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

Lab1 在线群聊程序

2113946　刘国民

年级：2021 级

专业：信息安全

2023 年 10 月 21 日

目录

一、 实验介绍

本次实验基于 C++ 中基本的 Socket 函数，通过多线程的方式，使用流式套接字实现了多人群聊的功能。该程序有基本的对话界面，输入”q” 后可以正确退出程序。程序基本功能介绍:

- 进入用户端和服务端时，会输出相应的提示信息
- 支持多人在线群聊，新的用户进入群聊后，会向其他用户提示进群的消息，同时提示新用户当前有哪些用户在线
- 群聊时每个用户有自己的用户名，群聊时服务端会将发送时间、用户名和原始消息一起发送给其他用户
- 服务端将历史聊天记录和用户进入/离开在线聊天室的信息显示在控制台上
- 有用户退出在线群聊时，会告诉其他用户该用户已退出群聊，且不影响其他用户正常聊天
- 发生错误时返回错误码

二、 协议设计

(一) 原理

1. 套接字

线程通过套接字发送消息和接收消息。主要包括以下两种类型:

- 数据报套接字: 使用 UDP 协议，支持主机之间面向非连接、不可靠的数据传输
- 流式套接字: 使用 TCP 协议，支持主机之间面向连接的、顺序的、可靠的字节流传输

2. 协议

网络协议是计算机网络中用于交换数据和进行通信的规定和标准。简单来说，它定义了两个或多个通信设备之间如何互相发送和接收数据的规则和约定。

以下为网络协议的组成要素:

- 语法: 数据与控制信息的结构或格式（协议数据单元 PDU）
- 语义: 需要发出何种控制信息，完成何种动作以及做出何种响应
- 时序: 事件实现顺序的详细说明

在 SOCKET 编程中，服务端和客户端通过 SOCKET 套接字来进行通信。按照 Windows SOCKET API 提供的函数接口，符合语法规范和参数含义对函数进行调用。以下给出利用流式套接字通信的过程:

- 服务端/客户端通过 SOCKET 函数创建套接字，绑定传输层协议
- 服务端/客户端通过 bind 函数将 IP 地址和端口与创建的套接字进行绑定
- 客户端发送连接请求（connect）给服务端
- 服务端监听（listen）到来的连接请求，返回新的套接字用于直接通信
- 客户端发送（send）消息给服务端
- 服务端接收（recv）发送的消息
- 服务端/客户端断开（close）连接

本次实验中，一共有三种消息格式:

- 进入消息: 以“$n$ame:”开头的消息，代表用户进入群聊，用这一消息告诉服务端用户名
- 普通消息: time Name:Message，用于转发给其他用户。
- 退出消息: 即字符串 q，代表用户退出群聊，用这一消息通知退出用户的子线程

(二) 实现

按照实验要求和目标，本次实验采取以下协议设计方案:

- 使用 TCP 传输协议，选用流式套接字，采用多线程方式
- 调用 Socket 函数时，输出错误码，便于调试程序
- 设计两个源程序，*Server.cpp* 和 *Client.cpp*。开启程序时需要首先打开 *Server.exe*，再启动多个 *Client.exe*。关闭时需要多个 *Client.exe* 正确退出（输入 q 退出），之后再关闭 *Server.exe*，结束通信。

三、 设计思路

(一) 服务端

首先进入服务端时，会输出提示语句。服务端通过一个全局链表来维护当前成功和服务端连接的用户。链表节点包含用户的 SOCKET 套接字和用户名。

1. 主线程

服务端的 main 函数负责打开 WSA（Windows SOCKET API），进行服务端 IP 地址和端口号的初始化。调用 socket 返回套接字，并通过 bind 将套接字与服务端 IP 地址和端口进行绑定。之后调用 listen 函数持续监听到来的连接请求，如果成功 accept 到新的客户端请求，就为该客户创建新的线程 AcceptThread，并且把 Accept 返回的新套接字传递给创建的线程，将用户信息插入到链表中。

2. 子线程

服务端子线程如果成功 accept 到客户端，则输出“用户 xxx 成功连接的消息”。子线程中服务端通过 recv 来接收连接用户发送的消息。消息一共有三种类型:

- 客户端在用户成功与服务端连接后，会向服务端发送一条“$n$ame:”(这个奇怪的符号是为了避免与用户正常输入的消息相冲突)+ 用户名的消息，用于服务端设置新加入链表节点的用户名，同时将新加入的用户广播给其他用户，即：xx 用户加入群聊，同时遍历链表中所有节点的用户名并发送给这个新加入的用户，告诉该用户目前群聊中有哪些用户在线。
- 对于普通消息，即用户想发送给其他用户的消息。服务器只需遍历整个链表，跳过发送消息的用户，把当前时间、发送者的名字和消息相拼接，然后转发给其他用户。
- 对于退出提示符“q”，需要通知客户端子线程，结束 receive 线程，客户端的 ReceiveThread 线程也需要用“q”来判别，如果客户端收到服务端发回的“q”，则通过 return 0 结束线程；同时需要从链表上删除该用户，表示该用户已下线；最后还需要把该用户退出的信息广播给其他用户。

(二) 客户端

进入客户端后与服务端类似，会输出提示语句。客户端主线程用于发送消息，线程用于接收服务端的消息。

1. 主线程

客户端 main 函数首先完成 WSAStartup、socket 等初始化函数（这里不需要 bind，因为操作系统会自动为客户端选择一个适当的 IP 和一个随机的临时端口号），之后通过 cin.getline 获取用户名，调用 connect 与服务端进行连接，连接成功后会输出提示信息，并且在这里会将“$n$ame:”和输入的用户名拼接后发给服务端，之后则创建线程 ReceiveThread，用于接收服务端发送的消息，主线程则持续获取用户的输入，并且把内容发送给服务端，进而转发给其他用户。如果输入“q”，则会退出该群聊，结束主线程。

2. 子线程

子线程用于接收服务端发来的消息，消息一共有两类：

- 其他用户发送给服务端的消息，服务端进而转发给除发送者外的所有用户，在这里得到消息后正常输出即可。
- 用户自己发送给服务端的退出提示符“q”，服务端发回给用户自己，用于结束子线程。

四、 代码实现

服务端和客户端的主线程实现较为简单，按照 socket 函数的调用语法，进行初始化即可。在此不再赘述，下面给出服务端和客户端子线程中的核心代码片段：

(一) 服务端

给出子线程收到三种类型消息的处理：

1、收到用户退出提示符 q

```
if (strcmp(Message, "q") == 0) {
    // 通知客户端子进程，结束receive
    if (SOCKET_ERROR == send(ClientSocket, Message, sizeof(Message), 0)) {
        cout << "send函数产生错误，具体错误信息见下：" << endl;
        cout << WSAGetLastError() << endl;
    }
    // 从链表上删除该客户端
    char ExitInfo[MAX_NAME_LEN+14] = { 0 };
    strcpy(ExitInfo, "用户");
    for (list<ClientInfo>::iterator curr = mylist.begin(); curr != mylist.end
        (); curr++) {
        if (curr->S_Client == ClientSocket) {
            cout << "用户" << curr->Name << "已断开连接！" << endl;
            strcat(ExitInfo,curr->Name);
            mylist.erase(curr);
            break;
        }
    }
    strcat(ExitInfo, "退出聊天！");
    // 广播给其他用户该用户已退出聊天
```

```
    for (list<ClientInfo>::iterator curr = mylist.begin(); curr != mylist.end(); curr++) {
        if (SOCKET_ERROR == send(curr->S_Client, ExitInfo, sizeof(ExitInfo), 0)) {
            cout << "send函数产生错误，具体错误信息见下：" << endl;
            cout << WSAGetLastError() << endl;
            WSACleanup();
            return 0;
        }
    }
    return 0; // 该用户已退出，结束该进程
}
```

消息 q 用于表明发送该消息的用户已经退出，为了确保聊天程序继续正常执行，需要结束服务端为该用户创建的 Accept 线程，即服务端把该用户正常消息广播出去的线程，还需要结束该用户自己的接收其他用户消息的线程。所以步骤如下：

- 把消息 q 发送回退出用户的客户端接受线程，通知其退出
- 链表上删除该用户的节点
- 服务端后台输出“用户 xxx 断开连接!”
- 以 socket 为关键字遍历链表，把“用户 xxx 退出聊天!”广播给其他用户
- return 0 结束该线程

2、收到前缀为 $n$ame: 的消息

```
// Message是recv到的消息
string str(Message);
string pre = str.substr(0, 7);
if (pre == "$n$ame:") {      // 服务端设置用户名字
    strcpy(mylist.back().Name, Message + 7);// 在链表中存下名字
    cout << "用户" << mylist.back().Name << "成功建立连接！" << endl;

    char to_other[MAX_NAME_LEN+14] = { 0 };                 // 将新加入的用户消息广播给其他用户
    char to_self[100] = { 0 };                              // 用于存储已在线的用户

    strcpy(to_other, "用户");
    strcat(to_other, mylist.back().Name);
    strcat(to_other, "加入聊天！");                 //  “用户xxx加入聊天”推送给其他用户
    strcpy(to_self, "当前在线用户：");

    list<ClientInfo>::iterator self = mylist.begin();
    for (list<ClientInfo>::iterator curr = mylist.begin(); curr != mylist.end(); curr++) {
        if (curr->S_Client != ClientSocket) { // 跳过发送消息的人
```

```
            if (SOCKET_ERROR == send(curr->S_Client, to_other, sizeof(
                to_other), 0)) {
                cout << "send函数产生错误，具体错误信息见下: " << endl;
                cout << WSAGetLastError() << endl;
                WSACleanup();
                return 0;
            }
        }
        else {
            self = curr;
        }
        strcat(to_self, curr->Name);    // 加上用户的名字
        strcat(to_self, "␣␣");
    }
    // 发给新加入的用户to_self
    if (SOCKET_ERROR == send(self->S_Client, to_self, sizeof(to_self), 0)) {
        cout << "send函数产生错误，具体错误信息见下: " << endl;
        cout << WSAGetLastError() << endl;
        WSACleanup();
        return 0;
    }
    continue;
}
```

这类消息是成功连接的用户用来传输用户名字的特殊消息。处理步骤为:

- 服务端后台输出“用户 xxx 成功建立连接”的提示语
- 根据 socket 套接字遍历所有成功连接的用户链表，把“用户 xxx 加入聊天的消息”广播给其他人
- 遍历到新用户自身时用一个临时指针保存，之后把所有在线用户的用户名发送通过临时指针发送给新用户

3、普通消息

```
char MessageInfo[MAX_MESSAGE_LEN + MAX_NAME_LEN + MAX_TIME_LEN + 2] = { 0 };
    // MessageInfo = time  name:Message
strcpy(MessageInfo, GetTimeNow());
strcat(MessageInfo, "\n");
// 遍历链表，根据SOCKET找到发送者名字
for (list<ClientInfo>::iterator curr = mylist.begin(); curr != mylist.end();
    curr++) {
    if (curr->S_Client == ClientSocket) {
        strcat(MessageInfo, curr->Name);
        break;
    }
}
strcat(MessageInfo, ":");
strcat(MessageInfo, Message);
```

```
cout << MessageInfo << endl;
// 对消息进行广播
for (list<ClientInfo>::iterator curr = mylist.begin(); curr != mylist.end();
    curr++) {
    if (curr->S_Client != ClientSocket) { // 跳过发送消息的人
        if (SOCKET_ERROR == send(curr->S_Client, MessageInfo, sizeof(
            MessageInfo), 0)) {
            cout << "send函数产生错误，具体错误信息见下: " << endl;
            cout << WSAGetLastError() << endl;
            WSACleanup();
            return 0;
        }
    }
}
```

普通消息处理较为简单，只需根据 Socket 套接字从链表中找到发送用户的用户名，然后把时间和消息一起拼接后发送给其他用户。

(二) 客户端

给出子线程收到两种类型消息的处理:

```
DWORD WINAPI ReceiveThread(LPVOID param) {
                                                    // 用来接收服务器发送的消息
    char ReceiveMessage[MAX_MESSAGE_LEN + MAX_NAME_LEN + MAX_TIME_LEN + 3] =
        { 0 };      // 名字最大长度和消息最大长度和冒号
    SOCKET ClientSocket = (SOCKET)param;
                                                        // 类型转换，拿到主线
        程传递的参数
    while (true) {
        if (SOCKET_ERROR == recv(ClientSocket, ReceiveMessage, sizeof(
            ReceiveMessage), 0)) {
            cout << "receive接收失败" << endl;
            cout << WSAGetLastError() << endl;
            break;
        }
        if (strcmp(ReceiveMessage, "q") == 0) {
            // 客户输入q退出时，服务端会发送q到接收进程，关闭该进程
            return 0;
        }
        cout << ReceiveMessage << endl << endl;
    }
    return 0;// 正常退出
}
```

如上述设计思路所说，用户端的子线程用于接收两类消息:

- 其他用户发送给服务端的消息，服务端进而转发给除发送者外的所有用户，在这里得到消息后正常输出即可。

- 用户自己发送给服务端的退出提示符“q”, 服务端发回给用户自己，用于结束子线程。

所以在接收到消息后，如果是退出消息 q，则直接 return 0 结束线程，如果非 q 则为正常消息，直接输出即可。

五、 程序运行

下面分析程序执行流程和通过 WireShark 捕获的消息

(一) 流程分析

如图，首先打开 1 个服务端程序和 3 个客户端程序

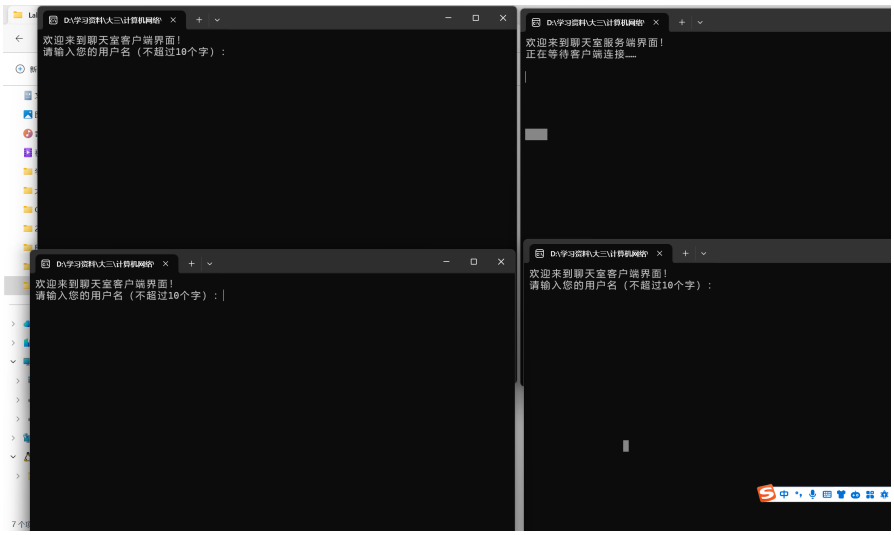

图 1: 初始化

然后按照 NKU1-NKU2-NKU3 的顺序输入用户名进入群聊，可以看到，其他用户进入时，已经在线的用户会收到通知

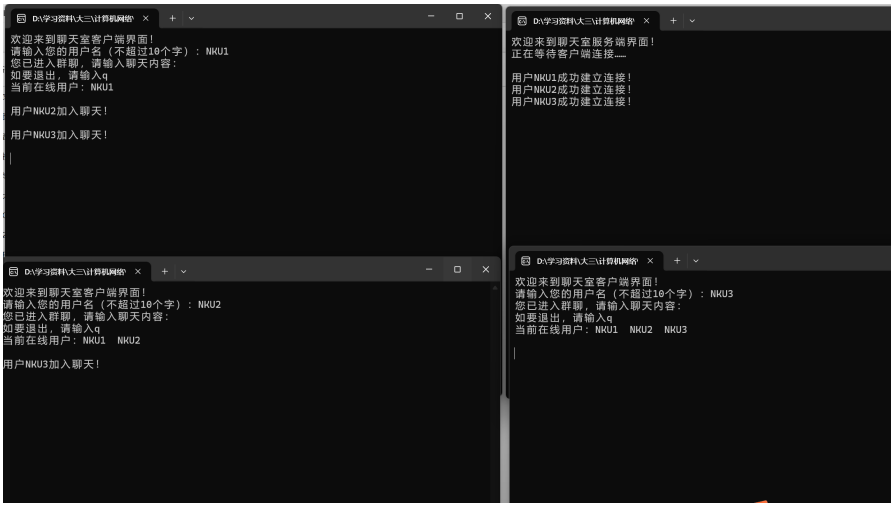

图 2: 进入群聊

之后开始发送消息，其他用户都可以正常收到消息，后台服务端正常输出聊天记录

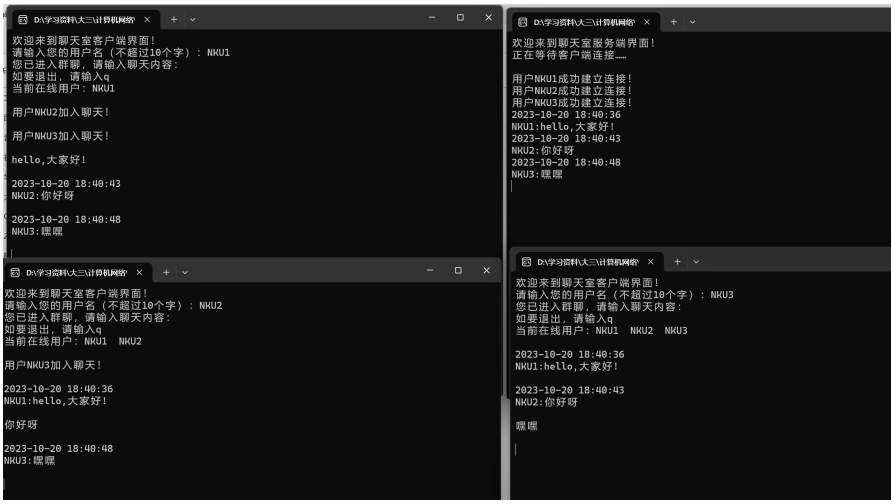

图 3: 群聊中

NKU3 退出后，其他用户可以收到退出信息，服务端也显示 NKU3 断开连接

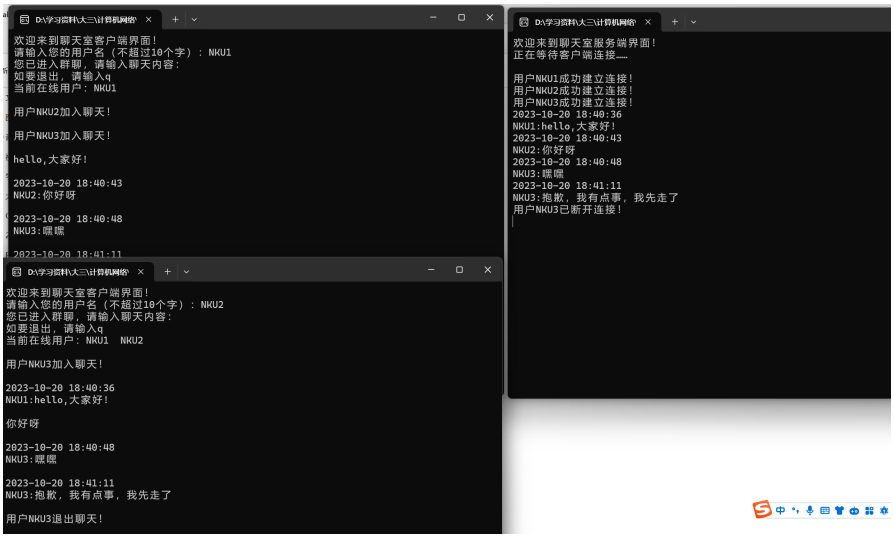

图 4: 一人退出

所有人退出后，服务端中显示聊天记录和成功/断开连接的记录

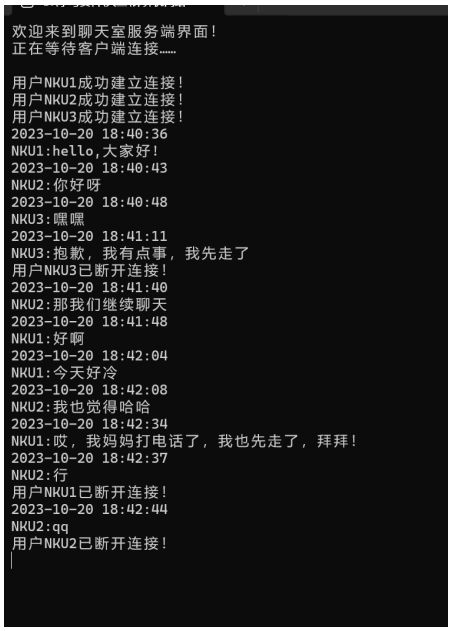

图 5: 后台记录

(二) 抓包分析

下面用 WireShark 软件对网络通信进行分析，首先打开软件，点击 Adapter for loopback traffic capture, 用于抓取本地主机（即 127.0.0.1 或 localhost）上的通信数据。

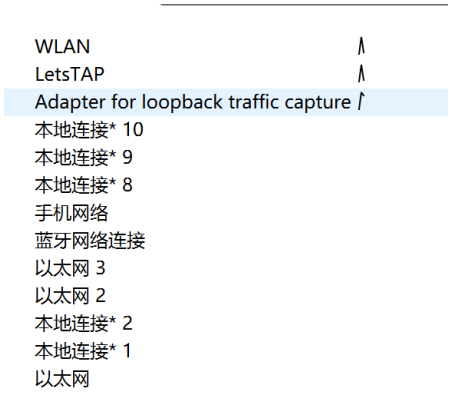

图 6: WireShark 初始化

打开后, 因为在实验中使用 TCP 协议来进行流式传输，以及服务端端口号为 8080，所以设置输出过滤:$tcp.port == 8080$

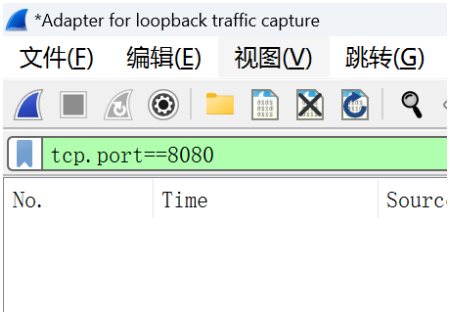

图 7: 过滤条件

点击开始捕获后，运行可执行文件，输入第一个用户的名字“NKU1”，按照实验代码，用户端会向服务端传输一个“$n$ame:NKU1”的消息，如下图可以看到，已经成功捕获到该消息。另外在图中，我也看到了 TCP 的三次握手过程，即前三行内容：[SYN]、[SYN,ACK]、[ACK]，第四行即是发出的消息。同时根据 TCP 协议的特点，发送方发送一个带有 [PSH, ACK] 标志的数据包，接收方收到后会回应一个带有 [ACK] 标志的数据包，表示已经收到发送的数据。第四、五内容验证了这一特点。而第六行消息则是服务端返回给用户的当前在线用户的消息，第七行则是 [ACK] 应答，符合预期。通过抓包看到，TCP 协议是更像是“一问一答”的模式，在实际通信中发送方也在接收接收方回应的数据包。

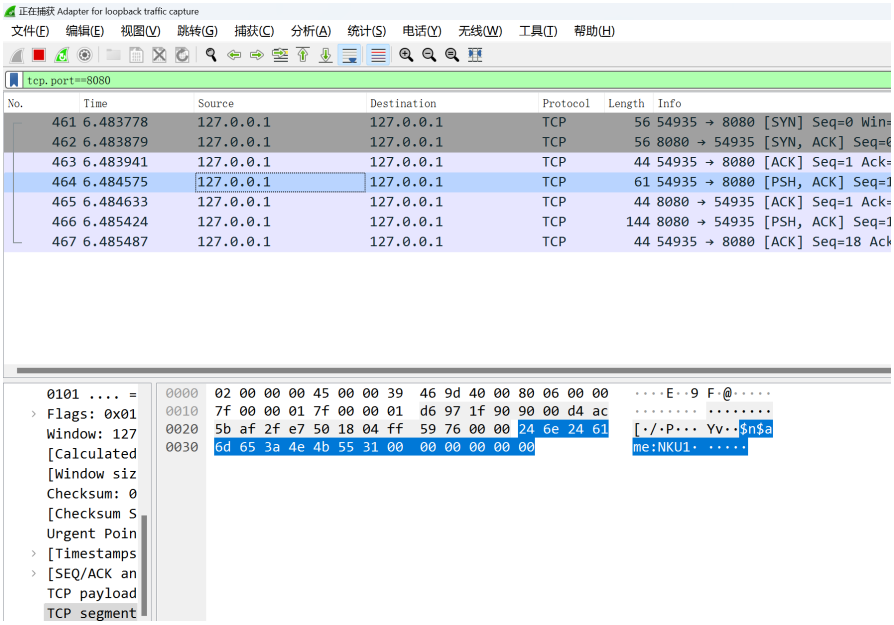

图 8: 捕获结果

点开第四行消息查看详细内容，可以看到使用的是 IPv4 协议，源端口号为 54935（这个端口号是操作系统为客户端分配的），目的端口号即为服务端设置的端口号（8080）。

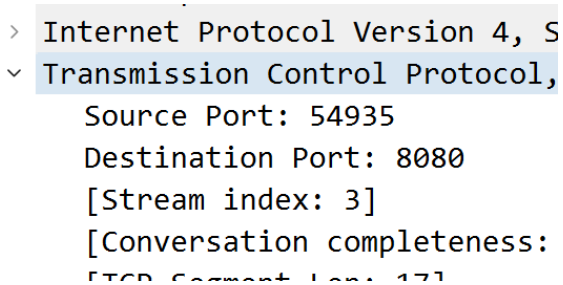

图 9: 详细信息

综上，通过 WireShark 能够捕获到符合实验预期的通信内容，验证成功。

六、 问题 & 解决方案

在实验过程中，遇到了一系列问题并且最终成功解决

问题 1

线程的创建以及结束，实验要求使用多线程的方式完成聊天程序。由于对线程的相关知识了解不足，在编写代码时不能正确处理线程函数。在用户端输入 q 退出线程的时候，用户端主函数 return 0 结束线程，但子线程仍在接收服务器发送的内容。会出现 recv 失败的情况。

解决

用户端输入 q 退出时，仍会将 q 发送给服务端，并且服务端接收到后，以 SOCKET 套接字为索引，发送给用户端的子线程。子线程接收到 q 后通过 return 0 即可正常退出。

问题 2

用户端输入名字后，每次发送消息时，可以在用户端把时间、用户名和原始消息封装起来一起发给服务端，但每发送一次就需要读取一次用户名，效率较低。

解决

用全局链表维护在线用户，用户连接成功后就立即向服务端发送一条包含用户名特殊消息，服务端拿到后把用户名存储在链表中，下次再收到该用户发送的消息时就从链表中获取用户名.

七、 思考总结

在本次实验中，由于对线程等相关知识还不够熟悉，对线程的处理只考虑了基本的创建和退出功能，没有考虑并发等情况，之后使用多线程方式编程时，会尝试加入互斥量等机制来避免可能出现的问题。通过本次实验，我初步了解了 socket 编程的编程方式，对理论知识有了更深入的了解，熟悉了如何采用多线程编程，熟悉了如何使用 WireShark 进行抓包分析，提高了编程能力。