

实验一：利用Socket编写一个聊天程序

物联网工程_2111194_胡博程

一、实验要求

摘自学院网站。

利用Socket编写一个聊天程序

要求：

- (1) 给出你聊天协议的完整说明。
- (2) 利用C或C++语言，使用基本的Socket函数完成程序。不允许使用CSocket等封装后的类编写程序。
- (3) 使用流式套接字、采用多线程（或多进程）方式完成程序。
- (4) 程序应有基本的对话界面，但可以不是图形界面。程序应有正常的退出方式。
- (5) 完成的程序应能支持多人聊天，支持英文和中文聊天。
- (6) 编写的程序应该结构清晰，具有较好的可读性。
- (7) 在实验中观察是否有数据的丢失，提交源码和实验报告。

二、前期准备

1、开发环境搭建

使用vscode集成开发环境，编程需要使用winsock2.h，用于支持套接字编程，提供了例如 `SOCKET`——用于表示套接字连接的数据类型以及 `socket()`、`bind()`、`listen()`、`accept()`、`send()` 和 `recv()` 等必要的函数。所有需要的头文件如下

```
#include <iostream>
#include <winsock2.h>
#include <string>
#include <vector>
#include <thread>
#include <cstdint>
#include <codecvt>
#include <locale>
#include <algorithm>
#include <unordered_map>
```

此外我们还需要使用ws2_32的动态链接库，因为winsock2.h头文件中定义的函数，实际的实现都位于ws2_32.dll中，设置vscode中Code Runner插件的配置文件settings.json如下，编译链接时选用-lws2_32命令链接额外的动态链接库

```
{
  "code-runner.executorMap": {
    "c": "cd $dir && gcc $fileName -o  
${fileDirname}\\output\\$fileNameWithoutExt -  
L$workspaceRoot\\Network_technology\\SDK\\Lib\\x64 -lwpcap -lPacket && cd  
$dir\\bin\\ && $fileNameWithoutExt",
    "cpp": "cd $dir && g++ $fileName -o $fileNameWithoutExt -lws2_32 -  
L$workspaceRoot\\Network_technology\\SDK\\Lib\\x64 -lwpcap -lPacket &&  
./$fileNameWithoutExt"
  },
  "files.associations": {
    "iostream": "cpp",
    "ostream": "cpp",
    "array": "cpp",
    "atomic": "cpp",
    "/*.tcc": "cpp",
    "bitset": "cpp",
    "cctype": "cpp",
    "chrono": "cpp",
    "clocale": "cpp",
    "cmath": "cpp",
    "cstdarg": "cpp",
    "cstddef": "cpp",
    // 余下很多，不一一列举
  }
}
```

2、字符编码设置

题目要求实现中文对话，需要使用支持中文字符的编码，但是由于vscode中集成终端使用windows本身的cmd终端，cmd终端不支持utf-8编码，只支持GBK2312编码，尝试将cmd终端的编码改为utf-8完成本实验。



但是由于beta版本的cmd还是无法完全支持utf-8编码，且由于程序需要处理的中文字符不仅来源于工作区的cpp文件中的字符常量、还源于输入输出流或者cmd终端的输入输出，这些部分设计复杂的编码机制，所有上面的尝试失败。导致输入输出流的中文编码可以正常显示但是cpp文件中的常量中文字符不可以正确展现。

随后决定在vscode的工作区中设置编码为GBK2312，且在数据跨进程传递的过程中使用自定义的函数 `ConvertToUTF8` 与 `ConvertFromUTF8` 函数进行编码转换，使得进程间通信的编码正确

```
行 1, 列 1 (已选择206)  空格: 4  GB 2312  CRLF  {} C++
```

三、协议设计

1、程序实现的预期效果

- 客户端之间消息可以传递，是聊天程序的基本功能，一个客户端内输入的消息可以传递到其他所有客户端和服务端
- 客户端内设置一些指令如"QUIT", "SHOW", "CHANGE NAME" 等指令由服务器进行解析，功能分别为退出服务器，展示客户端总数目、改变客户端用户名功能

由上面两点我们发现：**需要设计协议让服务器辨别消息和指令**

- 支持中文字符聊天

由上面的一点我们发现：**需要在协议中规定编码及其转换方式保证中文字符的正确显示**

- 采用多线程（或多进程）方式完成程序。

由上面的一点我们发现：**需要在协议中规定服务器端和客户端多线程的实现细节**

- 赋予新加入的服务器的客户端一个默认的命名

由上面的一点我们发现：**需要在协议中规定客户端命名的相关细节**

- 程序应有正常的退出方式。

由上面的一点我们发现：**客户端退出连接的时候不仅需要在自己的进程中退出，还需要维护服务器端正常运行，这涉及程序退出时的时序设计**

- 多人聊天室的人数存在上限

由上面的一点我们发现：**客户端在加入聊天区的时候需要告诉客户端我想加入，服务器也要同意并告诉客户端你能加入。当聊天区满时，服务器告诉客户端拒绝加入。上述接受和拒绝加入的消息需要特殊的消息格式或内容完成**

2、详细的协议设计

采用流式套接字，使用TCP协议进行通信，面向连接的、可靠的、基于字节流的传输层协议。此外，本实验采用单服务器——多客户端的组织方式组织多客户端之间聊天。

除此以外在传输数据的结构方面，我们将传输数据前加上了一个类型头，标注后面的数据是一个消息类型的数据，还是一个指令类型的数据。具体的举例分析如下。

消息类型数据的字段：<type头>:<源数据>——Message: 123

指令类型数据的字段：<type头>:<源数据>——Command: 123

总体协议三要素设计分析

1. 语法：

- **传输数据的结构**：套接字传输的数据包含type头与源数据本身，type头有 Message 和 Command 两种，Message表示的源数据将会由服务器广播给所有的客户端，Command 表示的源数据为客户端发送给服务器且不广播的，是一些特殊的操作（比如关闭客户端、展示群聊现有人数等）
- **编码问题**：ConvertToUTF8()：将一个字符串从本地编码转换为 UTF-8；ConvertFromUTF8()：将一个 UTF-8 编码的字符串转换为本地编码。数据在传输以前调用函数 ConvertToUTF8()，接收端接受到数据后调用函数 ConvertFromUTF8()，可以确保数据的兼容性、完整性和正确性。

2. 语义：

- **Command指令设计**：“QUIT”, “SHOW”, “CHANGE NAME” 等指令由服务器进行解析，功能分别为退出服务器，展示客户端总数目、改变客户端用户名功能
- **Message标识**：该标识后的数据为广播数据，服务器简单地将这些数据广播向所有的客户端即可
- **赋予初始用户名**：当一个客户端加入群聊，服务器端需要给这个用户赋予一个初始的用户名，这时我们规定这个由服务器端发送到客户端，为了赋予客户端用户名的信息开头为“**Your ID**：”，当服务器发送这样的数据到客户端，客户端将会把用户名切换成“Your ID后面的字段”，这个信息再Command指令中的CHANGE NAME也被用到了。同时“Your ID”的发送也**正式意味着客户端加入了群聊，这个信息与下面决绝新客户端加入的返回信息对立。**
- **聊天人数达上限时拒绝新客户端加入**：此时我们规定服务器将会短暂与欲加入的客户端建立连接，同时给他发送一个“**CONNECTION_REJECTED**”消息，这之后客户端的socket将被释放。
-

3. 时序：

- **通讯总体的链接与线程规定**：
 - 服务器端启动，绑定IP，监听端口，线程分离循环接受客户端消息
 - 客户端启动，与服务器建立连接，线程分离准备接受服务器消息，本线程负责发送消息给服务器
- **客户加入群聊时的时序规定**：服务器端将查看人数上限，未达上限则给这个客户端赋予一个默认ID，具体细节是服务器端给客户端发消息“**Your name <名字>**”，标志着客户端加入成功。
- **聊天室中客户端达上限时的时序规定**：此时新的客户端无法继续加入聊天室，但是这时候TCP的链接已经建立，我们的协议设计服务器短暂接受新的连接，并给新的客户端返回一个“**CONNECTION_REJECTED**”，客户端接受到这个信息后将关闭socket连接，并退出程序。
- **客户端退出聊天室时的时序规定**：客户端发送“**QUIT**”到服务器端，服务器不会广播这个消息，而是将其从socket列表（代码中用vector）删除并关闭socket连接、退出程序。
- **客户端执行指令改变用户名的时序规定**：客户端发送“**CHANGE NAME <名字>**”时服务器会将更新socket到用户名的映射数组（代码中的clientNameMap），同时返回一个消息“**Your name <名字>**”，这时客户端会修改本地的用户名为新的名字

聊天程序主要还是遵循TCP协议，其三要素设计详细解释如下：

1. 语法 (Syntax)：

- **报文格式**：TCP 报文段包括源端口、目标端口、序列号、确认号、数据偏移、保留、控制位（如 SYN, ACK, FIN, RST 等）、窗口大小、校验和、紧急指针以及选项和填充。
- **编码**：TCP 报文段的字段是按照特定的格式和长度进行编码的，这确保了发送方和接收方都能正确地解析和处理报文段。

2. 语义 (Semantics)：

- **连接管理**：TCP 是面向连接的，所以 SYN, SYN-ACK, ACK 等控制位用于建立和终止连接。
- **数据传输**：数据字段携带应用数据。序列号和确认号用于确保数据的可靠传输。
- **流控制**：窗口大小字段允许接收方控制发送方的发送速率，这防止了发送方“淹没”接收方。
- **错误处理**：校验和字段用于检测数据在传输过程中的错误。

3. 时序 (Timing)：

- **三次握手**：在建立连接时，TCP 使用三次握手（SYN, SYN-ACK, ACK）来同步双方的序列号和确认连接的建立。
- **数据传输**：数据报文段的传输和确认需要遵循特定的时序，以确保可靠性。

- **拥塞控制**: TCP 使用多种机制（如慢启动、拥塞避免、快速恢复等）来检测和响应网络拥塞，这些机制都涉及到特定的时序。
- **四次挥手**: 在终止连接时，TCP 使用四次挥手（FIN, ACK, FIN, ACK）来确保双方都知道连接已经结束，且所有数据都已正确传输。

四、实现思路

1、服务器端

(1) 首先初始化socket并创立服务器端的socket

使用 `WSAStartup` 来初始化Winsock库，并用 `socket` 函数创建了一个新的TCP套接字。这个套接字将作为服务器端的主要套接字，用于监听和接受来自客户端的连接请求。

```
WSADATA wsaData;  
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)  
{  
    cerr << "初始化socket失败." << endl;  
    return 1;  
}  
cout << "初始化socket成功" << endl;  
  
SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0); // 创建一个TCP套接字
```

(2) 绑定服务器地址

为了使客户端能够连接到服务器，服务器需要绑定其套接字到一个特定的IP地址和端口。这里使用的是通配IP地址（即 `INADDR_ANY`），表示服务器将在其所有可用的网络接口上监听。

```
sockaddr_in server_address;  
server_address.sin_family = AF_INET;  
server_address.sin_port = htons(PORT);  
server_address.sin_addr.S_un.S_addr = INADDR_ANY;  
bind(server_socket, (sockaddr *)&server_address, sizeof(server_address));
```

(3) 监听客户端端口信息，并循环接收客户端消息

服务器使用 `listen` 函数开始在指定的端口上监听客户端的连接请求。之后进入一个无限循环，在这个循环中，它使用 `accept` 函数等待并接受客户端的连接请求。

```
listen(server_socket, MAX_CLIENTS); // 开始监听来自客户端的连接  
cout << "客户端正在监听端口 " << PORT << endl;  
  
while (true)  
{  
    if (cur_client < MAX_CLIENTS)  
    {  
        SOCKET client_socket = accept(server_socket, NULL, NULL); // 接受新的  
        客户端连接  
        clients.push_back(client_socket); // 将新的客  
        户端套接字添加到clients向量中  
        cur_client++;  
    }
```

```

        clientNameMap[client_socket] = "Client" +
std::to_string(cur_client);

        cout << "客户端 " << client_socket << "加入聊天" << endl;
        cout << "当前客户端数量: " << cur_client << "/" << MAX_CLIENTS << endl;

        thread(handle_client, client_socket).detach(); // 为每一个客户端创建一个
新的线程来处理其消息

        string idMessage = "YourID:client" + to_string(cur_client);
        send(client_socket, idMessage.c_str(), idMessage.size(), 0);
    }
    else
    {
        cout << "拒绝一个连接请求" << endl;
    }
}

```

(4) 循环接受消息时采用多线程

使用了多线程来处理每个连接到服务器的客户端。具体来说，每当一个新的客户端连接到服务器时，就会为这个客户端创建一个新的线程来处理与它的通信。这样可以使得服务器能够同时处理多个客户端，而不是在处理一个客户端时阻塞其他客户端的请求。

```
thread(handle_client, client_socket).detach();
```

创建了一个新的线程，并将 `handle_client` 函数作为线程的入口点，同时将 `client_socket` 作为参数传递给 `handle_client` 函数。然后，它调用 `detach()` 函数，将新线程与当前线程分离，使得新线程可以独立运行。这样，`handle_client` 函数就可以在新线程中运行，而不会阻塞当前线程。

`handle_client` 函数将在下一小部分讲解。

(5) 不同种类的消息采用不同的处理

首先使用 `memset` 函数清空缓冲区，然后使用 `recv` 函数从客户端接收消息，并将消息存储在缓冲区 `buffer` 中。如果接收到的字节数小于等于 0，说明客户端已经断开连接，它会关闭客户端套接字。如果正常，代码根据客户端传输来的消息的 type 头划分消息类型，Message 类型的数据由服务器分发给所有客户端，Command 类型的数据将由服务器处理，调用的函数是 `process_command(rawCommand, client_socket)` 【下一小部分讲】

```

void handle_client(SOCKET client_socket)
{
    char buffer[BUFFER_SIZE];
    while (true)
    {
        memset(buffer, 0, BUFFER_SIZE); // 清空
缓冲区
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0); // 从客
户端接收消息
        // cout << "BUFFER:" << buffer << endl;

        if (bytes_received <= 0)
        { // 如果接收到的字节少于或等于0，说明客户端已断开连接
            closesocket(client_socket);
            cout << "Client disconnected." << endl;
        }
    }
}

```

```

        break;
    }

    string message1(buffer, bytes_received);
    string message = ConvertFromUTF8(message1);

    if (message.find("Command") == 0) // 传输来的是指令——处理系统指令
    {
        cout << "需处理系统指令，指令来自" << clientNameMap[client_socket] <<
endl;

        string rawCommand = message.substr(8); // 清除类型头**和冒号**!
        cout << "处理指令: " << rawCommand << endl; // test point
        process_command(rawCommand, client_socket);
        continue; // 不执行消息广播
    }
    if (message.find("Message") == 0) // 传输来的是消息
    {
        cout << "将接收客户端消息" << endl;
        message = message.substr(8); // 清除类型头**和冒号**!
    }
    // 打印来自哪个客户端的消息
    std::cout << "Message from " << clientNameMap[client_socket] << ':' <<
message << std::endl;

    // 给消息附上来源
    message = clientNameMap[client_socket] + ":" + message;

    // 广播消息给其他所有客户端
    // cout << clients.size() << endl; // test point
    for (SOCKET client : clients)
    {
        if (client != client_socket)
            // if (true)
            {
                // string message_with_id = "NO." +
to_string(global_message_id++) + ":" + message;
                string utf8Message = ConvertToUTF8(message);
                send(client, utf8Message.c_str(), utf8Message.size(), 0);
            }
    }
}
}

```

(6) 关闭服务器套接字并清理winsock

```

closesocket(server_socket); // 关闭服务器套接字
WSACleanup(); // 清理winsock
return 0;

```

(7) 补充一些用到的辅助函数

- `process_command(string commandStr, SOCKET client_socket)` 函数
 - 参数设置：一个是命令消息的字符串表示，另一个是发送命令消息的客户端套接字
 - 执行流程：首先判断命令消息的内容，根据指令的不同采用不同的解决方案，如果是 "QUIT"，它会从 `clients` 向量中删除客户端套接字，并输出一条提示；如果是 "SHOW"，它会输出一条提示信息表示客户端要查询人数，并将当前人数发送给客户端。如果是

"HELP" 或 "CLEAR", 则表示客户端要查询帮助信息或清空聊天记录, 这里不需要服务器端任何操作。如果是 "CHANGE NAME", 则表示客户端要修改用户名, 它会从命令消息中提取新的用户名, 并将其更新到 `clientNameMap` 中。如果是其他未知指令, 则输出一条提示信息表示未知指令。

```
void process_command(string commandStr, SOCKET client_socket)
{
    if (commandStr == "QUIT")
    {
        clients.erase(std::remove(clients.begin(), clients.end(),
client_socket), clients.end()); // 删除客户端套接字
        clientNameMap.erase(client_socket);

        cout << "客户端" << clientNameMap[client_socket] << "退出聊天" <<
endl;
        cur_client--;
        cout << "清理客户端socket-----complete" << endl;
        cout << "当前客户端数量: " << cur_client << "/" << MAX_CLIENTS <<
endl;
        int temp = client_socket;
        closesocket(client_socket);
        // cout << "message:" << message << endl; // test point
    }
    else if (commandStr == "SHOW")
    {
        cout << "客户" << clientNameMap[client_socket] << "将获得以下信息" <<
endl;
        cout << "当前客户端数量: " << cur_client << "/" << MAX_CLIENTS <<
endl;
        string temp = "Serve:" + to_string(cur_client) + "/" +
to_string(MAX_CLIENTS);
        send(client_socket, temp.c_str(), 100, 0);
    }
    else if (commandStr == "HELP")
    {
        ;
    }
    else if (commandStr == "CLEAR")
    {
        ;
    }
    else if (commandStr.find("CHANGE NAME ") == 0)
    {
        string newName = commandStr.substr(12); // ?为什么是12
        string oldName = clientNameMap[client_socket];
        clientNameMap[client_socket] = newName;
        cout << "客户端" << client_socket << "将用户名从" << oldName << "修改
为" << newName << endl;
    }
    else
    {
        cout << "未知指令" << endl;
    }
}
```

- `ConvertToUTF8()` 函数与 `ConvertFromUTF8()` 函数

前者将一个 ANSI 编码的字符串转换为 UTF-8 编码的字符串。先使用 `MultiByteToWideChar` 函数将 ANSI 编码的字符串转换为宽字符编码的字符串，然后使用 `WideCharToMultiByte` 函数将宽字符编码的字符串转换为 UTF-8 编码的字符串。如果转换过程中出现错误，函数会返回一个空字符串

后者将一个 UTF-8 编码的字符串转换为 ANSI 编码的字符串。首先使用 `MultiByteToWideChar` 函数将 UTF-8 编码的字符串转换为宽字符编码的字符串，然后使用 `WideCharToMultiByte` 函数将宽字符编码的字符串转换为 ANSI 编码的字符串。如果转换过程中出现错误，函数会返回一个空字符串。

2、客户端

(1) 首先初始化socket并创建客户端的socket

```
WSADATA wsaData; // 用于初始化Winsock库
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
{
    std::cerr << "Failed to initialize winsock." << std::endl;
    return 1;
}
cout << "初始化socket成功" << endl;

SOCKET client_socket = socket(AF_INET, SOCK_STREAM, 0); // 创建一个TCP套接字
// 参数解释
// AF_INET: 使用IPv4地址
// SOCK_STREAM: 面向网路的流式套接字，即TCP套接字
// 0: 自动选择协议
if (client_socket == INVALID_SOCKET)
{
    cout << "创建客户端socket失败" << endl;
    return -1;
}
cout << "创建客户端socket成功" << endl;
```

调用 `WSAStartup` 函数初始化 Winsock 库，如果初始化失败，则输出一条错误信息并返回。然后，它调用 `socket` 函数创建一个 TCP 套接字，如果创建失败，则输出一条错误信息并返回。

(2) 初始化服务器参数，客户端socket与服务器建立连接。

```
// 配置服务器地址
sockaddr_in server_address;
server_address.sin_family = AF_INET; // 使用IPv4地址
server_address.sin_port = htons(PORT); // 端口号
server_address.sin_addr.S_un.S_addr = inet_addr("127.0.0.1"); // 服务器IP地址
// 服务器的 IP 地址为本地回环地址 127.0.0.1。
// connect操作
if (connect(client_socket, (sockaddr *)&server_address,
sizeof(server_address)) == SOCKET_ERROR)
{
    std::cerr << "Connect failed. Error: " << WSAGetLastError() <<
std::endl;
    if (WSAGetLastError() == WSAECONNREFUSED)
    { // 检查连接是否被服务器拒绝
        std::cerr << "服务器拒绝了连接请求。请稍后重试。" << std::endl;
    }
    closesocket(client_socket); // 关闭客户端套接字
```

```

WSACleanup(); // 清理winsock
return -1;
}
cout << "成功连接到服务器" << endl;
cout << "聊天界面初始化成功" << endl;

```

声明 `sockaddr_in` 结构体变量 `server_address`，据其成员变量配置IPV4地址、端口号和服务器IP地址。随后用 `connect` 函数将客户端套接字连接到服务器的套接字地址 `server_address` 上。如果连接失败，它会输出一条错误信息，表示连接失败的原因，并关闭客户端套接字，清理 Winsock 库，返回 -1。如果连接成功，会输出提示信息。其中，如果 `WSAGetLastError` 函数返回 `WSAECONNREFUSED` 错误码，则表示连接被服务器拒绝，这时它会输出一条提示信息，让用户稍后重试

(3) 创建线程接受来自服务器端的消息

```

std::thread t(receive_messages, client_socket); // 创建一个新的线程来接收消息
if (t.joinable())
{
    t.detach(); // 将线程与当前线程分离
    cout << "线程创建成功" << endl;
}
else
{
    // 线程创建失败，进行错误处理
    std::cerr << "线程创建失败" << std::endl;
    return -1;
}

```

创建了一个新的线程，并将 `receive_messages` 函数作为线程的入口点，同时将 `client_socket` 作为参数传递给 `receive_messages` 函数。使用 `joinable` 函数检查线程是否可以加入到当前线程中。如果线程可以加入，说明线程创建成功，它调用 `detach` 函数将线程与当前线程分离，随后按需输出提示信息。

回调函数 `receive_messages` 如下

```

// 接收来自服务器的消息的函数
void receive_messages(SOCKET client_socket)
{
    char buffer[BUFFER_SIZE]; // 用于接收消息的缓冲区
    while (true)
    {
        memset(buffer, 0, BUFFER_SIZE); // 清空缓冲区
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0); // 从服务器接收消息

        if (bytes_received <= 0)
        { // 如果接收到的字节少于或等于0，说明与服务器的连接已断开
            std::cerr << "已断开与服务器的链接" << std::endl;
            closesocket(client_socket);
            exit(1);
            // return; // 退出线程，而不是退出整个程序
        }
        if (bytes_received > 0 && strcmp(buffer, "CONNECTION_REJECTED") == 0)
        {
            std::cerr << "连接被服务器关闭，聊天室已满。" << std::endl;
            closesocket(client_socket);
        }
    }
}

```

```

        WSACleanup();
        exit(1);
    }

    std::string message1(buffer, bytes_received); // 将接收到的字符数组转换为
string字符串
    string message = ConvertFromUTF8(message1);

    // cout << "message!!!" << message << endl; // test point
    // cout << message.substr(7) << endl;
    if (message.find("YourID") != string::npos)
    {
        client_name = message.substr(7);
        // 使用 substr 函数从字符串 message 的第 6 个字符开始提取子字符串, 然后使用
stoi 函数将子字符串转换为整数类型
        // str.substr(7, 5)表示从字符串 str 的第 7 个字符开始提取长度为 5 的子字符串
        cout << "Your name is " << client_name << endl; // test point
        continue;
    }

    if (message.find("ACK") != string::npos)
    {
        continue;
    }
    cout << message << endl;
}
}

```

使用一个循环不断接收服务器发送的消息，直到与服务器的连接断开。在循环中，首先使用 `memset` 函数清空缓冲区，然后使用 `recv` 函数从服务器接收消息，并将消息存储在缓冲区 `buffer` 中。

若接收到的字节数小于等于 0，可以认为与服务器的连接已经断开，随后关闭客户端套接字，清理 Winsock 库，并退出程序。如果接收到的消息是 "CONNECTION_REJECTED"，说明服务器已经满员，随后关闭客户端套接字，清理 Winsock 库，并退出程序。如果接收到的消息包含 "YourID"，说明客户端已经成功连接到服务器，并且服务器已经为其分配了一个唯一的 ID，它会从消息中提取客户端的名称，并输出一条提示信息表示客户端的名称。如果接收到的消息包含 "ACK"，说明客户端发送的消息已经被服务器确认，它会跳过本次循环，继续等待下一条消息。

最后在客户端控制台输出接收到的消息。

(4) 本线程读取客户端输入并根据输入进行不同处理

```

std::string message;
while (true)
{
    std::getline(std::cin, message); // 从控制台读取用户输入

    // string temp = "哈哈";
    // cout << temp << endl;
    string type = typeMessage;

    if (message == "QUIT")
    {
        type = typeCommand;
        string fullMessage = type + ":" + message;
        string utf8FullMessage = ConvertToUTF8(fullMessage);
    }
}

```

```

        send(client_socket, utf8FullMessage.c_str(), utf8FullMessage.size(),
0);
        break;
    }
    else if (message.find("CLEAR") == 0)
    {
        type = typeCommand;
        string fullMessage = type + ":" + message;
        string utf8FullMessage = ConvertToUTF8(fullMessage);
        send(client_socket, utf8FullMessage.c_str(), utf8FullMessage.size(),
0);

        system("cls");
    }
    else if (message.find("CHANGE NAME ") == 0)
    {
        client_name = message.substr(12);
        type = typeCommand;
        string fullMessage = type + ":" + message;
        string utf8FullMessage = ConvertToUTF8(fullMessage);
        send(client_socket, utf8FullMessage.c_str(), utf8FullMessage.size(),
0);
    }
    else if (message.find("SHOW") == 0)
    {
        type = typeCommand;
        string fullMessage = type + ":" + message;
        string utf8FullMessage = ConvertToUTF8(fullMessage);
        send(client_socket, utf8FullMessage.c_str(), utf8FullMessage.size(),
0);
    }
    else if (message.find("HELP") == 0)
    {
        cout << "<=====>" << endl;
        cout << "HELP:显示帮助信息" << endl;
        cout << "SHOW:显示当前聊天室数目" << endl;
        cout << "QUIT:关闭客户端" << endl;
        cout << "CLEAN:清屏" << endl;
        cout << "CHANGE NAME <new name>: 更改用户名" << endl;
        cout << "<=====>" << endl;
    }
    else // 本客户端发送消息
    {
        type = typeMessage;
        string fullMessage = type + ":" + message;
        string utf8FullMessage = ConvertToUTF8(fullMessage);
        send(client_socket, utf8FullMessage.c_str(), utf8FullMessage.size(),
0);

        cout << client_name << "(ME):" << message << std::endl;
    }
}

```

使用 `std::getline` 函数从控制台读取用户输入，并将输入存储在变量 `message` 中，然后根据输入的内容判断消息类型，并将消息类型和消息内容拼接成一个完整的消息字符串。如果输入的是 "QUIT"、"CLRAR"、"CHANGE NAME"、"SHOW"、"HELP" 命令，它会将消息类型设置为命令类型，并将完整的消息字符串发送到服务器处理，本地也会根据这些命令进行一些客户端界面的展示。如果发现

输入不为命令而是消息，程序将把消息类型头附加到原始消息上，并将完整的消息字符串发送到服务器，然后输出消息内容。

(5) 关闭客户端套接字并退出程序

在死循环以外的代码，上面的死循环遇到QUIT命令将退出循环来到下面的代码，关闭套接字、清理winsock、退出程序。

```
closesocket(client_socket); // 关闭客户端套接字
WSACleanup();              // 清理winsock
return 0;
```

五、效果展示

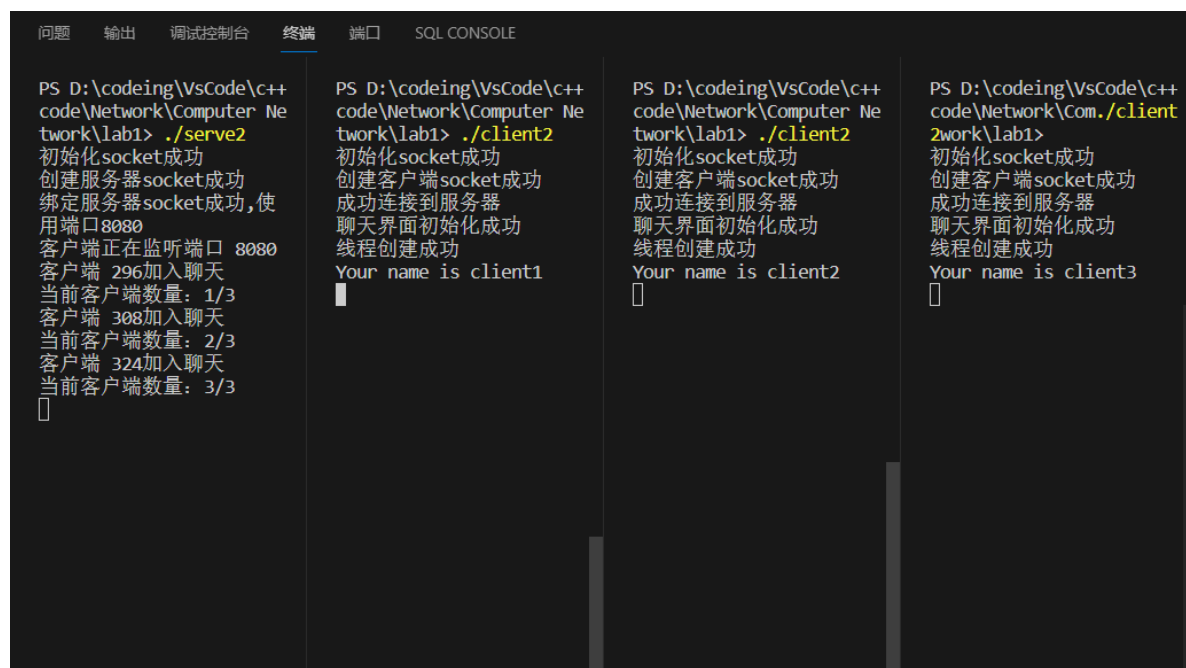
下面展示基于最多三人聊天，人数上限与相关配置可以在代码开头的全局变量设置中修改

```
const int PORT = 8080;          // 服务器端口
const int BUFFER_SIZE = 2048;   // 缓冲区大小
const int MAX_CLIENTS = 3;      // 最大客户端数量
const string typeCommand = "Command";
const string typeMessage = "Message";

int cur_client = 0; // 当前客户端数量

vector<SOCKET> clients; // 客户端套接字向量
std::unordered_map<SOCKET, std::string> clientNameMap; // 客户端套接字与客户端名字的映射
```

1、客户端加入



```
PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./serve2
初始化socket成功
创建服务器socket成功
绑定服务器socket成功,使用端口8080
客户端正在监听端口 8080
客户端 296加入聊天
当前客户端数量: 1/3
客户端 308加入聊天
当前客户端数量: 2/3
客户端 324加入聊天
当前客户端数量: 3/3
[]

PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2
初始化socket成功
创建客户端socket成功
成功连接到服务器
聊天界面初始化成功
线程创建成功
Your name is client1
[]

PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2
初始化socket成功
创建客户端socket成功
成功连接到服务器
聊天界面初始化成功
线程创建成功
Your name is client2
[]

PS D:\codeing\VsCode\c++code\Network\Com./client2work\lab1>
初始化socket成功
创建客户端socket成功
成功连接到服务器
聊天界面初始化成功
线程创建成功
Your name is client3
[]
```

客户端加入，服务器端实时更新在线人数，并给加入成功的客户端分配一个用户名，在正常聊天的时候会显示各客户端客户的用户名。

2、正常聊天（支持中英文）

| | | | |
|--|--|--|---|
| <pre>PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./serve2 初始化socket成功 创建服务器socket成功 绑定服务器socket成功,使用端口8080 客户端正在监听端口 8080 客户端 296加入聊天 当前客户端数量: 1/3 客户端 308加入聊天 当前客户端数量: 2/3 客户端 324加入聊天 当前客户端数量: 3/3 将接收客户端消息 Message from Client1:hello! 将接收客户端消息 Message from Client2:你好 将接收客户端消息 Message from Client3:高兴见到你 []</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client1 hello! client1(ME):hello! Client2:你好 Client3:高兴见到你 []</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client2 Client1:hello! 你好 client2(ME):你好 Client3:高兴见到你 []</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Com./client2work\lab1> 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client3 Client1:hello! Client2:你好 高兴见到你 client3(ME):高兴见到你 []</pre> |
|--|--|--|---|

各用户发的消息会在服务器端集中、打印、再广播，如上图所示。

3、客端更改名字的操作

使用命令 `CHANGE NAME <名字>` 更改用户名，服务器端显示更名日志，随后客户端发送消息使用新的用户名，也支持中文字符。

| | | | |
|--|--|--|---|
| <pre>客户端正在监听端口 8080 客户端 296加入聊天 当前客户端数量: 1/3 客户端 308加入聊天 当前客户端数量: 2/3 客户端 324加入聊天 当前客户端数量: 3/3 将接收客户端消息 Message from Client1:hello! 将接收客户端消息 Message from Client2:你好 将接收客户端消息 Message from Client3:高兴见到你 需处理系统指令, 指令来自Client1 处理指令: CHANGE NAME 张三 客户端296将用户名从Client1修改为张三 将接收客户端消息 Message from 张三:我叫张三</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client1 hello! client1(ME):hello! Client2:你好 Client3:高兴见到你 CHANGE NAME 张三 我叫张三 张三(ME):我叫张三 []</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client2 Client1:hello! 你好 client2(ME):你好 Client3:高兴见到你 张三:我叫张三 []</pre> | <pre>PS D:\codeing\VsCode\c++code\Network\Com./client2work\lab1> 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client3 Client1:hello! Client2:你好 高兴见到你 client3(ME):高兴见到你 张三:我叫张三 []</pre> |
|--|--|--|---|

4、其他命令操作

使用 `SHOW` `HELP` `CLEAR` 命令时，程序运行情况如下

| | | | |
|--|---|--|--|
| Message from Client1:hello! 将接收客户端消息 Message from Client2:你好 将接收客户端消息 Message from Client3:高兴见到你 需处理系统指令, 指令来自 Client1 处理指令: CHANGE NAME 张三 客户端296将用户名从Client1修改为张三 将接收客户端消息 Message from 张三:我叫张三 需处理系统指令, 指令来自 张三 处理指令: SHOW 客户张三将获得以下信息 当前客户端数量: 3/3 将接收客户端消息 Message from Client2:HEL ELP [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client1 hello! client1(ME):hello! Client2:你好 Client3:高兴见到你 CHANGE NAME 张三 我叫张三 张三(ME):我叫张三 SHOW Serve:3/3 Client2:HEL ELP [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client2 Client1:hello! 你好 client2(ME):你好 Client3:高兴见到你 张三:我叫张三 HELP client2(ME):HEL ELP [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client3 Client1:hello! Client2:你好 高兴见到你 client3(ME):高兴见到你 张三:我叫张三 Client2:HEL ELP CLEAR [] |
|--|---|--|--|

行 14 列 1 (已选择410)

| | | | |
|---|---|--|--|
| Message from Client2:你好 将接收客户端消息 Message from Client3:高兴见到你 需处理系统指令, 指令来自 Client1 处理指令: CHANGE NAME 张三 客户端296将用户名从Client1修改为张三 将接收客户端消息 Message from 张三:我叫张三 需处理系统指令, 指令来自 张三 处理指令: SHOW 客户张三将获得以下信息 当前客户端数量: 3/3 将接收客户端消息 Message from Client2:HEL ELP 需处理系统指令, 指令来自 Client3 处理指令: CLEAR [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client1 hello! client1(ME):hello! Client2:你好 Client3:高兴见到你 CHANGE NAME 张三 我叫张三 张三(ME):我叫张三 SHOW Serve:3/3 Client2:HEL ELP [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client2 Client1:hello! 你好 client2(ME):你好 Client3:高兴见到你 张三:我叫张三 HELP client2(ME):HEL ELP [] | |
|---|---|--|--|

5、服务器已满拒绝新服务器加入

现在设定人数上限为3, 继续有新的客户端加入的时候会拒绝其加入, 其余客户端的通信不受影响。

| | | | | |
|--|---|--|--------------------------------|--|
| 高兴见到你 需处理系统指令, 指令来自 Client1 处理指令: CHANGE NAME 张三 客户端296将用户名从Client1修改为张三 将接收客户端消息 Message from 张三:我叫张三 需处理系统指令, 指令来自 张三 处理指令: SHOW 客户张三将获得以下信息 当前客户端数量: 3/3 将接收客户端消息 Message from Client2:HEL ELP 需处理系统指令, 指令来自 Client3 处理指令: CLEAR 将接收客户端消息 Message from Client3:我 还在哦 拒绝一个连接请求 [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client1 hello! client1(ME):hello! Client2:你好 Client3:高兴见到你 CHANGE NAME 张三 我叫张三 张三(ME):我叫张三 SHOW Serve:3/3 Client2:HEL ELP Client3:我还在哦 [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 Your name is client2 Client1:hello! 你好 client2(ME):你好 Client3:高兴见到你 张三:我叫张三 HELP client2(ME):HEL ELP Client3:我还在哦 [] | 我还在哦 client3(ME):我还在哦 [] | PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> ./client2 初始化socket成功 创建客户端socket成功 成功连接到服务器 聊天界面初始化成功 线程创建成功 连接被服务器关闭, 聊天室已满。 PS D:\codeing\VsCode\c++code\Network\Computer Network\lab1> [] |
|--|---|--|--------------------------------|--|

6、客户端正常退出

使用 QUIT 命令后，客户端将会断开连接，服务器端也会输出相应的日志，并不影响其他两个客户端之间的通信。退出客户端3之后，张三和client2还能正常通讯

```
当前客户端数量: 3/3
将接收客户端消息
Message from Client2:HEL
P
需处理系统指令，指令来自
Client3
处理指令: CLEAR
将接收客户端消息
Message from Client3:我
还在哦
拒绝一个连接请求
需处理系统指令，指令来自
Client3
处理指令: QUIT
客户端退出聊天
清理客户端socket-----com
plete
当前客户端数量: 2/3
Client disconnected.
将接收客户端消息
Message from Client2:你
好
将接收客户端消息
Message from 张三:嗯，
你好
[]

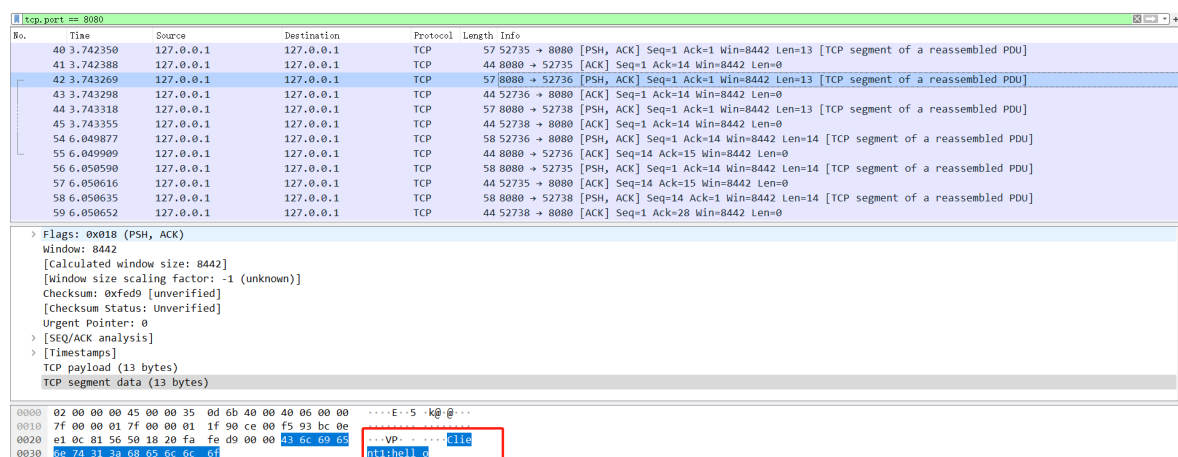
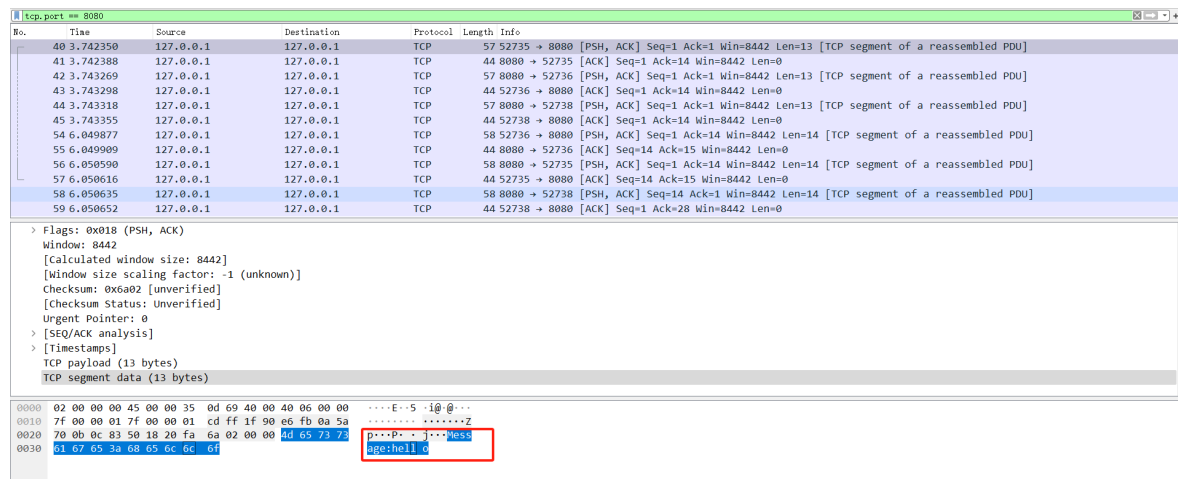
PS D:\codeing\VsCode\c++
code\Network\Computer Ne
twork\lab1> ./client2
初始化socket成功
创建客户端socket成功
成功连接到服务器
聊天界面初始化成功
线程创建成功
Your name is client1
hello!
client1(ME):hello!
Client2:你好
Client3:高兴见到你
CHANGE NAME 张三
我叫张三
张三(ME):我叫张三
SHOW
Serve:3/3
Client2:HELP
Client3:我还在哦
Client2:你好
嗯，你好
张三(ME):嗯，你好
[]

PS D:\codeing\VsCode\c++
code\Network\Computer Ne
twork\lab1> ./client2
初始化socket成功
创建客户端socket成功
成功连接到服务器
聊天界面初始化成功
线程创建成功
Your name is client2
Client1:hello!
你好
client2(ME):你好
Client3:高兴见到你
张三:我叫张三
HELP
client2(ME):HELP
Client3:我还在哦
你好
client2(ME):你好
张三:嗯，你好
[]

我还在哦
client3(ME):我还在哦
QUIT
已断开与服务器的链接
PS D:\codeing\VsCode\c++
code\Network\Computer Ne
twork\lab1> []
```

7、检查是否有数据丢失情况

使用TCP协议，一般不会有数据丢失的情况，使用wireshark抓包，监听回环地址127.0.0.1的8080端口，检查数据包具体情况



| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|----------|-----------|-------------|----------|--------|---|
| 40 | 3.742350 | 127.0.0.1 | 127.0.0.1 | TCP | 57 | 52735 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=8442 Len=13 [TCP segment of a reassembled PDU] |
| 41 | 3.742388 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 8080 → 52735 [ACK] Seq=1 Ack=14 Win=8442 Len=0 |
| 42 | 3.743269 | 127.0.0.1 | 127.0.0.1 | TCP | 57 | 8080 → 52736 [PSH, ACK] Seq=1 Ack=1 Win=8442 Len=13 [TCP segment of a reassembled PDU] |
| 43 | 3.743298 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 52736 → 8080 [ACK] Seq=1 Ack=14 Win=8442 Len=0 |
| 44 | 3.743318 | 127.0.0.1 | 127.0.0.1 | TCP | 57 | 8080 → 52738 [PSH, ACK] Seq=1 Ack=1 Win=8442 Len=13 [TCP segment of a reassembled PDU] |
| 45 | 3.743355 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 52738 → 8080 [ACK] Seq=1 Ack=14 Win=8442 Len=0 |
| 54 | 6.049877 | 127.0.0.1 | 127.0.0.1 | TCP | 58 | 52736 → 8080 [PSH, ACK] Seq=1 Ack=14 Win=8442 Len=14 [TCP segment of a reassembled PDU] |
| 55 | 6.049909 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 8080 → 52736 [ACK] Seq=14 Ack=15 Win=8442 Len=0 |
| 56 | 6.050590 | 127.0.0.1 | 127.0.0.1 | TCP | 58 | 8080 → 52735 [PSH, ACK] Seq=1 Ack=14 Win=8442 Len=14 [TCP segment of a reassembled PDU] |
| 57 | 6.050616 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 52735 → 8080 [ACK] Seq=14 Ack=15 Win=8442 Len=0 |
| 58 | 6.050635 | 127.0.0.1 | 127.0.0.1 | TCP | 58 | 8080 → 52738 [PSH, ACK] Seq=14 Ack=1 Win=8442 Len=14 [TCP segment of a reassembled PDU] |
| 59 | 6.050652 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 52738 → 8080 [ACK] Seq=1 Ack=28 Win=8442 Len=0 |

> Flags: 0x018 (PSH, ACK)
 Window: 8442
 [Calculated window size: 8442]
 [Window size scaling factor: -1 (unknown)]
 Checksum: 0xdc42 [unverified]
 [Checksum Status: Unverified]
 Urgent Pointer: 0
 > [SEQ/ACK analysis]
 > [Timestamps]
 TCP payload (14 bytes)
 TCP segment data (14 bytes)

| | | | | |
|------|-------------------------|-------------------------|------------------|----------|
| 0000 | 02 00 00 00 45 00 00 36 | 0d 77 40 00 40 06 00 00 |E--6 | 00000000 |
| 0010 | 7f 00 00 01 7f 00 00 01 | ce 00 1f 90 e1 0c 81 56 |V | 00010000 |
| 0020 | f5 93 bc 1b 50 18 20 fa | dc 42 00 00 4d 65 73 73 |P...B...Miss | 00020000 |
| 0030 | 61 67 65 3a e4 bd a0 e5 | a5 b6 | age..... | 00030000 |

并没有数据丢失，出现省略号是因为wireshark无法解析中文字符。