

# 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:  | 胡若凡                   |
| 学 院:  | 计算机学院                 |
| 系:    | 计算机科学与技术学院            |
| 专 业:  | 计算机科学与技术              |
| 学 号:  | 3200102312            |
| 指导教师: | 张泉方                   |

2022 年 10 月 30 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: 郭伟京 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

使用的请求包为一个 `char[256]` 的数组和一个代表请求类型的 `long` 组成

```
struct Message {
    long type;

    char data[256];
};
```

其中 `type` 对应了以下情况：

```
enum TYPE{ CANCEL = 1, TIME = 2, NAME = 3, CLIENT = 4, SEND = 5 };
```

使用了枚举类，定义了五种请求，分别是取消连接-CANCEL，请求时间-TIME，请求获取名字-NAME，请求给出当前连接的所有客户端信息-CLIENT，请求发送信息给其他的客户-SEND

具体实现时如图：

| 请求类型(type) | 附加信息(data[256])  |
|------------|--|
| CANCEL     | 无  |
| TIME       | 无  |
| NAME       | 无  |
| CLIENT     | 无  |
| SEND       | 额外附加 ip 地址，端口号，和要发送的内容 char ip[20];int port;char content[220];<br>最终会形成的格式是 ip: port: content，通过 “:” 实现了分割 |

```
● username@DESKTOP-GQ5FSI3:/mnt/c/Users/lenovo/Desktop/socketchat$ ./try
Target IP: 10.0.0.0
Target port: 20
Send content: we love you
10.0.0.0:20:we love you
```

上图为 type 为 SEND 类时，content 进行组装的实例。

● 描述响应数据包的格式（画图说明），响应类型的定义

回复的请求包同样为一个结构体类型：

```
struct Message {
    long type;
    char data[256];
};
```

而具体的 data 信息描述如下

| 请求类型(type) | 附加信息(data[256])                               |
|------------|---|
| CANCEL     | 普通字符串回应                                       |
| TIME       | Time 类型，供 Client 端接收更改形式                      |
| NAME       | 普通字符串回应                                       |
| CLIENT     | 客户间用分号隔开：Sock1,port1,IP1;Sock2,port2,IP2..... |
| SEND       | 对于接收的客户端，data 为发送内容，对于发宋客户端，data 为是否成功/错误接收   |

- 描述指示数据包的格式（画图说明），指示类型的定义

```
● username@DESKTOP-GQ5FSI3:/mnt/c/Users/lenovo/Desktop/socketchat$ ./try
Target IP: 10.0.0.0
Target port: 20
Send content: we love you
10.0.0.0:20:we love you
```

解释如请求数据包一样

- 客户端初始运行后显示的菜单选项

```
+-----+
|请给出你的选择|
|1.请求连接到指定的服务端上|
|2.请求断开现在服务端的连接|
|3.请求服务端给出现在的时间|
|4.请求服务端给出当前其机器的名称信息|
|5.请求服务端给出所有已有客户端的信息|
|6.请求服务端发送特定信息给指定客户端|
|0.断开连接并退出现有客户端|
+-----+
```

- 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）

```
void Run()
{
    while (1){
        Start();
        printf("请给出你的选择:");
        int choice;
        scanf("%d",&choice);
        switch (choice)
        {
            case 1:
                ConnectServer();
                sleep(1);
                break;
            case 2:
                CancelConnect();
                break;
            case 3:
                TimeRequest();
                break;
            case 4:
                NameRequest();
                break;
```

```

        case 5:
            ClientRequest();
            break;
        case 6:
            SendRequest();
            break;
        case 0:
            Done();
            break;
        default:
            printf("请输入 0-6 之间的整数操作符,请重新输入");
            break;
    }
}
}

```

如图所示，通过一个循环，不断给出选择表，并读入用户的选择，执行相应的功能函数。

- 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* Thread(void *arg)
{
    //不用子线程接收会出现堵塞的问题
    int sockfd=*((int*)arg); //强制转换，获得 socket 号
    key_t key = ftok("c", 12); //与主线程，保持一致
    int CmsgID = msgget(key, IPC_CREAT | 0666);
    //printf("I'm the Cthread msgid %d\n",CmsgID);
    if (CmsgID < 0)
        printf("[Error] Message queue create fail: %s", strerror(errno));
    //这里发现一定要在 ubuntu 里，在 wsl 会出错
    Message msg;
    //recv 接收的是 server 发来的 message
    recv(sockfd, &msg.data, sizeof(msg.data), 0);
    //printf("%s\n>", msg.data);
    cout<<msg.data<<endl;
    while (1) //一直做，直到主线程要求停下来
    {
        memset(&msg, 0, sizeof(msg));
        if (recv(sockfd, &msg, sizeof(msg), 0) < 0) //接收错误
        {
            printf("[Error] helper recv fail, error: %s\n", strerror(errno));
        }
        if (msg.type == SEND)
    }
}

```

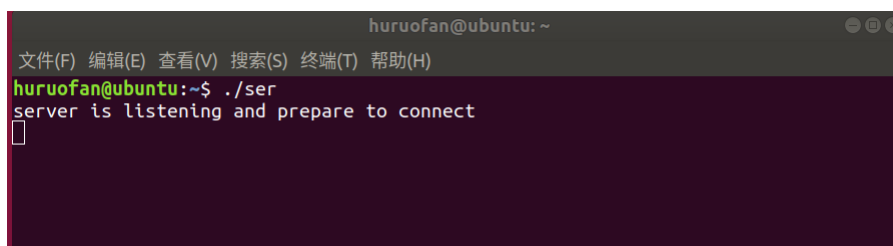
```

{
    printf("\n");
    printf("Receive repost: %s\n", msg.data);
    continue;
}
//printf("from cthread %s\n",msg.data);
//msgsnd 是子线程往主线程通信发送
msgsnd(CmsgID, &msg, 256, 0);
//printf("cthread has done\n");
}
}

```

如图，除了第一个 send 接收的是 hello 的对接信息，后续会一直接收信息，并通过消息队列再回传给主线程

- 服务器初始运行后显示的界面



A terminal window titled 'huruofan@ubuntu: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'huruofan@ubuntu:~\$' and the command './ser' has been entered. The output is 'server is listening and prepare to connect' followed by a cursor.

- 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(1)
{
    struct sockaddr_in cli_addr; //客户端地址
    socklen_t cli_addr_len = sizeof(cli_addr);
    //接受连接请求
    int cli_sock = accept(sock,(struct sockaddr *)&cli_addr,&cli_addr_len);
    struct cli_addr_port cli;
    cli.port = cli_addr.sin_port;
    cli.addr = inet_ntoa(cli_addr.sin_addr);
    cli.sock = cli_sock;
    clients.push_back(cli);
    //声明线程 id
    pthread_t server_thread;
    //创建线程
    pthread_create(&server_thread, NULL, thread_handle, &cli_sock);
}

```

如代码所示，通过一个不断的循环，每次接收一个 client 的连接，就会创建一个子线程，让子



线程去处理这次连接的各种请求。

- 服务器的处理子线程循环关键代码截图（描述总体，省略细节部分）

```
while(Flag)
{
    //接收失败
    if(recv(con_fd,&rec,sizeof(rec),0)<0){
        cout<<"it is faled";
    }
    mtx.lock();
    switch (rec.type)
    {
        //disconnect
        case CANCEL:{
            for(int i=0;i<clients.size();i++)
            {
                if(clients[i].sock == con_fd){
                    clients.erase(clients.begin()+i);
                    break;
                }
            }
            //关闭该链接
            Flag = 0;
            close(con_fd);
            cout<<"this is closed"<<endl;
            break;
        }
        //get time
        case TIME:{
            time_t time1;
            time(&time1);
            Message mes;
            mes.type = TIME;
            sprintf(mes.data, "%ld", time1);
            if(send(con_fd,&mes,sizeof(mes),0)<0){
                cout<<"it is error";
            }
            cout<<"the time is sent successfully"<<endl;
            break;
            //get name
        }
        case NAME:{
            Message mes;
            mes.type = NAME;
```

```

        gethostname(mes.data, sizeof(mes.data));
        if(send(con_fd,&mes,sizeof(mes),0)<0){
            cout<<"it is failed";
        }
        cout<<"the name is sent sucessfully"<<endl;
        break;
        //get cli_list
    }
    case CLIENT:{
        Message mes;
        mes.type = CLIENT;
        strcpy(mes.data,"");
        for(int i=0;i<clients.size();i++){
            int sock = clients[i].sock;
            int port = clients[i].port;
            string addr = clients[i].addr;
            //    cout<<sock<<endl<<port<<endl<<addr<<endl;
            string ans = to_string(sock)+","+to_string(port)+","+addr+"";
            strcat(mes.data, ans.c_str());
        }
        //    strcat(mes.data,0);
        if(send(con_fd,&mes,sizeof(mes),0)<0){
            cout<<"send error";
        }
        //cout<<mes.data<<endl;
        cout<< "the client is sent successfully"<<endl;
        break;
        //send message
    }
    case SEND:{
        //获得端口号和 ip 地址
        //data 顺序为 IP port 内容 切割
        string content = string(rec.data);
        string addr = content.substr(0,content.find(","));
        content = content.substr(content.find(",")+1);
        int port = atoi(content.substr(0,content.find(",")).c_str());
        content = content.substr(content.find(",")+1);
        string ans = content;
        //cout<<addr<<endl<<port<<endl;
        //遍历 cli_list 找到发送客户端
        int flag = 0 ;
        for(int i=0;i<clients.size();i++)
        {
            if(clients[i].addr == addr && clients[i].port == port)

```

```

        {
            Message mes;
            mes.type = SEND;
            strcpy(mes.data, content.c_str());
            cout<<clients[i].sock<<endl;
            if(send(clients[i].sock,&mes,sizeof(mes),0)<0)
            {
                cout<<"it is error to send";
            }
            cout<<"it is sent to ip :"+ addr
+"port :"+to_string(port)<<endl;
            flag = 1;
            break;
        }
    }
    if(!flag){
        Message mes;
        mes.type = SEND;
        strcpy(mes.data,"no this port");
        cout<<"not found the destination"<<endl;
        send(con_fd,&mes,sizeof(mes),0);
    }
}
default:
    break;
}
memset(&rec,0,sizeof(rec));//清空 rec, 准备接受下一次消息
mtx.unlock();
}
return NULL;
}
}

```

如果是请求时间、名字、用户名等，将会进行一次信息的组装，用规定的协议将内容通过 send 发回。而如果是向其他用户发送消息，则会解析出端口号、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。

- 客户端选择连接功能时，客户端和服务端显示内容截图。

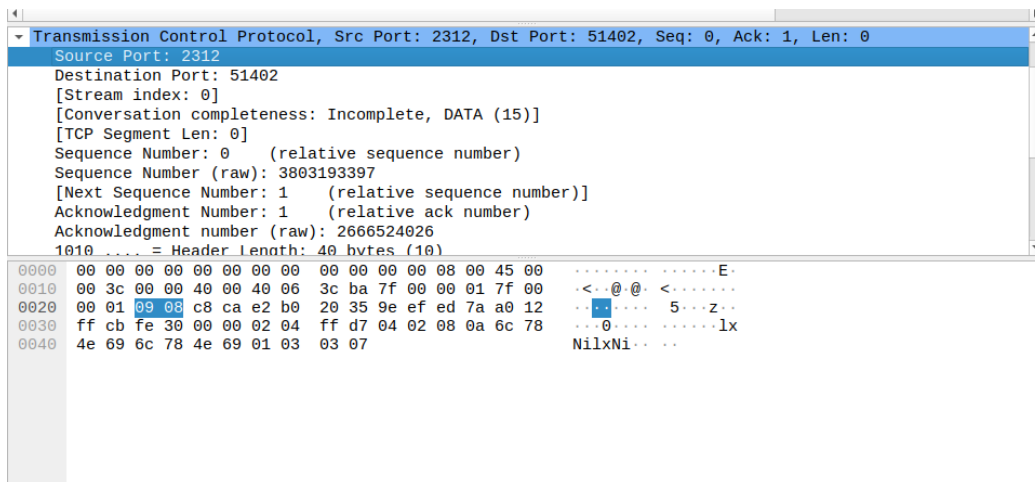
```

264+-----+
|请给出你的选择|
|1.请求连接到指定的服务端上|
|2.请求断开现在服务端的连接|
|3.请求服务端给出现在的时间|
|4.请求服务端给出当前其机器的名称信息|
|5.请求服务端给出所有已有客户端的信息|
|6.请求服务端发送特定信息给指定客户端|
|0.断开连接并退出现有客户端|
+-----+
请给出你的选择:1
[Info] Connected!

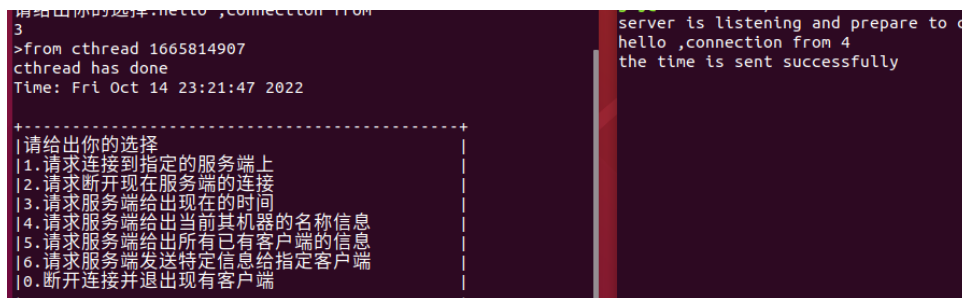
gwj@ubuntu:~$ ./server
server is listening and prepare to connect
hello ,connection from 4

```

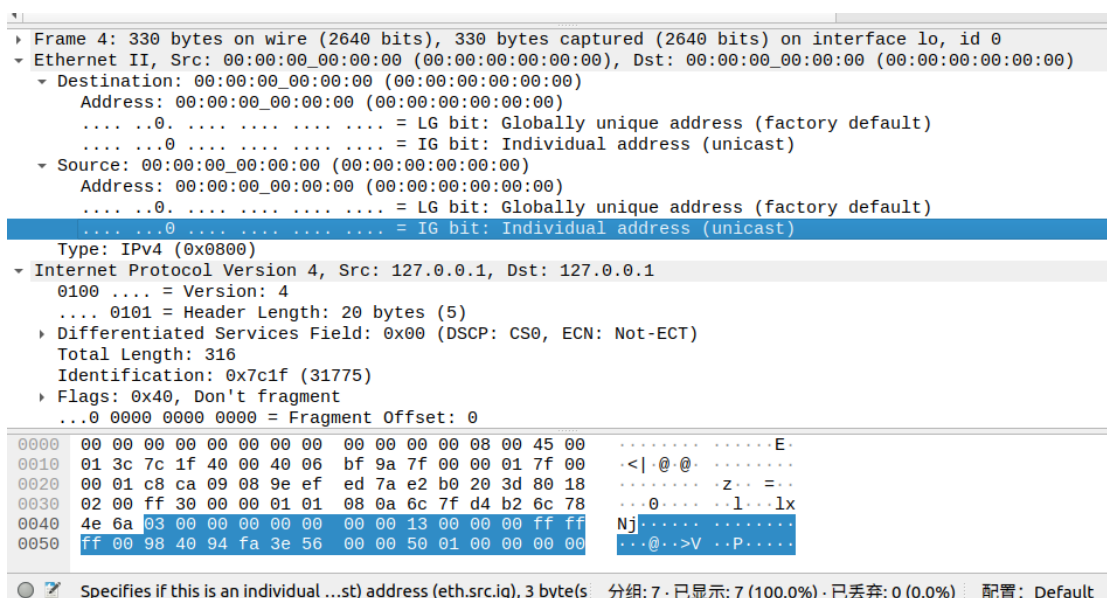
Wireshark 抓取的数据包截图：



- 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。



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。

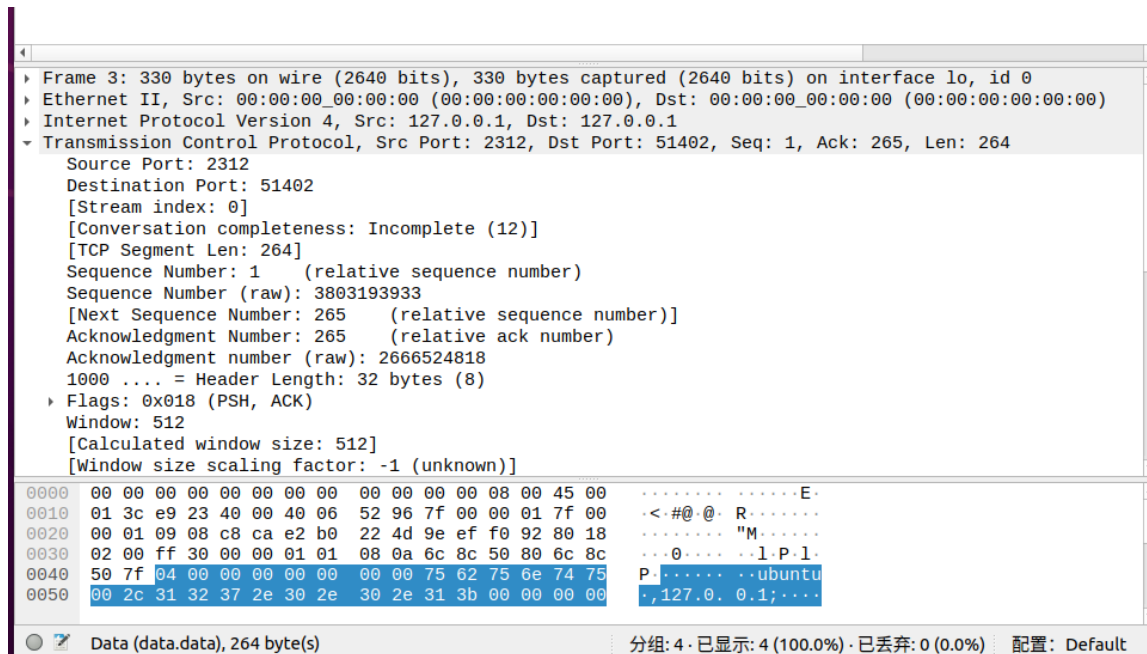
```

3
>from cthread 1665814907
cthread has done
Time: Fri Oct 14 23:21:47 2022

+-----+
|请给出你的选择|
|1.请求连接到指定的服务端上|
|2.请求断开现在服务端的连接|
|3.请求服务端给出现在的时间|
|4.请求服务端给出当前其机器的名称信息|
|5.请求服务端给出所有已有客户端的信息|
|6.请求服务端发送特定信息给指定客户端|
|0.断开连接并退出现有客户端|
+-----+

server is listening and prepare to co
hello ,connection from 4
the time is sent successfully
  
```

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）：



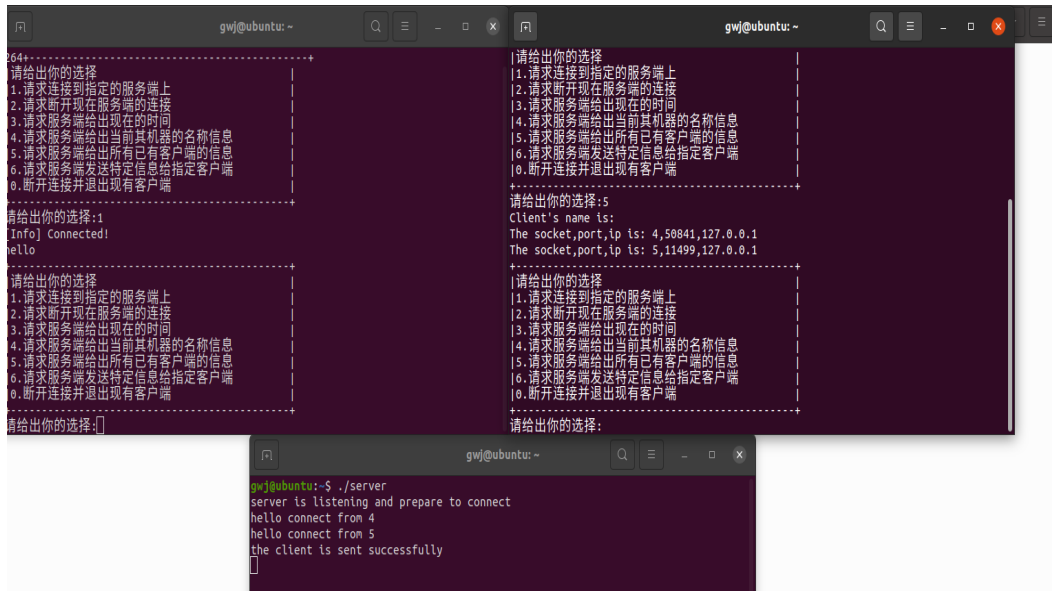
相关的服务器的处理代码片段：

```

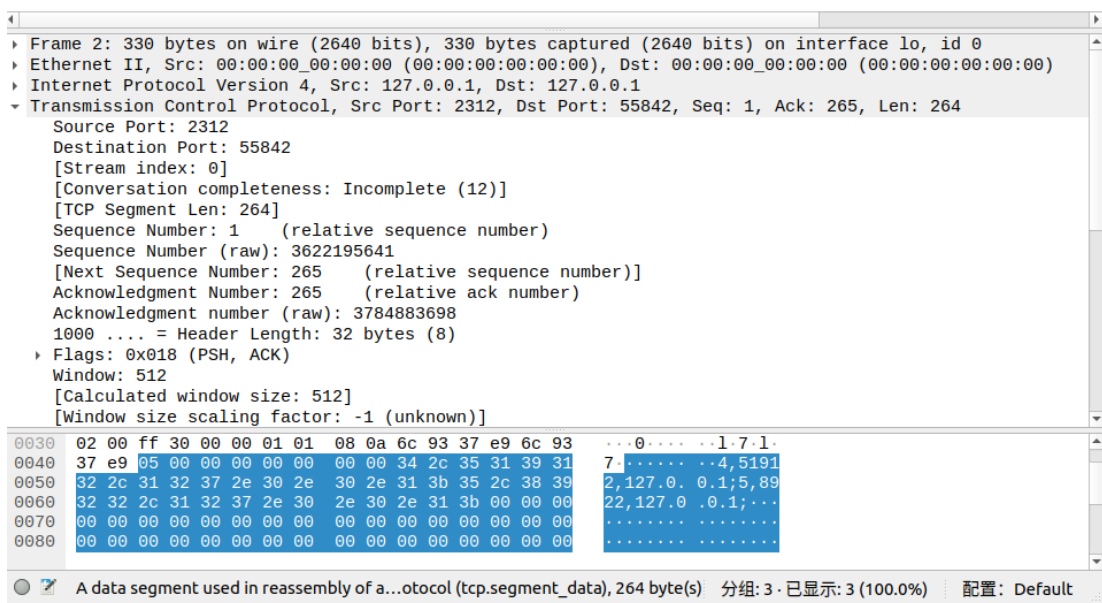
case NAME:{
    Message mes;
    mes.type = NAME;
    gethostname(mes.data, sizeof(mes.data));
    if(send(con_fd,&mes,sizeof(mes),0)<0){
        cout<<"it is failed";
    }
    cout<<"the name is sent sucessfully"<<endl;
    break;
    //get cli_list
}
  
```

- 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。

（开两个客户端进行尝试）



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）：



相关的服务器的处理代码片段：

```
case CLIENT:{
    Message mes;
    mes.type = CLIENT;
    strcpy(mes.data,"");
}
```

```

for(int i=0;i<clients.size();i++){
    int sock = clients[i].sock;
    int port = clients[i].port;
    string addr = clients[i].addr;
    //    cout<<sock<<endl<<port<<endl<<addr<<endl;
    string ans = to_string(sock)+","+ to_string(port)+ ","+ addr+"";
    strcat(mes.data, ans.c_str());
}
//    strcat(mes.data,0);
if(send(con_fd,&mes,sizeof(mes),0)<0){
    cout<<"send error";
}
//cout<<mes.data<<endl;
cout<< "the client is sent successfully"<<endl;
break;
        //send message
}

```

- 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：

```

+-----+
|请给出你的选择
|1.请求连接到指定的服务端上
|2.请求断开现在服务端的连接
|3.请求服务端给出现在的时间
|4.请求服务端给出当前其机器的名称信息
|5.请求服务端给出所有已有客户端的信息
|6.请求服务端发送特定信息给指定客户端
|0.断开连接并退出现有客户端
+-----+
请给出你的选择:6
请给出要传递到的IP地址:127.0.0.1
请给出要选择的端口号:50841
请给出你需要发送的内容: hello,50841

```

服务器：

```

it is sent to ip :127.0.0.1port :50841

```

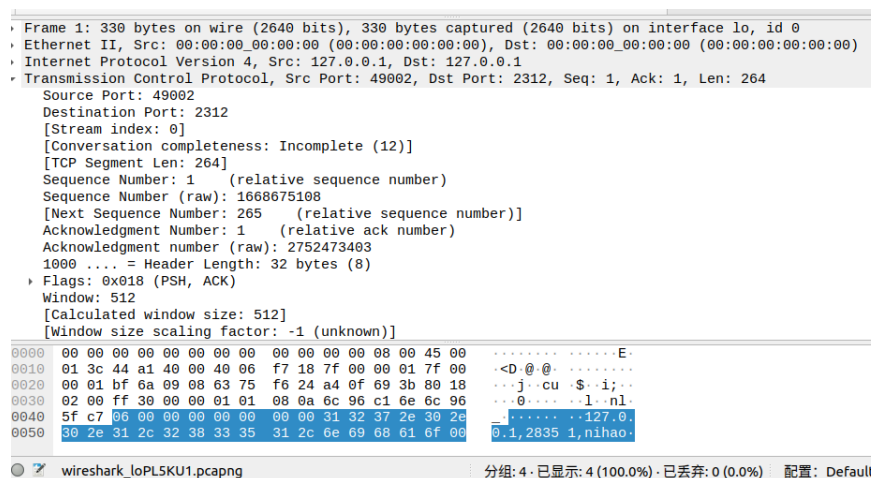
接收消息的客户端：

```

|请给出你的选择
|1.请求连接到指定的服务端上
|2.请求断开现在服务端的连接
|3.请求服务端给出现在的时间
|4.请求服务端给出当前其机器的名称信息
|5.请求服务端给出所有已有客户端的信息
|6.请求服务端发送特定信息给指定客户端
|0.断开连接并退出现有客户端
+-----+
请给出你的选择:
Receive repost: hello,50841

```

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 SendRequest()
{
    if (sockfd == -1)
    {
        printf("[Error] No connection detected!\n");
        return;
    }
    Message req;
    char ip[20];
    int port;
    char content[220];
    //分别为, 要发送的 ip 地址, 端口号, 以及要发送的内容
    //这里我们规定协议如下, 对于发送的 req.type 如上定义
    //对于发送的额外 data[256], ip:port:content, 其中 ip 预留 20 位即可, int 是 4 位, 分号
    //3 位, 还剩 229 位, 这里留 220 位
    req.type=SEND;

```



```

printf("请给出要传递到的 IP 地址:");
scanf("%s",ip);
printf("请给出要选择的端口号:");
scanf("%d",&port);
printf("请给出你需要发送的内容: ");
getchar();
cin.get(content,200);
//scanf("%s", content);
sprintf(req.data, "%s,%d", ip, port);
sprintf(req.data + strlen(req.data), "%s", content);
//本次要发送的信息较多
if (send(sockfd, &req,sizeof(req), 0) < 0)
{
    printf("[Error] SendRequest send fail, error: %s\n", strerror(errno));
    return;
}
Message msg;
long type = SEND;
if (msgrcv(msgID, &msg, 256, 0 , 0) < 0)
{
    printf("[Error] SendRequest recv fail, error: %s\n", strerror(errno));
    return;
}
printf("Server response:%s\n",msg.data);//接收服务端回答我们出现的问题等
}

```

相关的客户端（发送和接收消息）处理代码片段：

```

case SEND:{
    //获得端口号和 ip 地址
    //data 顺序为 IP port 内容 切割
    string content = string(rec.data);
    string addr = content.substr(0,content.find(","));
    content = content.substr(content.find(",")+1);
    int port = atoi(content.substr(0,content.find(",")).c_str());
    content = content.substr(content.find(",")+1);
    string ans = content;
    //cout<<addr<<endl<<port<<endl;
    //遍历 cli_list 找到发送客户端
    int flag = 0 ;
    for(int i=0;i<clients.size();i++)
    {
        if(clients[i].addr == addr && clients[i].port == port)

```

```

        {
            Message mes;
            mes.type = SEND;
            strcpy(mes.data, content.c_str());
            cout<<clients[i].sock<<endl;
            if(send(clients[i].sock,&mes,sizeof(mes),0)<0)
            {
                cout<<"it is error to send";
            }
            cout<<"it is sent to ip :"+ addr
+"port :"+to_string(port)<<endl;
            flag = 1;
            break;
        }
    }
    if(!flag){
        Message mes;
        mes.type = SEND;
        strcpy(mes.data,"no this port");
        cout<<"not found the destination"<<endl;
        send(con_fd,&mes,sizeof(mes),0);
    }
}

```

- 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。

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。

```
for (int i=0 ;i<=100 ;i++){
    Message req;
    req.type=TIME;
    if (send(sockfd, &req,sizeof(req), 0) < 0)
    {
        printf("[Error] TimeRequest send fail, error: %s\n", strerror(errno));
        return;
    }
    Message msg;
    long type = TIME;
    if (msgrcv(msgID, &msg, 256, 0, 0) < 0)
    {
        printf("[Error] TimeRequest recv fail, error: %s\n", strerror(errno));
        return;
    }
    time_t t;
    //sscanf 是按照自己希望的类型重新读入，这里变相转换了类型
    sscanf(msg.data, "%ld", &t);
    printf("Time: %s\n", ctime(&t));
}
```

The image shows two terminal windows side-by-side. The left window, titled 'gwj@ubuntu: ~', displays the program's output for the first six iterations of a loop, each showing 'Time: Tue Nov 1 05:16:59 2022'. Below this, a menu is displayed with the prompt '请给出你的选择' (Please give your choice) and a list of options: 1. 请求连接到指定的服务端上 (Request connection to the specified server), 2. 请求断开现在服务端的连接 (Request to disconnect from the current server), 3. 请求服务端给出现在的时间 (Request server to give current time), 4. 请求服务端给出当前机器的名称信息 (Request server to give machine name), 5. 请求服务端给出所有已有客户端的信息 (Request server to give info of all clients), 6. 请求服务端发送特定信息给指定客户端 (Request server to send specific info to specific client), and 0. 断开连接并退出有客户端 (Disconnect and exit if there are clients). The right window, also titled 'gwj@ubuntu: ~', shows the program sending 100 successful time requests, with output lines ranging from 'the time is sent successfully77' to 'the time is sent successfully99'.



## 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？**

答：不需要，因为客户端只需要寻找一个能够使用的端口发送数据包即可。其源端口会通过 Socket 的 API 来自动选择一个还没有被占用的端口，每一次调用时客户端的端口不会保持不变。

- **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：1) 客户端 IP 地址互不相同时，可以直接通过 IP 地址加以区分。

2) IP 地址存在相同的情况时，对不同的 IP 地址使用 (1) 方法加以区分，对相同的 IP 地址则根据客户端产生的 Socket 描述符加以区分。

- **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，仍然有效。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

在本次实验中，我觉得最困难的部分在于理解进程间通信的那些函数，其实后来看的话并不多，但是刚接触的时候对那些复杂的参数真的是一头雾水，如果能给予一个较好的说明文档的话可能会加快对这个的理解。另外我觉得困难的地方在于理解子线程通过接收并且不断用消息通道发给主线程的这一目的，因为服务端处理一个请求的时间可能很慢，如果不用子线程的话，就会使得客户端程序卡顿，无法进行其他操作，这样其实是不方便的。另外，对于线程间传递消息和客户端服务端传递消息的那些函数我曾一度搞混过，后来要自己多敲几次才能更加熟悉后区分清楚。

另外，在实验中由于存在通信的两方，所以我第一次感受到了需要双方约定协议，约定怎么发送 Message，怎么解析，怎么传回。这个实验由于我是分组实验，其间也体会了协议要如何做出，如何协调。

可以说，通过这次实验，我对于 socket 编程有了更好的理解，明白了这其中很多函数的目的，在之后能够更好的编写这一方面的有关程序。