线程负责和用户进行 I/O 交互, 而当链接服务器成功以后, 会创建一个负责接受消息的子进程, 这个子进程应该负责循环调用 recv 函数接受服务器发来的消息, 并在处理完消息之后将结果反馈给 main 线程, 这需要用到线程通信

这里采取模拟的方式, 因为在 windows 下消息队列比较难用, 而且不支持管道操作, 这里手动实现了一个消息队列, 开在 main 进程的栈区, 每当子线程接收到一个包, 就把这个包放到消息队列里面, 同时给一个 flag 提醒 main 进程下一次循环记得看一眼消息队列, 然后依次处理消息队列中的信息, 等到队列中的信息都处理完毕, 才会进行下次的输出. 注意加锁, 因为无法保证在处理输出的时候会不会收到新的包.

```
lock_guard<mutex> lock(queueMutex);
Decodemsg(messageQueue.front());
messageQueue.pop();
```

问题 4:

Client 没能成功发包

这是个愚蠢的错误, 因为 memcpy 有问题, 不能做同一片内存上的移动, 需要用更安全的 memmove 来实现

心得

本实验不仅有原理上的难度,还有各种知识综合的难度,此前我从来没有在 windows 下实现多线程项目的开发,而且涉及到 socket 的操作全靠自己一点点搜来学,所以走了不少弯路。这个实验没有止步于最简单的首发模型,而是双向的通信,所以必须要引入线程来实现。服务器扮演生产者,客户端扮演消费者,这是一个经典的产销模型。同时维护消息队列的过程也是个产销模型,主线程定期处理队列信息,子线程不定时加入信息。

不同客户端之间是并行的,这就要求服务器需要轮转时间片来达成类似及时响应的效果。

发包的界定定义问题,还有请求响应的模型,基本模拟了数据链路层的处理。

此外学了一些 TCP 的知识,以及 wireshark 的使用。

对于 Client 需要实现同时 IO 和处理监听信息,对于 Server 要实现 accept 新 client 和为每个已经连接的 client 做处理。建议事先画一个大概的设计图,然后一点点补全各个框架。

后续的并发测试还是比较直观的体现了服务器在处理高频访问时候的状态。总的来说虽然这个实验做了我很久,但是收获还是很大的。