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：

学　　院：　计算机学院

系：　计算机系

专　　业：　信息安全

学　　号：

指导教师：　张泉方

2020 年 12 月 27 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：贾双铖　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

```
struct data{
    char time[20];
    int targetID;
    int contentLength;
    char packageType;
    char actionType;
    char body[];
}model;
```

time	返回当前时间
targetID	返回自己的目标 ID
contentLength	返回当前连接的所有用户的信息长度
body	消息字段
packageType	数据包类型
actionType	服务功能类型

- 描述响应数据包的格式（画图说明），响应类型的定义

```
struct data{
    char time[20];
    int targetID;
    int contentLength;
    char packageType;
    char actionType;
    char body[];
}model;
```

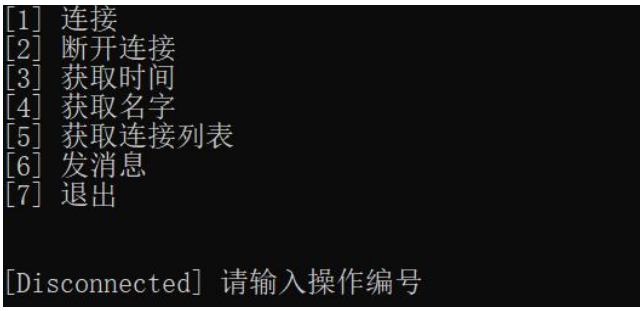
time	返回当前时间
targetID	返回自己的目标 ID
contentLength	返回当前连接的所有用户的信息长度
body	消息字段
packageType	数据包类型
actionType	服务功能类型

- 描述指示数据包的格式（画图说明），指示类型的定义

```
struct connection{
    u_short port;
    struct in_addr  ip;
    SOCKET  clientSock;
    int     state;
}clientList[LIST_SIZE];
```

port	端口
ip	IP 地址
clientSock	如服务器转发客户端的消息内容和类型
state	连接状态

- 客户端初始运行后显示的菜单选项



- 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 (mainState != STOP){
    int op;
    if (connectState){
        printf("\n[Connected] 请输入操作编号\n");
    } else {
        printf("\n[Disconnected] 请输入操作编号\n");
    }
    fflush(stdin);
    scanf("%d",&op);
    if (!connectState && op >= 3 && op <= 6){
        printf("请连接服务器\n");
        continue;
    }
    switch (op) {
        case 1:
            mainState = BLOCK;
            connectServer();
            break;
        case 2:
            closeSocket();
            break;
        case 3:
            mainState = BLOCK;
                messageBuffer[0] = '\0';
                packData(PACKAGE_TYPE_GET,ACTION_TYPE_TIME);
            break;
        case 4:
            mainState = BLOCK;
            messageBuffer[0] = '\0';
            packData(PACKAGE_TYPE_GET,ACTION_TYPE_NAME);
            break;
        case 5:
            mainState = BLOCK;
            messageBuffer[0] = '\0';
            packData(PACKAGE_TYPE_GET,ACTION_TYPE_CLIENT_LIST);
            break;
        case 6:
            mainState = BLOCK;
            printf("请输入目标的ID\n");
            scanf("%d",&id);
            sendMessage();
            break;
        case 7:
            mainState = STOP;
            closeSocket();
            break;
        default:
            printf("操作无效\n");
            break;
    }
    while(mainState == BLOCK);
}
```

- 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）

```
while (messageState != STOP){
    if (messageState == BLOCK){
        struct data* unpacked = unpackData();
        switch (unpacked->packageType) {
            case PACKAGE_TYPE_RESPONSE:
                if (unpacked->actionType == ACTION_TYPE_TIME){
                    printf("服务器当前时间: %s",unpacked->time);
                } else if ( unpacked->actionType == ACTION_TYPE_CONNECT_SUCCESS){
                    connectState = true;
                    printf("%s",unpacked->body);
                } else {
                    printf("%s",unpacked->body);
                }
                break;
            case PACKAGE_TYPE_ORDER:
                if (unpacked->actionType == ACTION_TYPE_CONNECT_FAIL){
                    closeSocket();
                    printf("\n[Disconnected] 请输入操作编号\n");
                } else if (unpacked->actionType == ACTION_TYPE_SEND_MESSAGE){
                    printf("[Receive]\n对方客户端编号：%d\n发送时间：%s",unpacked->targetID,unpacked->time);
                    printf("%s",unpacked->body);
                    printf("\n[Connected] 请输入操作编号\n");
                }
                break;
        }
        free(unpacked);
        while (messageState == BLOCK) messageState = RUN;
        while (mainState == BLOCK) mainState = RUN;
    }
}
```

- 服务器初始运行后显示的界面

```
[Hello] 服务器已启动(127.0.0.1:5507)，最大连接数 3
```

- 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）

```
while (mainState){
    clientSock = accept(servSock, (SOCKADDR*)&clientAddr, &nSize);
    char tempPort[2];
    tempPort[1] = *((char*)&((*(SOCKADDR_IN*)&clientAddr).sin_port));
    tempPort[0] = *((char*)&((*(SOCKADDR_IN*)&clientAddr).sin_port)+1);
    u_short port = *(u_short*)&tempPort;
    printf("[Request] 收到%s:%d的连接请求 \n",
           inet_ntoa(*(struct in_addr*)&clientAddr.sa_data[2]),port);
    int i;
    for (i = 0; i < LIST_SIZE; i++){
        if (clientList[i].state == STOP) {
            clientList[i].ip = (*(SOCKADDR_IN*)&clientAddr).sin_addr;
            clientList[i].port = port;
            clientList[i].state = RUN;
            clientList[i].clientSock = clientSock;
            break;
        }
    }
    if (i == LIST_SIZE){
        int randomKill = rand() % LIST_SIZE;
        sprintf(messageBuffer,"连接已终止\n");
        printf("[Ponit] 随机关闭并使用 %d 号作为新的连接\n",randomKill);
        packData(PACKAGE_TYPE_ORDER,ACTION_TYPE_CONNECT_FAIL,clientList[randomKill].clientSock);
        handleState[randomKill] = STOP;
        clientList[randomKill].ip = (*(SOCKADDR_IN*)&clientAddr).sin_addr;
        clientList[randomKill].port = port;
        clientList[randomKill].clientSock = clientSock;
        i = randomKill;
    }
    _beginthread(handleRequest,0,(void*)&i);
}
```

- 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）

```
while (handleState[index]){
    int a = recv(client, requestBuffer, MAXWORD, 0);
    if(a < 0) break;
    if (handleState[index] == STOP) break;
    struct data* unpacked = unpackData();
    switch (unpacked->actionType) {
        case ACTION_TYPE_TIME:
            messageBuffer[0] = '\0';
            printf("[Response] 响应%d号客户端的时间请求\n",index);
            packData(PACKAGE_TYPE_RESPONSE, ACTION_TYPE_TIME, client);
            break;
        case ACTION_TYPE_NAME:
            sprintf(messageBuffer,"服务器名称为: 3180105507\n");
            printf("[Response] 响应%d号客户端的名称请求\n",index);
            packData(PACKAGE_TYPE_RESPONSE, ACTION_TYPE_NAME, client);
            break;
        case ACTION_TYPE_CLOSE:
            handleState[index] = STOP;
            clientList[index].state = STOP;
            printf("[Request] 关闭%d号客户端的连接\n",index);
            break;
        case ACTION_TYPE_CLIENT_LIST:
            handleState[index] = BLOCK;
            printf("[Response] 响应%d号客户端的列表请求\n",index);
            clientListResponse(index, client);
            break;
        case ACTION_TYPE_SEND_MESSAGE:
            handleState[index] = BLOCK;
            sendMessage(index, client, unpacked);
            break;
    }
```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

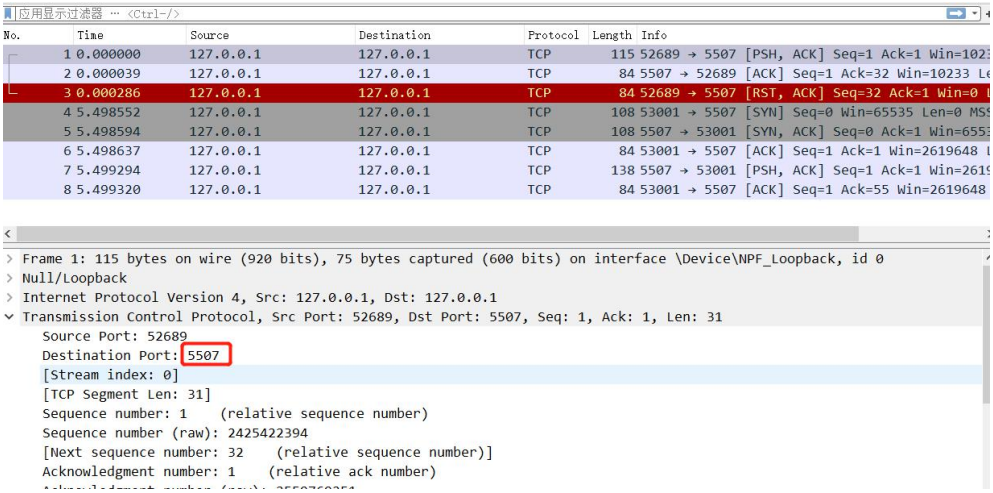
  服务端：

```
[Hello] 服务器已启动(127.0.0.1:5507)，最大连接数 3
[Request] 收到127.0.0.1:3244的连接请求
[Hello] 已建立新连接，编号： 0
```

  客户端：

```
[Disconnected] 请输入操作编号
1
请输入目标IP(127.0.0.1)与端口(5507)
127.0.0.1 5507
[Hello] 已连接至服务器
```

  Wireshark 抓取的数据包截图：



- 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。

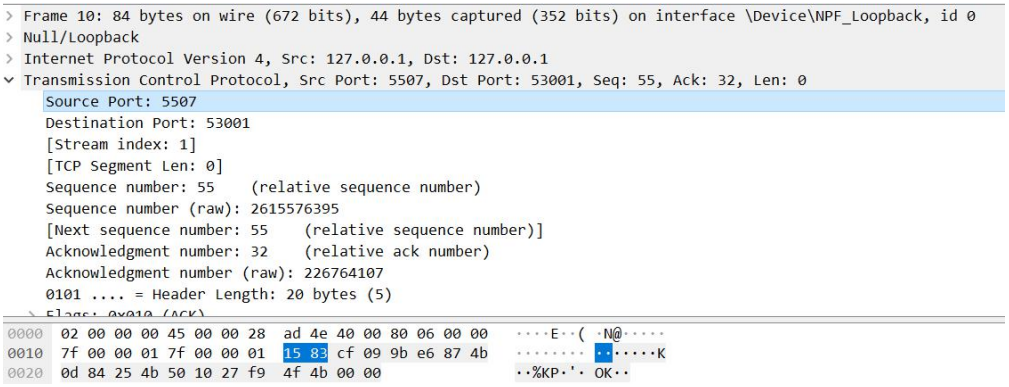
  客户端：

```
[Disconnected] 请输入操作编号
[Hello] 已连接至服务器
3
服务器当前时间: 2021/01/13 21:31:31
```

  服务端：

```
[Response] 响应0号客户端的时间请求
```

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）：

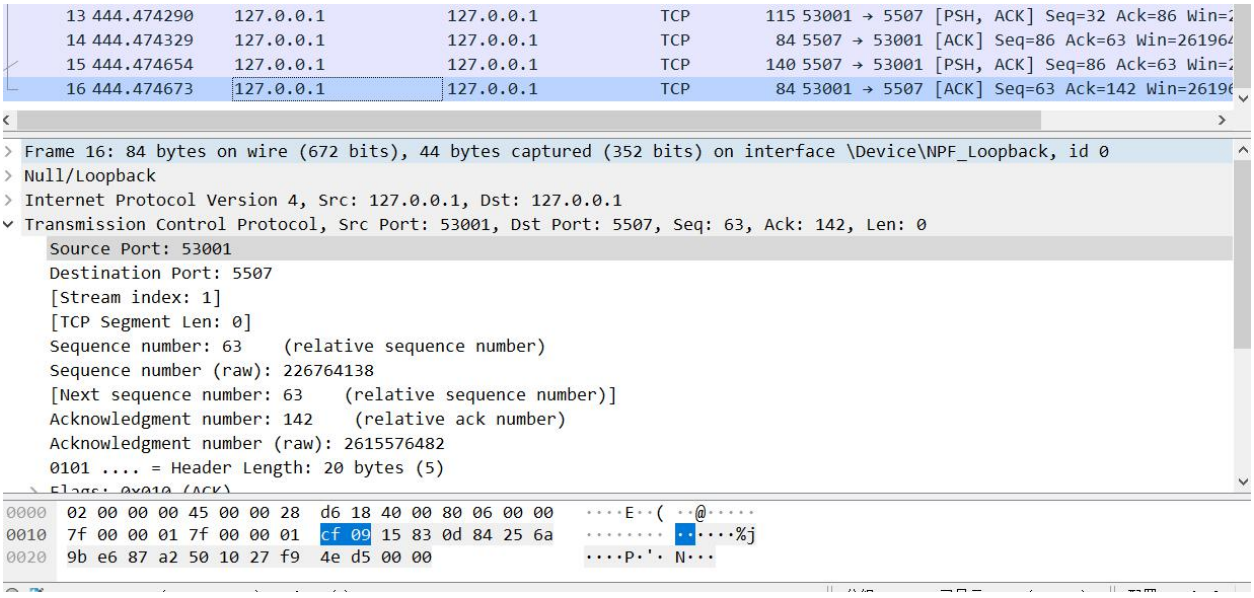


- 客户端选择获取名字功能时，客户端和服务端显示内容截图。

客户端：

服务端：

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）：



相关的服务器的处理代码片段：

```
void packData(char type, char action, SOCKET targetSock)
{
    memset(responseBuffer, 0, sizeof(responseBuffer));
    SYSTEMTIME sys;
    GetLocalTime(&sys);
    sprintf(responseBuffer, "%4d/%02d/%02d %02d:%02d:%02d\n", sys.wYear, sys.wMonth, sys.wDay, sys.wHour, sys.
    wMinute, sys.wSecond);
    int length = (int)strlen(messageBuffer);
    strcpy(&responseBuffer[CONTENT_LENGTH_BIT], (char *)&length);
    strcpy(&responseBuffer[SEND_MSG_SOURCE_BIT], (char *)&id);
    responseBuffer[PACKAGE_TYPE_BIT] = type;
    responseBuffer[ACTION_TYPE_BIT] = action;
    strcpy(&responseBuffer[CONTENT_BIT], messageBuffer);
    send(targetSock, responseBuffer, CONTENT_BIT + length + sizeof(char), 0);
    memset(responseBuffer, 0, sizeof(responseBuffer));
}
```

- 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：

```
[Connected] 请输入操作编号
5
[0] 客户端 -- 来自127.0.0.1:53001
1个客户端与服务器相连，当前客户端编号：0
```

服务端：

```
[Response] 响应0号客户端的列表请求
```

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、返回的客户端列表数据对应的位置）：

```
    17 494.698037    127.0.0.1       127.0.0.1       TCP      115 53001 → 5507 [PSH, ACK] Seq=63 Ack=142 Win=
    18 494.698091    127.0.0.1       127.0.0.1       TCP       84 5507 → 53001 [ACK] Seq=142 Ack=94 Win=26196
    19 494.698253    127.0.0.1       127.0.0.1       TCP      191 5507 → 53001 [PSH, ACK] Seq=142 Ack=94 Win=
    20 494.698279    127.0.0.1       127.0.0.1       TCP       84 53001 → 5507 [ACK] Seq=94 Ack=249 Win=2619
> Frame 19: 191 bytes on wire (1528 bits), 151 bytes captured (1208 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 5507, Dst Port: 53001, Seq: 142, Ack: 94, Len: 107
    Source Port: 5507
    Destination Port: 53001
    [Stream index: 1]
    [TCP Segment Len: 107]
    Sequence number: 142    (relative sequence number)
    Sequence number (raw): 2615576482
    [Next sequence number: 249    (relative sequence number)]
    Acknowledgment number: 94    (relative ack number)
    Acknowledgment number (raw): 226764169
    0101 .... = Header Length: 20 bytes (5)
0010  7f 00 00 01 7f 00 00 01  15 83 cf 09 9b e6 87 a2   ........ ........
0020  0d 84 25 89 50 18 27 f9  cd 14 00 00 32 30 32 31   ..%.P.'. ....2021
0030  2f 30 31 2f 31 33 20 32  31 3a 33 38 3a 35 31 0a   /01/13 2 1:38:51.
```

相关的服务器的处理代码片段：

```
void clientListResponse(int index, SOCKET client)
{
    int count = 0;
    char tempBuffer[MAXBYTE] = {0};
    memset(messageBuffer, 0, MAXWORD);
    int check;
    for (check = 0; check < LIST_SIZE; check++)
    {
        if (clientList[check].state)
        {
            count++;
            sprintf(tempBuffer, "[%d] 客户端 -- 来自%s:%d\n", check,
                    inet_ntoa(clientList[check].ip), clientList[check].port);
            strcat(messageBuffer, tempBuffer);
        }
    }
    sprintf(tempBuffer, "%d个客户端与服务器相连，当前客户端编号：%d \n", count, index);
    strcat(messageBuffer, tempBuffer);
    packData(PACKAGE_TYPE_RESPONSE, ACTION_TYPE_CLIENT_LIST, client);
    while (handleState[index] == BLOCK)
        handleState[index] = RUN;
}
```

- 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：

```
[Connected] 请输入操作编号
6
请输入目标的ID
0
输入发送给[0]号的信息，结束请按两次回车
hello

[Receive]
对方客户端编号：0
发送时间：2021/01/13 21:39:49
hello

[Connected] 请输入操作编号

[Connected] 请输入操作编号
发送成功
```

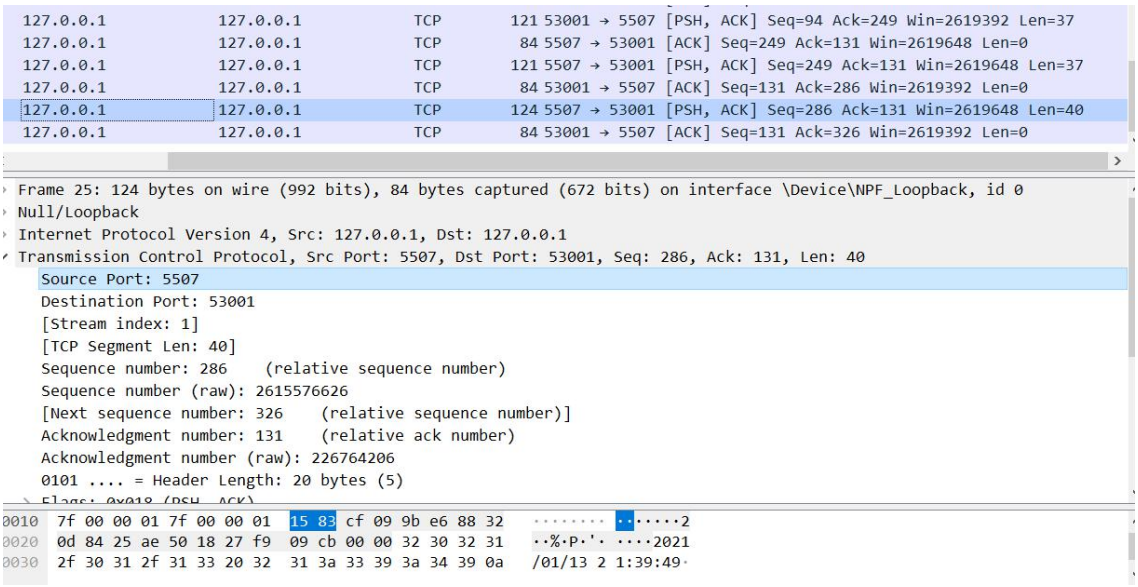
服务器：

```
[Request] 0发消息给0号客户端
[Point] 向0号客户端发送消息：
hello
```

接收消息的客户端：

```
[Receive]
对方客户端编号：0
发送时间：2021/01/13 21:39:49
hello
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：



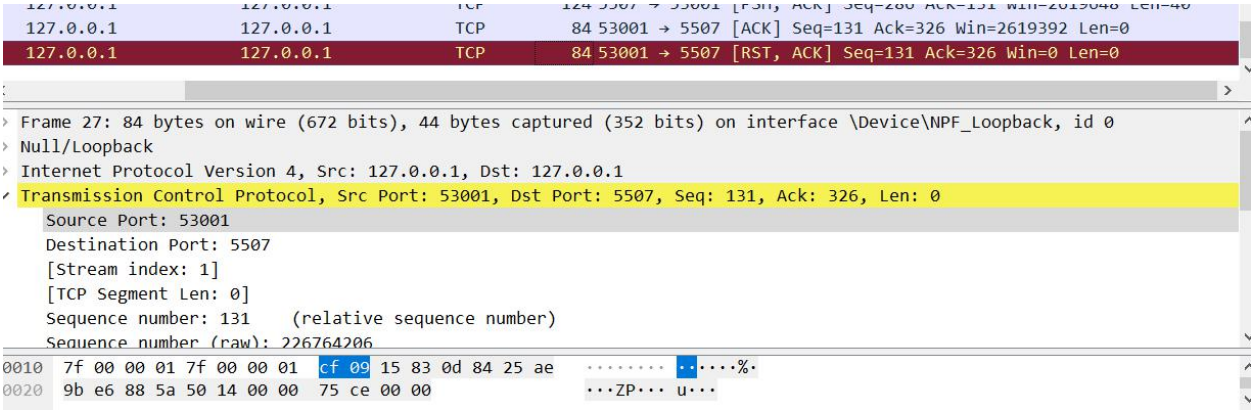
相关的服务器的处理代码片段：

```
void sendMessage(int index, SOCKET client, struct data *unpacked)
{
    int target = unpacked->sourceID;
    printf("[Request] %d发消息给%d号客户端\n", index, target);
    memset(messageBuffer, 0, MAXWORD);
    if (target < 0 || target >= LIST_SIZE)
    {
        sprintf(messageBuffer, "无效的客户端编号\n");
        printf("无效的客户端编号\n");
    }
    else if (clientList[target].state == STOP)
    {
        sprintf(messageBuffer, "客户端未连接至服务器\n");
        printf("客户端未连接至服务器\n");
    }
    else
    {
        strcpy(messageBuffer, unpacked->body);
        id = index;
        printf("[Point] 向%d号客户端发送消息：\n%s", target, unpacked->body);
        packData(PACKAGE_TYPE_ORDER, ACTION_TYPE_SEND_MESSAGE, clientList[target].clientSock);
        sprintf(messageBuffer, "发送成功\n");
        printf("[Hello] 发送成功\n");
    }
    packData(PACKAGE_TYPE_RESPONSE, ACTION_TYPE_SEND_MESSAGE, client);
    while (handleState[index] == BLOCK)
        handleState[index] = RUN;
}
```

相关的客户端（发送和接收消息）处理代码片段：

```
void sendMessage(){
    fflush(stdin);
    printf("输入发送给[%d]号的信息，结束请按两次回车\n",id);
    char ch;
    int index = 0;
    enum bool over = false;
    while(true){
        ch = (char)getchar();
        if (ch == '\n'){
            if (over) break;
            else over = true;
        } else over = false;
        messageBuffer[index++] = ch;
    }
    messageBuffer[index] = '\0';
    packData(PACKAGE_TYPE_GET,ACTION_TYPE_SEND_MESSAGE);
}
```

- 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。



客户端发出了 TCP 释放请求，服务端的 TCP 连接状态在较长时间内不发生变化，始终为 established。

- 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？

```
[Disconnected] 请输入操作编号
1
请输入目标IP(127.0.0.1)与端口(5507)
127.0.0.1 5507
[Hello] 已连接至服务器

[Connected] 请输入操作编号
5
[0] 客户端 -- 来自127.0.0.1:53001
[1] 客户端 -- 来自127.0.0.1:49746
2个客户端与服务器相连，当前客户端编号：1

[Connected] 请输入操作编号
6
请输入目标的ID
0
输入发送给[0]号的信息，结束请按两次回车
hi

发送成功
```

- 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。

```
case ACTION_TYPE_TIME:
    for (int i = 0; i < 100; i++)
    {
        messageBuffer[0] = '\0';
        printf("[Response] 响应%d号客户端的时间请求\n", index);
        packData(PACKAGE_TYPE_RESPONSE, ACTION_TYPE_TIME, client);
    }
    break;
```

127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=2969 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3000 Win=2616576 Len=0
127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=3000 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3031 Win=2616576 Len=0
127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=3031 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3062 Win=2616576 Len=0
127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=3062 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3093 Win=2616576 Len=0
127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=3093 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3124 Win=2616576 Len=0
127.0.0.1	127.0.0.1	TCP	115 5507 → 50127 [PSH, ACK] Seq=3124 Ack=32 Win=2619648 Len=31
127.0.0.1	127.0.0.1	TCP	84 50127 → 5507 [ACK] Seq=32 Ack=3155 Win=2616576 Len=0

服务器正确处理了 100 次请求

客户端也收到了 100 个响应

wireshark 检测到了来自服务器的 201 个 TCP 数据包

服务器：

```
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
```

客户端：

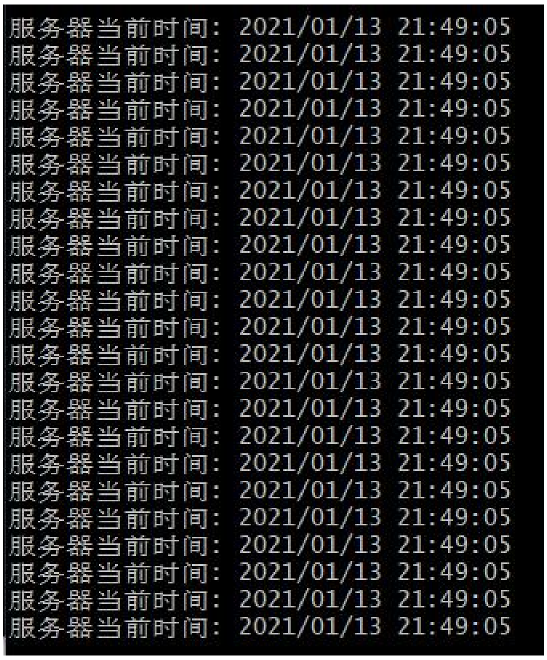
```
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
```

- 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

服务器：

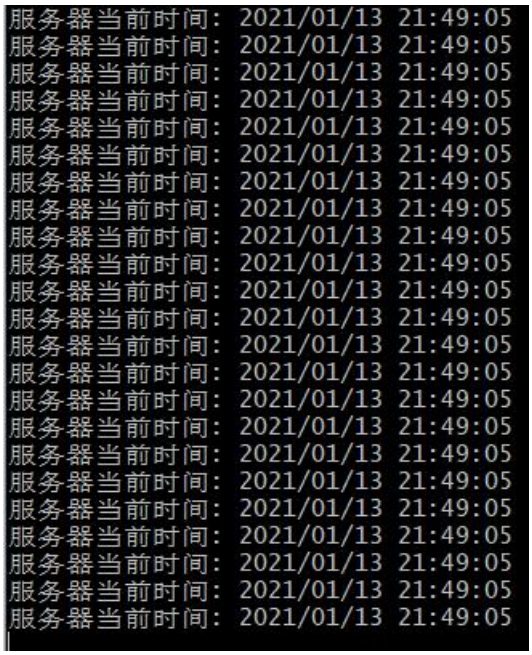
```
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
[Response] 响应0号客户端的时间请求
```

客户端 0：



```
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
服务器当前时间: 2021/01/13 21:49:05
```

客户端 1：



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？

  （1）不需要调用 bind 操作

  （2）由系统内核产生源端口

  （3）每一次调用 connect 时客户端的端口理论上都有可能变化，因为是系统分配的空闲端口。

- 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？

  可以连接成功

- 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？

  不完全一致

- 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？

通过 sockfd 可以区分是哪个客户端。在客户端发送信息的 send 函数中，会有一项 sockfd 字段，该字段唯一确定地标记了 socket 连接。

- 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 的连接状态为 TIME_WAIT，保持了 1 分钟左右。

- 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？

  （1）由于没有检测机制，服务器 TCP 连接状态不会变化。

  （2）可以在使用“心跳包”的方式验证连接是否断开，当客户端断开超过一定时间之后，服务器就会断开连接，TCP 连接会不复存在。

七、 讨论、心得

socket 编程直接编译会报错 undefined reference to `__imp_xxx'，经过查找是因为在一般编译器中不会直接连接编译 wsock 动态库，需要手动添加库，首先尝试代码中直接引入动态库，#pragma comment(lib,"ws2_32")没有用，之后在编译选项里加入-lwsock32 或者在 makefile 中加入-L"ws2_32"解决了问题。

```
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x118): undefined reference to `__imp__send@16'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x210): undefined reference to `__imp__recv@16'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x42b): undefined reference to `__imp__socket@12'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x48e): undefined reference to `__imp__inet_addr@4'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x4aa): undefined reference to `__imp__htons@4'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x4d2): undefined reference to `__imp__connect@12'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x5eb): undefined reference to `__imp__WSAStartup@8'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x7fb): undefined reference to `__imp__closesocket@4'
C:\Users\saisai\AppData\Local\Temp\ccX1HoVf.o:client.c:(.text+0x805): undefined reference to `__imp__WSACleanup@0'
```

经过本次实验，我对 socket 编程及服务器和客户端是如何进行通信的有了更加深刻的理解，并且结合操作系统中学到的多线程的知识，能够让服务器对应多个客户端，响应多个客户端的命令。