

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： 计算机科学与技术

学 号：

指导教师： 陆魁军

姓名	学号	分工	工作量
徐铭	3210102037	Server 端所有内容	50%
程天乐	3210102097	Client 端所有内容	50%

2023 年 10 月 9 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: 徐铭、程天乐 实验地点: 计算机网络实验室

一、实验目的

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

二、实验内容

根据自定义的协议规范, 使用 Socket 编程接口编写基本的网络应用软件。

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

三、主要仪器设备

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

四、操作方法与实验步骤

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

- ✧ 调用 `send()`，发送一个 `hello` 消息给客户端（可选）
- ✧ 循环调用 `receive()`，如果收到了一个完整的请求数据包，根据请求类型做相应的动作：
 1. 请求类型为获取时间：调用 `time()` 获取本地时间，然后将时间数据组装进响应数据包，调用 `send()` 发给客户端
 2. 请求类型为获取名字：将服务器的名字组装进响应数据包，调用 `send()` 发给客户端
 3. 请求类型为获取客户端列表：读取客户端列表数据，将编号、IP 地址、端口等数据组装进响应数据包，调用 `send()` 发给客户端
 4. 请求类型为发送消息：根据编号读取客户端列表数据，如果编号不存在，将错误代码和出错描述信息组装进响应数据包，调用 `send()` 发回源客户端；如果编号存在并且状态是已连接，则将要转发的消息组装进指示数据包。调用 `send()` 发给接收客户端（使用接收客户端的 `socket` 句柄），发送成功后组装转发成功的响应数据包，调用 `send()` 发回源客户端。

d) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 `Socket`，主程序退出。

- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

五、实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

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

前三位代表包的类型	从第4位开始至第一个' '处中间为不定长ID	后面为不定长文本信息
-----------	------------------------	------------

前三位表示请求类型：

000:连接

001:断开连接

010:获取时间

011:获取名字

100:获取所有用户信息

101:发消息

● 描述响应数据包的格式（画图说明），响应类型的定义

前3位代表包的类型	从第4位开始至第一个' '（竖线）处中间为不定长ID	后面为不定长文本信息
-----------	----------------------------	------------

010:返回时间

011:返回名字

100:返回所有用户信息

101:返回发消息是否成功（发送客户端）或发送的消息（接受客户端）

● 描述指示数据包的格式（画图说明），指示类型的定义

前3位代表包的类型	从第4位开始至第一个' '（竖线）处中间为不定长ID	后面为不定长文本信息
-----------	----------------------------	------------

000:连接信息

● 客户端初始运行后显示的菜单选项

```
1) 连接功能
q) 退出功能
请选择功能: |
```

在连接服务器后菜单选项会变为：

```
1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能:
```

● 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(true){
    printChoices(connected);
    std::cin >> opt;
    if((!connected && opt != '1' && opt != 'q')){
        std::cout << "没有与输入对应的功能，请输入正确的指令。\\n" << std::endl;
        continue;
    }

    switch(opt){
        case '1': //连接功能
            if(connected){
                printf("已经连接服务端。\\n\\n");
                break;
            }
            std::cout << "请输入服务器IP: ";
            std::cin >> ip;
            std::cout << "请输入服务器端口: ";
            std::cin >> port;
            serverAddr.sin_port = htons(port);
            serverAddr.sin_addr.s_addr = inet_addr(ip.c_str());
            if (connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
                std::cerr << "Error connecting to server" << std::endl;
                close(clientSocket);
                return -1;
            }
            connected = true;

            res = pthread_create(&receiver, nullptr, receive, &clientSocket);
            if (res != 0)
            {
                printf("ERROR:\\tCreate pthread failed; Return code: %d\\n", res);
                return 0;
            }
            puts("");
    }
}
```

```

        mark_connected();
        break;
    case '2':    //断开功能
        beClosed();
        connected = false;
        std::cout << std::endl;
        break;
    case '3':    //获取时间功能
        getTime();
        break;
    case '4':    //获取名字功能
        getName();
        break;
    case '5':    //获取客户端列表功能
        getList();
        break;
    case '6':    //发送消息功能
        sendMessage();
        break;
    case 'q':
        if(connected) beClosed();
        std::cout << "拜拜~" << std::endl;
        return 0;
    default: std::cout << "没有与输入对应的功能, 请输入正确的指令。\\n" << std::endl;
}
}
}

```

根据用户选择的功能调用对应的功能函数，直到用户输入 q 退出为止。

- 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）


```

void *receive(void *args){
    int clientSocket2 = *reinterpret_cast<int*>(args);
    long res;

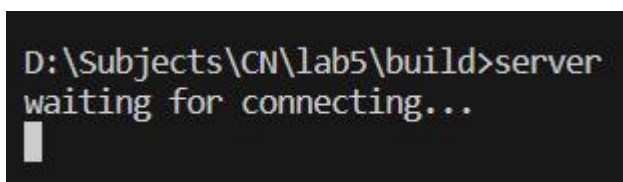
    while(true){
        char buffer[MAXLENGTH] = {0};
        res = recv(clientSocket2, buffer, MAXLENGTH, 0);
        if (res <= 0) break;
        packet receive = analyzePacket(buffer);

        if(receive.getType() == 2) {
            time_t timestamp = std::stoi(receive.getText());
            char time[128]= {0};
            strftime(time, 64, "%Y-%m-%d %H:%M:%S", localtime(&timestamp));
            printf("当前服务器时间为: %s\n\n", time);
            alterLock(false);
        }
        else if(receive.getType() == 3) {
            printf("客户端名称为:\n%s\n\n", receive.getText().c_str());
            alterLock(false);
        }
        else if(receive.getType() == 4) {
            printf("客户端列表为:\n%s\n", receive.getText().c_str());
            alterLock(false);
        }
        else if(receive.getType() == 5) {
            if(waitingStatus){
                std::cout << receive.getText() << "\n" << std::endl;
                alterLock(false);
            }else{
                std::cout << "\n\n收到来自客户端的消息:\n" << receive.getText() << "\n" << std::endl;
                printChoices(true);
            }
        }
    }
}

```

将接收到的字符串信息解析为定义的包类型，然后根据包的内容，输出对应的信息。

- 服务器初始运行后显示的界面



```

D:\Subjects\CN\lab5\build>server
waiting for connecting...

```

- 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）


```
// 循环接收数据
while(1) {
    printf( "waiting for connecting...\n" );
    // prepare the client address/IP/PORT
    struct sockaddr_in clientAddr;
    int clientAddrLen = sizeof( clientAddr );
    SOCKET client_sock = accept( server_sock, (struct sockaddr *)&clientAddr, &clientAddrLen );
    if( client_sock == INVALID_SOCKET ) {
        printf( "accept error!\n" );
        continue;
    }
    clientINFO cliinfo;
    cliinfo.clisock = client_sock;
    cliinfo.cliAddr = clientAddr;
    clients.push_back( cliinfo );
    printf( "receive a connect: %s\n", inet_ntoa(clientAddr.sin_addr) );
    thread ithread = thread( thread_handle, cliinfo );
    ithread.detach();
}
```

如上图所示，主进程开启后，在一个死循环内每接受到一个 client 连接就创建一个子线程去处理这个连接，并在主进程中通过线程池去管理这些子线程；

- 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）

```
void thread_handle ( clientINFO client )
{
    // send a "Hello"
    // send( client.clisock, sendData.data(), sendData.length(), 0 );
    // 准备接收各种信息
    char rev[2048];
    int Flag = 1; //判断是否断开连接
    while( Flag )
    {
        if( recv( client.clisock, rev, sizeof(rev), 0 ) < 0 ) //接收失败
            printf( "accept failed!\n" );
        string revv(rev);
        packet receive = analyzePacket(revv);
        mtx.lock();
        switch ( (int)receive.getType() )
        {
            case 0:{
                printf( "client %d has connected\n", client.clisock );
                break;
            }
            //disconnect
            case 1:{
                for( int i = 0; i < clients.size(); i++ )
                {
                    if( clients[i].clisock == client.clisock ) {
                        clients.erase( clients.begin() + i );
                        break;
                    }
                }
                // 关闭该链接
                Flag = 0;
                printf( "Client Closed!\n" );
                break;
            }
        }
    }
}
```

```

//get time
case 2:{
    time_t seconds;
    time( &seconds );
    packet mes( 2, client.clisock, to_string(seconds) );
    if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
        printf( "Time error!\n" );
    }
    printf( "Time sent success!\n" );
    break;
}
//get name
case 3:{
    char name[256];
    gethostname( name, sizeof(name) );
    packet mes( 3, client.clisock, name );
    if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
        printf( "Name error!\n" );
    }
    printf( "Name sent success!\n" );
    break;
}
}

```

子线程处理每个连接请求的过程也是一个循环,只有当 client 和服务端处于连接状态时会持续循环,否则直接结束;

```

//get cli_list
case 4:{
    string lists;
    for( int i = 0; i < clients.size(); i++ ) {
        int sock = clients[i].clisock;
        int port = clients[i].cliAddr.sin_port;
        string addr = inet_ntoa( clients[i].cliAddr.sin_addr );
        string ans = to_string(sock) + ", " + to_string(port) + ", " + addr + ";\n";
        lists += ans;
    }
    packet mes( 4, client.clisock, lists );
    if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
        printf( "client lists error!\n" );
    }
    printf( "Client lists sent success!\n" );
    break;
}
}

```

如果客户端请求时间、名字、用户列表,这里将会进行一次信息的组装,用上面规定的协议组成相应数据包,将内容通过 send 发回。而如果是向其他用户发送消息,则会解析出编号、IP 和发送内容,找到已经在连接里的这个客户,并向他发送一条消息;

```

        case 5:{
            int flag = 0 ;
            for( int i = 0; i < clients.size(); i++ ) {
                if( clients[i].clisock == receive.getId() ) {
                    if( client.clisock == receive.getId() ) {
                        packet mes3( 5, client.clisock, "\n你成功给自己发了一条信息:" + receive.getText() );
                        if( send( client.clisock, pac2str(mes3).data(), pac2str(mes3).length(), 0 ) < 0 ) {
                            printf( "Sending error!\n" );
                        }
                    } else {
                        packet mes( 5, receive.getId(), receive.getText() );
                        if( send( clients[i].clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
                            printf( "Sending error!\n" );
                        }
                        packet mes2( 5, client.clisock, "Sending success!");
                        if( send( client.clisock, pac2str(mes2).data(), pac2str(mes2).length(), 0 ) < 0 ) {
                            printf( "Sending error!\n" );
                        }
                    }
                    printf( "Sending success!\n" ); // need ip and port
                    flag = 1;
                    break;
                }
            }
            if( !flag ){
                packet mes( 5, client.clisock, "No this client!" );
                if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
                    printf( "Sending error!\n" );
                }
                printf( "Sending success!\nNOT found the destination!\n" );
            }
        }
        default:
            break;
    }
}
memset( &rev, 0, sizeof(rev) );//清空rec, 准备接受下一次消息
mtx.unlock();

```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

```

1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：

```

```

D:\Subjects\CN\lab5\build>server
waiting for connecting...
receive a connect: 10.192.52.126
waiting for connecting...
client 272 has connected

```

Wireshark 抓取的数据包截图：

Wireshark · 分组 93 · WLAN

[Window size scaling factor: 256]
Checksum: 0x628b [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
> [Timestamps]
v [SEQ/ACK analysis]
 [iRTT: 0.016053000 seconds]
 [Bytes in flight: 5]
 [Bytes sent since last PSH flag: 5]
TCP payload (5 bytes)
v Data (5 bytes)
 Data: 303030307c
 [Length: 5]

0000	e8 84 a5 bd 6f fa 74 3a 20 b9 e8 02 08 00 45 00	...o.t:E.
0010	00 2d 7b c5 40 00 7d 06 06 cd 0a c0 34 7e 0a a2	--{.@.}-.4~..
0020	31 59 28 24 07 f5 23 90 9b 90 90 11 74 56 50 18	1Y(\$..#..tVP.
0030	02 01 62 8b 00 00 30 30 30 30 7c	..b...00 00

- 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) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：3
当前服务器时间为：2023-10-12 13:31:22

```

```

receive a connect: 10.192.52.126
waiting for connecting...
client 212 has connected
Time sent success!

```

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）：

Wireshark · 分组 76 · WLAN

0101 = Header Length: 20 bytes (5)

▼ Flags: 0x018 (PSH, ACK)

000. = Reserved: Not set

...0 = Accurate ECN: Not set

... 0... = Congestion Window Reduced: Not set

... .0.. = ECN-Echo: Not set

... ..0. = Urgent: Not set

... ...1 = Acknowledgment: Set

... 1... = Push: Set

... ..0.. = Reset: Not set

...0. = Syn: Not set

... ..0.. = Fin: Not set

[TCP Flags:AP...]

Window: 513

[Calculated window size: 513]

[Window size scaling factor: -1 (unknown)]

Checksum: 0x7b69 [unverified]

[Checksum Status: Unverified]

Urgent Pointer: 0

▼ [Timestamps]

[Time since first frame in this TCP stream: 0.000422000 seconds]

[Time since previous frame in this TCP stream: 0.000422000 seconds]

▼ [SEQ/ACK analysis]

[This is an ACK to the segment in frame: 75]

[The RTT to ACK the segment was: 0.000422000 seconds]

[Bytes in flight: 22]

[Bytes sent since last PSH flag: 22]

TCP payload (22 bytes)

▼ Data (22 bytes)

Data: 30313031313031303130307c31363937303838363832

[Length: 22]

0000	74 3a 20 b9 e8 02 e8 84 a5 bd 6f fa 08 00 45 00	t:o...E.
0010	00 3e 9e c3 40 00 00 06 00 00 0a a2 31 59 0a c0	->..@.. ...1Y..
0020	34 7e 07 f5 28 24 90 11 74 56 23 90 9b 9a 50 18	4~..(\$...tv#...P.
0030	02 01 7b 69 00 00 30 31 30 31 31 30 31 30 31 30	..{i...01 01101010
0040	30 7c 31 36 39 37 30 38 38 36 38 32	0 169708 8682

- 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。

```

1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：4
客户端名称为：
LAPTOP-JGT50I8I

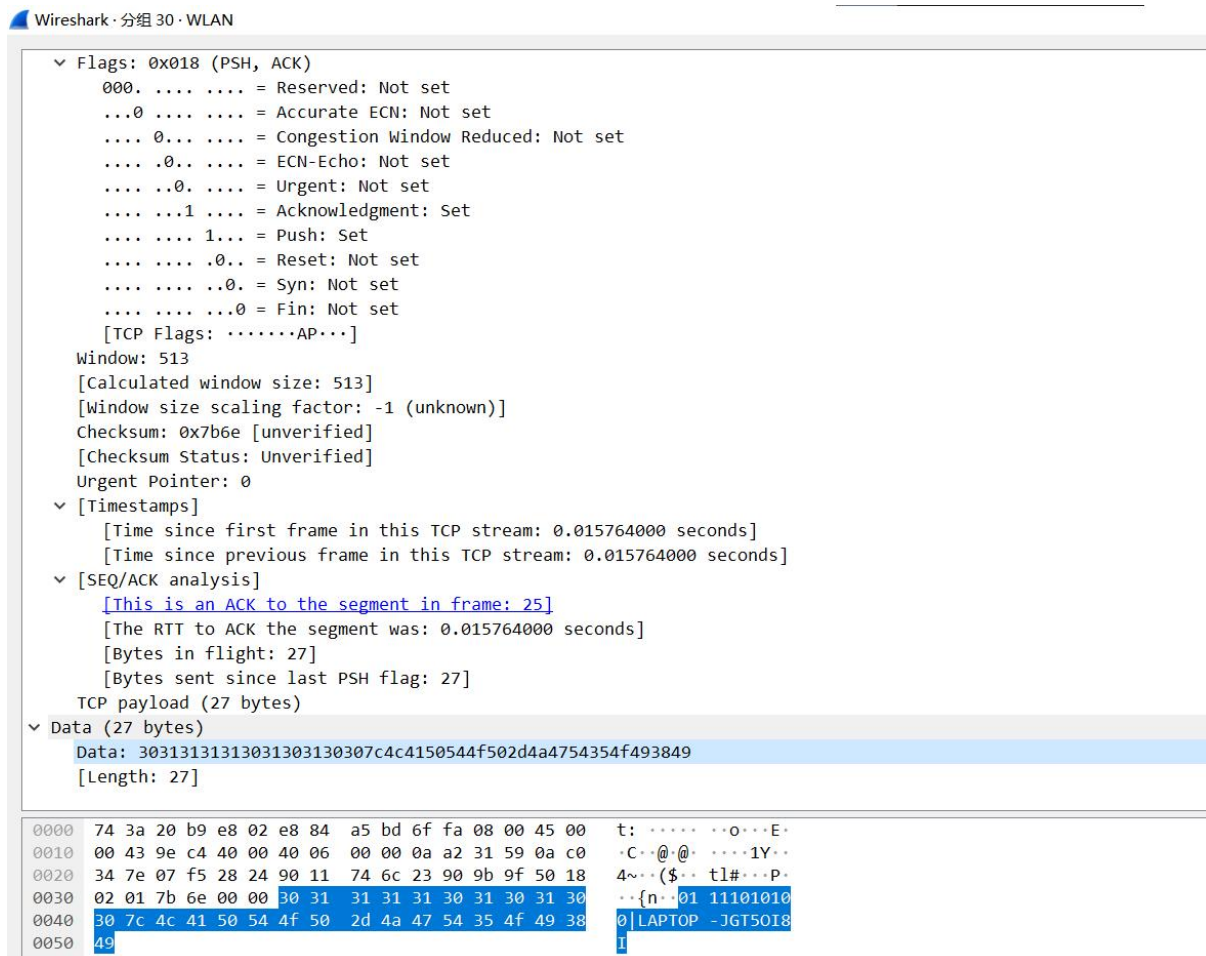
```

```

receive a connect: 10.192.52.126
waiting for connecting...
client 212 has connected
Time sent success!
Name sent success!

```

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：



- 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。

```
lyb@lyb-virtual-m: lyb@lyb-virtual-mach

6) 发送消息功能
q) 退出功能
请选择功能：3
当前服务器时间为：2023-10-12 13:31:22

1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：4
客户端名称为：
LAPTOP-JGT50I8I

请选择功能：1
请输入服务器IP：10.162.49.89
请输入服务器端口：2037

1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：5
客户端列表为：
212, 9256, 10.192.52.126;
640, 55337, 10.192.52.126;
```

```
receive a connect: 10.192.52.126
waiting for connecting...
client 212 has connected
Time sent success!
Name sent success!
receive a connect: 10.192.52.126
waiting for connecting...
client 640 has connected
Client lists sent success!
[]
```

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）：

```
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
.... ...1 .... = Acknowledgment: Set
.... .... 1... = Push: Set
.... .... .0.. = Reset: Not set
.... .... ..0. = Syn: Not set
.... .... ...0 = Fin: Not set
[TCP Flags: .....AP...]
Window: 513
[Calculated window size: 131328]
[Window size scaling factor: 256]
Checksum: 0x7b96 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
▼ [Timestamps]
  [Time since first frame in this TCP stream: 5.394852000 seconds]
  [Time since previous frame in this TCP stream: 0.000254000 seconds]
▼ [SEQ/ACK analysis]
  [This is an ACK to the segment in frame: 159]
  [The RTT to ACK the segment was: 0.000254000 seconds]
  [iRTT: 0.015128000 seconds]
  [Bytes in flight: 67]
  [Bytes sent since last PSH flag: 67]
  TCP payload (67 bytes)
▼ Data (67 bytes)
  Data: 313030313031303030303030307c3231322c20393235362c2031302e3139322e35322e31...
  [Length: 67]

0000  74 3a 20 b9 e8 02 e8 84 a5 bd 6f fa 08 00 45 00  t: .....o...E
0010  00 6b 9e c7 40 00 40 06 00 00 0a a2 31 59 0a c0  .k..@..  ....1Y..
0020  34 7e 07 f5 29 d8 1c 8e be 39 82 60 58 33 50 18  4~...)...`9~X3P
0030  02 01 7b 96 00 00 31 30 30 31 30 31 30 30 30 30  ..{...10 01010000
0040  30 30 30 7c 32 31 32 2c 20 39 32 35 36 2c 20 31  000|212, 9256, 1
0050  30 2e 31 39 32 2e 35 32 2e 31 32 36 3b 0a 36 34  0.192.52 .126;.64
0060  30 2c 20 35 35 33 33 37 2c 20 31 30 2e 31 39 32  0, 55337 , 10.192
0070  2e 35 32 2e 31 32 36 3b 0a                          .52.126; .
```

相关的服务器的处理代码片段：


```

//get cli_list
case 4:{
    string lists;
    for( int i = 0; i < clients.size(); i++ ) {
        int sock = clients[i].clisock;
        int port = clients[i].cliAddr.sin_port;
        string addr = inet_ntoa( clients[i].cliAddr.sin_addr );
        string ans = to_string(sock) + ", " + to_string(port) + ", " + addr + ";\n";
        lists += ans;
    }
    packet mes( 4, client.clisock, lists );
    if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
        printf( "Client lists error!\n" );
    }
    printf( "Client lists sent success!\n" );
    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：

```

1) 连接功能
2) 断开功能
3) 获取时间功能
4) 获取名字功能
5) 获取客户端列表功能
6) 发送消息功能
q) 退出功能
请选择功能：6
请输入目标客户端的ID：640
请输入要传输的内容：123456
Sending success!

```

服务器：

```

receive a connect: 10.192.52.126
waiting for connecting...
client 640 has connected
Client lists sent success!
Sending success!

```

接收消息的客户端：

```

收到来自客户端的消息：
123456

```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

Data (28 bytes)	
Data: 31303131313031303130307c53656e64696e67207375636365737321	
[Length: 28]	
0000	74 3a 20 b9 e8 02 e8 84 a5 bd 6f fa 08 00 45 00 t:o...E.
0010	00 44 9e c9 40 00 40 06 00 00 0a a2 31 59 0a c0 .D..@. @.1Y..
0020	34 7e 07 f5 28 24 90 11 74 87 23 90 9b b3 50 18 4~..(\$.. t.#...P.
0030	02 01 7b 6f 00 00 31 30 31 31 31 30 31 30 31 30 ..{o...10 11101010
0040	30 7c 53 65 6e 64 69 6e 67 20 73 75 63 63 65 73 0 Sendin g succes
0050	73 21 s!
Data (20 bytes)	
Data: 3130313130313030303030307c313233343536	
[Length: 20]	
0000	74 3a 20 b9 e8 02 e8 84 a5 bd 6f fa 08 00 45 00 t:o...E.
0010	00 3c 9e c8 40 00 40 06 00 00 0a a2 31 59 0a c0 .<..@. @.1Y..
0020	34 7e 07 f5 29 d8 1c 8e be 7c 82 60 58 33 50 18 4~..).... `X3P.
0030	02 01 7b 67 00 00 31 30 31 31 30 31 30 30 30 30 ..{g...10 11010000
0040	30 30 30 7c 31 32 33 34 35 36 000 1234 56

相关的服务器的处理代码片段：

```
case 5:{
    int flag = 0 ;
    for( int i = 0; i < clients.size(); i++ ) {
        if( clients[i].clisock == receive.getId() ) {
            if( client.clisock == receive.getId() ) {
                packet mes3( 5, client.clisock, "\n你成功给自己发了一条信息:" + receive.getText() );
                if( send( client.clisock, pac2str(mes3).data(), pac2str(mes3).length(), 0 ) < 0 ) {
                    printf( "Sending error!\n" );
                }
            } else {
                packet mes( 5, receive.getId(), receive.getText() );
                if( send( clients[i].clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
                    printf( "Sending error!\n" );
                }
                packet mes2( 5, client.clisock, "Sending success!");
                if( send( client.clisock, pac2str(mes2).data(), pac2str(mes2).length(), 0 ) < 0 ) {
                    printf( "Sending error!\n" );
                }
            }
            printf( "Sending success!\n" ); // need ip and port
            flag = 1;
            break;
        }
    }
    if( !flag ){
        packet mes( 5, client.clisock, "No this client!" );
        if( send( client.clisock, pac2str(mes).data(), pac2str(mes).length(), 0 ) < 0 ) {
            printf( "Sending error!\n" );
        }
        printf( "Sending success!\nNOT found the destination!\n" );
    }
}
```

相关的客户端（发送和接收消息）处理代码片段：

```

if(waitingStatus){
    std::cout << receive.getText() << "\n" << std::endl;
    alterLock(false);
}else{
    std::cout << "\n\n收到来自客户端的消息:\n" << receive.getText() << "\n" << std::endl;
    printChoices(true);
}

```

通过 waitingStatus 来判断是发送端还是接受端：如果 waitingStatus 为 true，则说明是发送端，输出得到的包中的反馈信息；如果 waitingStatus 为 false，则说明是接收端，输出收到的信息。

- 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（10 分钟以上）是否发生变化。

答：从 wireshark 抓包观察得到，客户端并没有释放 TCP 连接释放的消息，但一段时间之后，服务器的连接断开

- 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？

答：从客户端列表中发现之前异常退出的连接并不存在，而且无法给其发消息，因为服务器已经断掉了与其的连接

- 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。

当前服务器时间为: 2023-10-12 14:10:29

87
当前服务器时间为: 2023-10-12 14:10:29

88
当前服务器时间为: 2023-10-12 14:10:29

89
当前服务器时间为: 2023-10-12 14:10:29

90
当前服务器时间为: 2023-10-12 14:10:29

91
当前服务器时间为: 2023-10-12 14:10:29

92
当前服务器时间为: 2023-10-12 14:10:30

93
当前服务器时间为: 2023-10-12 14:10:30

94
当前服务器时间为: 2023-10-12 14:10:30

95
当前服务器时间为: 2023-10-12 14:10:30

96
当前服务器时间为: 2023-10-12 14:10:30

97
当前服务器时间为: 2023-10-12 14:10:30

98
当前服务器时间为: 2023-10-12 14:10:30

99

Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!
Time sent success!

48	4.772215	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=1 Ack=1 Win=513 Len=5
49	4.772549	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=1 Ack=6 Win=513 Len=24
50	4.782427	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=6 Ack=25 Win=513 Len=5
51	4.782729	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=25 Ack=11 Win=513 Len=24
52	4.803069	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=11 Ack=49 Win=513 Len=5
53	4.803405	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=49 Ack=16 Win=513 Len=24
54	4.822452	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=16 Ack=73 Win=513 Len=5
55	4.822787	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=73 Ack=21 Win=513 Len=24
56	4.832420	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=21 Ack=97 Win=512 Len=5
57	4.832773	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=97 Ack=26 Win=513 Len=24
58	4.844831	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=26 Ack=121 Win=512 Len=5
59	4.845181	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=121 Ack=31 Win=513 Len=24
60	4.858673	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=31 Ack=145 Win=512 Len=5
61	4.858920	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=145 Ack=36 Win=513 Len=24
62	4.863242	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=36 Ack=169 Win=512 Len=5
63	4.863459	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=169 Ack=41 Win=513 Len=24
64	4.879102	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=41 Ack=193 Win=512 Len=5
65	4.879428	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=193 Ack=46 Win=513 Len=24
66	4.888738	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=46 Ack=217 Win=512 Len=5
67	4.889008	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=217 Ack=51 Win=513 Len=24
68	4.898725	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=51 Ack=241 Win=512 Len=5
69	4.899068	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=241 Ack=56 Win=512 Len=24
70	4.911484	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=56 Ack=265 Win=512 Len=5
71	4.911801	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=265 Ack=61 Win=513 Len=24
73	4.923790	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=61 Ack=289 Win=512 Len=5
74	4.924149	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=289 Ack=66 Win=513 Len=24
75	4.929165	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=66 Ack=313 Win=512 Len=5
76	4.929514	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=313 Ack=71 Win=512 Len=24
77	4.942281	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=71 Ack=337 Win=511 Len=5
78	4.942544	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=337 Ack=76 Win=512 Len=24
223	5.935767	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2065 Ack=436 Win=511 Len=24
224	5.941943	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=436 Ack=2089 Win=510 Len=5
225	5.942157	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2089 Ack=441 Win=511 Len=24
226	5.959136	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=441 Ack=2113 Win=510 Len=5
227	5.959465	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2113 Ack=446 Win=511 Len=24
228	5.977237	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=446 Ack=2137 Win=510 Len=5
229	5.977504	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2137 Ack=451 Win=511 Len=24
230	5.989696	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=451 Ack=2161 Win=510 Len=5
231	5.989951	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2161 Ack=456 Win=511 Len=24
232	6.013301	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=456 Ack=2185 Win=510 Len=5
233	6.013555	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2185 Ack=461 Win=511 Len=24
234	6.017623	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=461 Ack=2209 Win=510 Len=5
235	6.017801	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2209 Ack=466 Win=511 Len=24
236	6.037009	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=466 Ack=2233 Win=510 Len=5
237	6.037340	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2233 Ack=471 Win=511 Len=24
238	6.055387	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=471 Ack=2257 Win=510 Len=5
239	6.055732	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2257 Ack=476 Win=511 Len=24
240	6.071603	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=476 Ack=2281 Win=510 Len=5
241	6.071953	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2281 Ack=481 Win=511 Len=24
242	6.088780	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=481 Ack=2305 Win=510 Len=5
243	6.089112	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2305 Ack=486 Win=511 Len=24
244	6.099639	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=486 Ack=2329 Win=509 Len=5
245	6.099920	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2329 Ack=491 Win=511 Len=24
246	6.109572	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=491 Ack=2353 Win=509 Len=5
247	6.109878	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2353 Ack=496 Win=511 Len=24
248	6.118353	10.192.52.126	10.162.49.89	TCP	59	11002 → 2037	[PSH, ACK]	Seq=496 Ack=2377 Win=509 Len=5
249	6.118595	10.162.49.89	10.192.52.126	TCP	78	2037 → 11002	[PSH, ACK]	Seq=2377 Ack=501 Win=511 Len=24
250	6.182285	10.192.52.126	10.162.49.89	TCP	56	11002 → 2037	[ACK]	Seq=501 Ack=2401 Win=509 Len=0

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？

不需要。

在客户端建立与服务器的连接时，客户端的操作系统会为该连接分配一个临时的源端口号。

是会发生变化的。

- 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？

答：能够连接成功

- 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？

答：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？

答：如果 IP 地址不同，则通过 IP 地址区分；如果 IP 地址相同，则通过客户端产生的 Socket 描述符或者连接时间的先后加以区分。

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

答：TCP 的连接状态为 TIME_WAIT，维持了大约 70s。

- 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？

答：服务端的连接状态没有变化；可以尝试向客户端发送一个数据包。观察是否有回应或是设定超时时间，当客户端间隔一定时间没有进行操作时，服务端自动断开连接。

七、 讨论、心得

本次实验中客户端部分比较难的部分就是多线程的处理。因为是第一次写涉及多线程的程序，所以写起来有一定的难度。通过不断地查询相关资料以及和同学之间的交流，最终还是解决了这些问题，成功地按要求完成了这次实验。