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:	刘雨祺
学　　院:	计算机学院
系:	计算机科学与技术
专　　业:	计算机科学与技术
学　　号:	3190105147
指导教师:	张泉方

2021 年　　12 月　　25 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：马骥　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

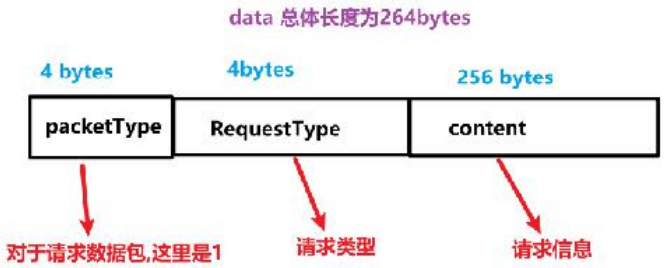
请将以下内容和本实验报告一起打包成一个压缩文件上传：

- **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

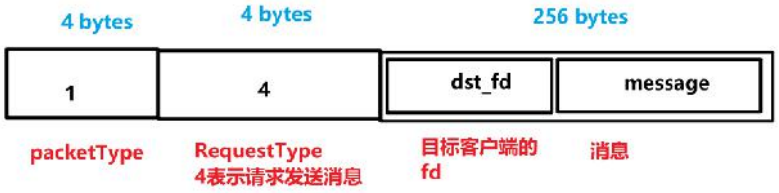
```
typedef enum { REQUEST = 1, RESPONSE, INSTRUCT } packetType;
typedef enum { TIME = 1, NAME, LIST, MESSAGE ,DISCONNECT} requestType;
typedef enum { FORWARD = 1, TERMINATE } instructType;
typedef enum { CORRECT = 1, WRONG } typeOfResponse;
```

数据包类型均用枚举定义，下文中描写数据包类型时，直接使用其枚举成员表示。

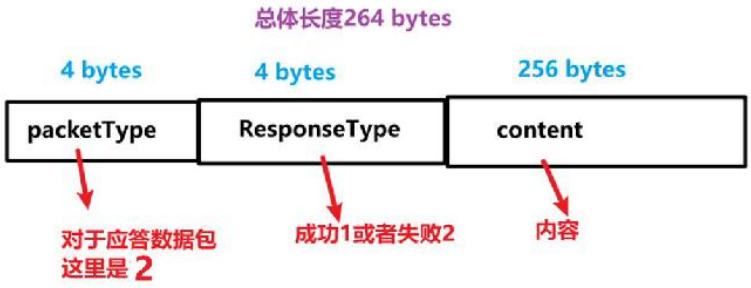


packetType	RequestType	content	类型
REQUEST	TIME	(MiniClient) Get Time Request	Time
REQUEST	NAME	(MiniClient) Get Name Request	Name
REQUEST	LIST	(MiniClient) Get List Request	List
REQUEST	MESSAGE	目标客户端的 fd + message	Message
REQUEST	DISCONNECT	(MiniClient) DisConnect Request	Disconnect

MESSAGE 数据包的格式如下：

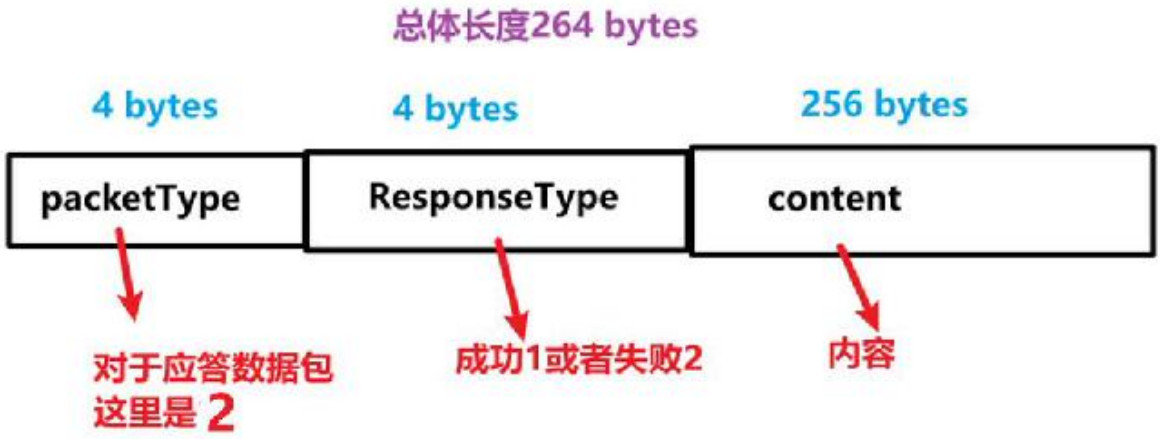


- 描述响应数据包的格式（画图说明），响应类型的定义



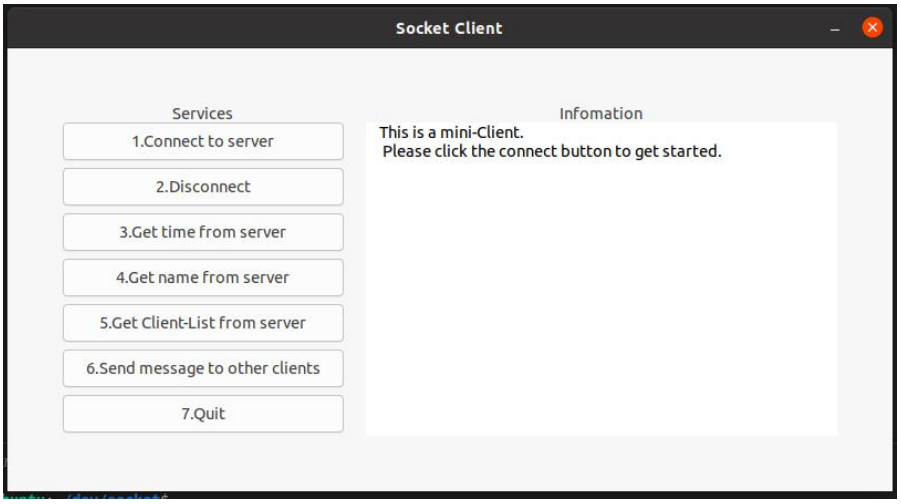
packetType	ResponseType	类型	content
RESPONSE	1	响应时间请求	(Miniserver) Date:.....Time:.....
RESPONSE	1	响应名字请求	(Miniserver) ******
RESPONSE	1	响应客户端列表请求	(Miniserver) Socketfd:** Port: ** IP: **
RESPONSE	1	响应消息请求	(Miniserver)src_fd:** src_ip:**src_port:**
RESPONSE	2(表示出错)	转发消息失败	(Miniserver)no socketfd:**

- 描述指示数据包的格式（画图说明），指示类型的定义



packetType	InstructType	content
INSTRUCT	FORWORD	(MiniServer)src_fd:**;src_ip:**;src_port:**;message
INSTRUCT	TERMINATE	(MiniServer)closed server

- 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）

客户端使用 Linux 图形界面开发工具 gtk 实现，主线程循环为 gtk 实现的 gtk_main()函数，gtk_main()函数内部实现了主循环。

我们初始化组件之后，调用 gtk_main()，即开始运行主线程循环，等待鼠标点击事件的发生。

```
//show widgets
gtk_widget_show_all (window);
//wait for events occur
gtk_main ();
```

- 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 *waitServer(void* para){
    packet pkt;
    int ret = 0;
    while(1) {
        memset(pkt.data, 0, sizeof(pkt.data));
        ret = recv( *(int*)para, (char *)&pkt, sizeof(pkt), 0);
        if(pkt.type == TERMINATE && pkt.pType == INSTRUCT) {
            print_info("(Client) Server connection terminated!\n");
            pthread_exit(0);
        }

        print_info(pkt.data);
    }
}
```

子线程循环执行 recv()函数，直到接收到有效数据包。

- 服务器初始运行后显示的界面

```
lyq@ubuntu:~/dev/socket/server$ ./miniserver
Initialize the Server !
server IP:127.0.0.1
BIND START!
LISTEN START!SERVER START!
```

- 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）

主线程循环中循环调用 IsAcceptConnect()函数。

```
while (TRUE)
{
    // dead loop and continue listenning
    IsAcceptConnect();
}
```

```
//do not beyond the LISTENQUEUESIZE
if (cfd.tail < LISTENQUEUESIZE){
    int tmp = accept(fd_of_server, (struct sockaddr *)&addr_of_client, &sizeof_sockaddr_in);
    // source mutex
    pthread_mutex_lock(&mutex);
    // normal linklist operation:add node
    cfd.fd[cfd.tail] = tmp;
    cfd.tail++;
    IsCFD_full = TRUE;
    for (int i = 0; i < LISTENQUEUESIZE; ++i){
        if (list_client[i].fd == 0){
            list_client[i].fd = cfd.fd[cfd.tail - 1];
            list_client[i].port = addr_of_client.sin_port;
            list_client[i].addr = addr_of_client.sin_addr;
            //inet_ntoa()用来将参数in 所指的网络二进制的数字转换成网络地址，然后将指向此网络地址字符串的指针返回.
            printf(" Socketfd:%d Port:%hu IP:%s Connecting\n",cfd.fd[cfd.tail - 1], list_client[i].port,inet_ntoa(list_client[i].addr));
            IsCFD_full = 0;
            break;
        }
    }
    if (IsCFD_full!=TRUE){
        printf("socketfd:%d connect SUCCESSFULLY!\n", cfd.fd[cfd.tail-1]);
        create_connection_pthread();
        }
    else{
        printf("Socketfd:%d connect FAILED!\n", cfd.fd[cfd.tail-1]);
    }
    pthread_mutex_unlock(&mutex);
}
else{
    printf("Client Connect is full!\n");
}
return 0;
```

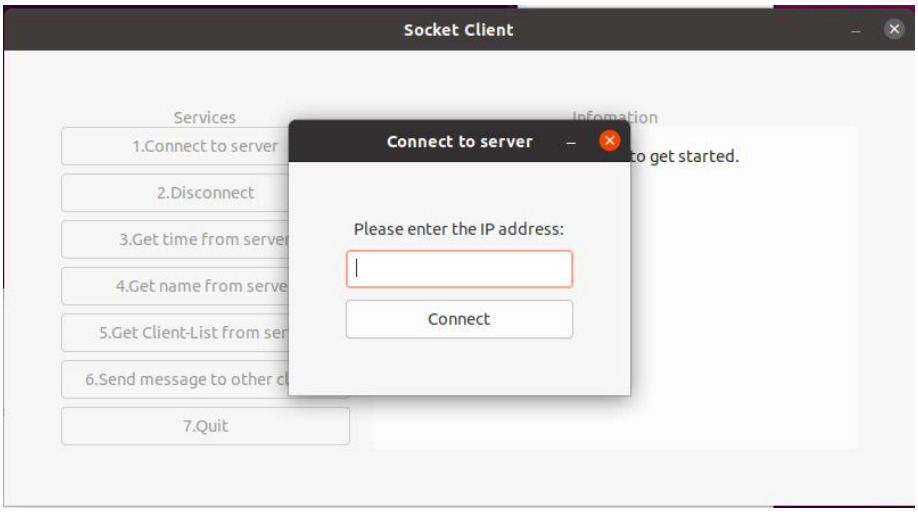
通过 accept()函数接收到客户端建立连接的请求后，为客户端建立连接。

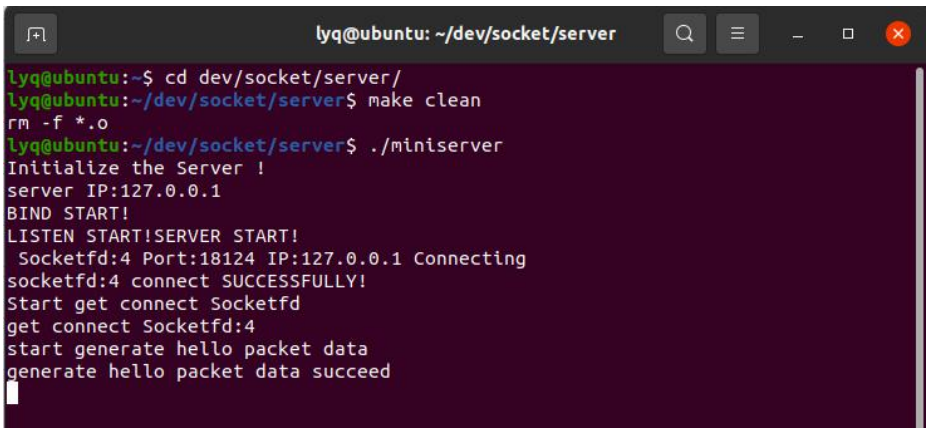
- 服务器的客户端处理子线程循环关键代码截图（描述总体，省略细节部分）

```c
while (TRUE)
{
    //recv return the value it read or retunr -1 if error occur
    num_of_bytes = recv(fd, (char *)&pckt, sizeof(pckt), 0);
    if (num_of_bytes < 0)
    {
        printf("from socketfd:%d get error packet\n", fd);
        break;
    }
    printf("from socketfd:%d get packet\n", fd);
    printf("from socketfd:%d packet packetType:%d, type:%d,data:%s\n", fd, pckt.packetType, pckt.type, pckt.
    if (handle_Packet(&pckt, fd) == -1)
    {
        return NULL;
    }
}
```

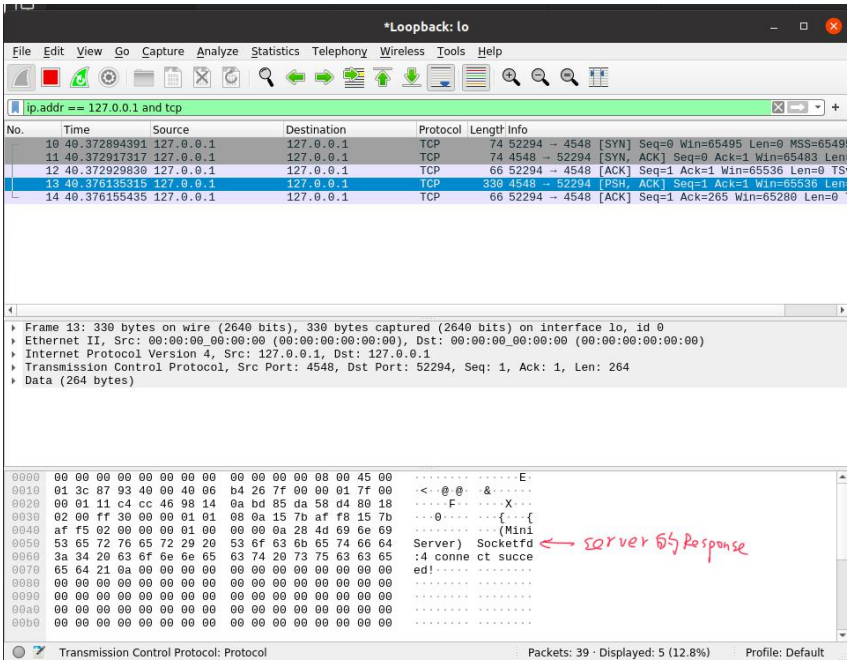
与客户端建立连接后，循环调用 recv()函数，等待客户端发送数据。
接收到的数据调用 handle_Packet()函数处理。

- 客户端选择连接功能时，客户端和服务端显示内容截图。

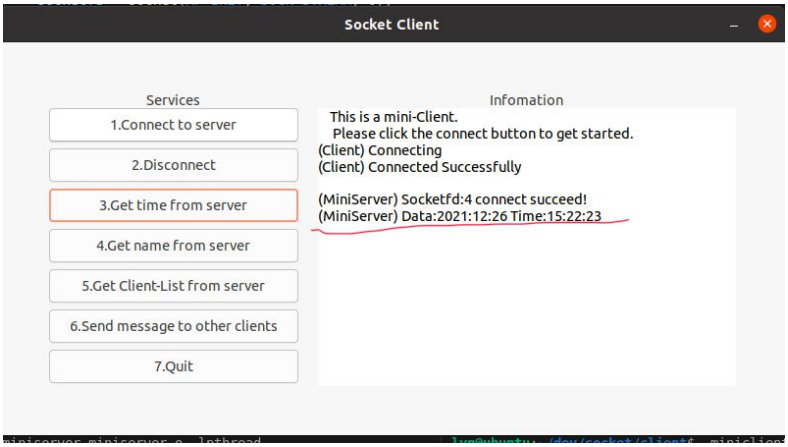




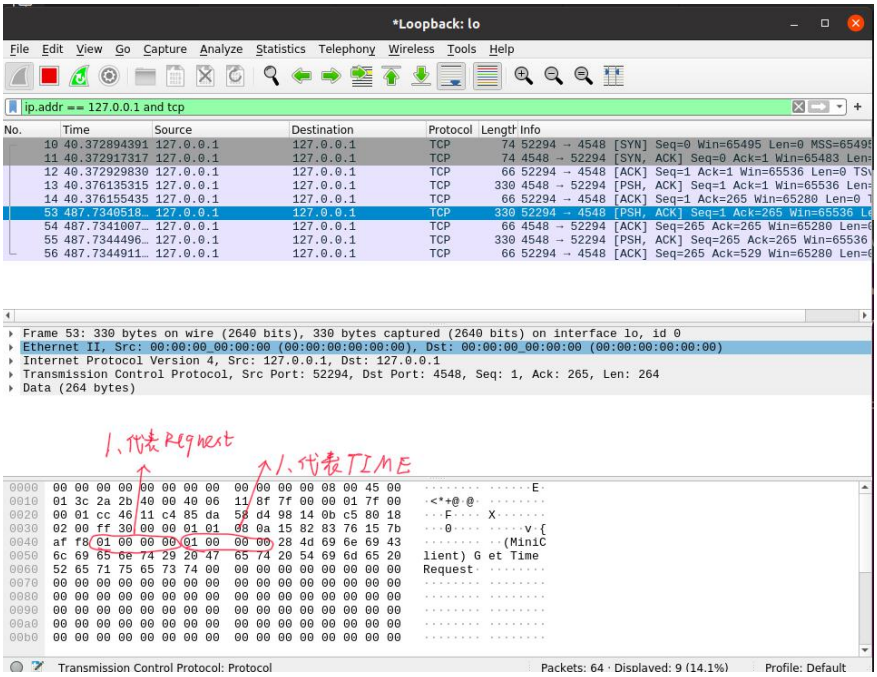
Wireshark 抓取的数据包截图：

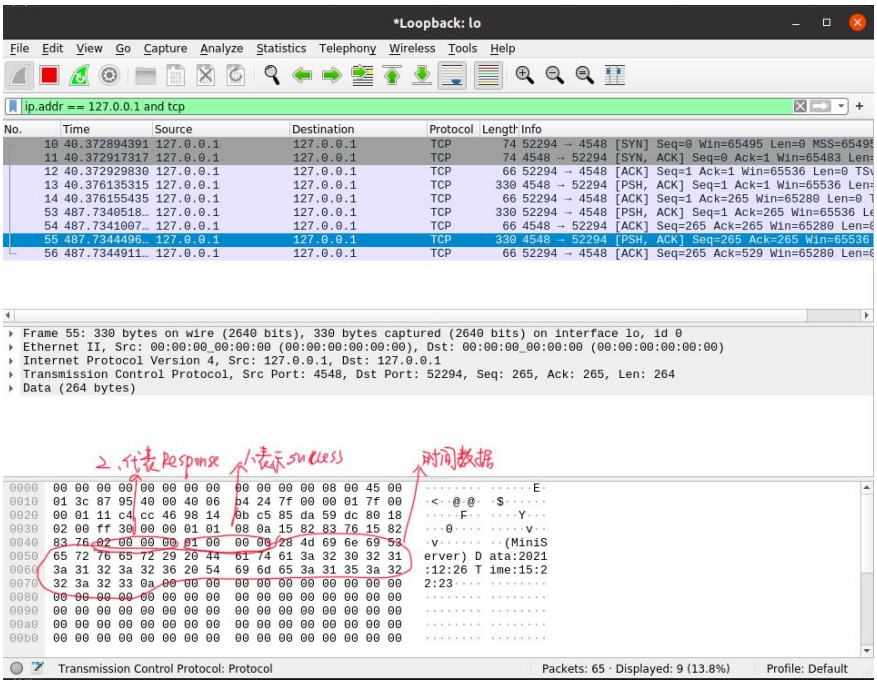


- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

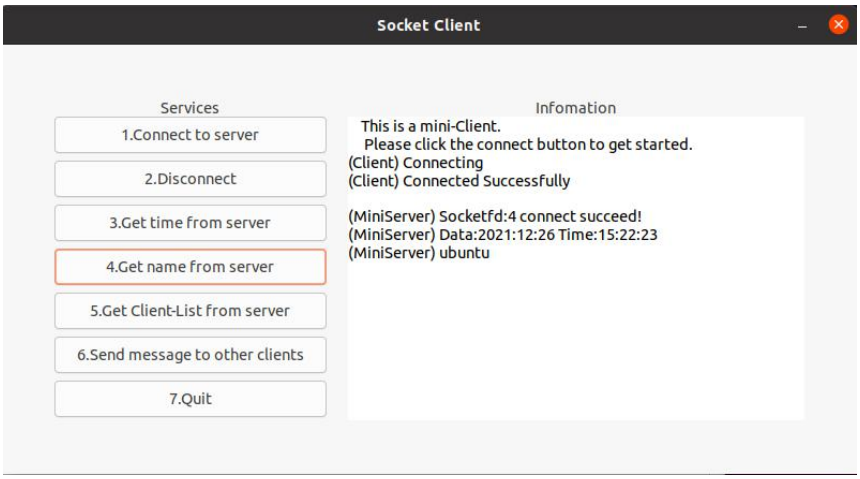


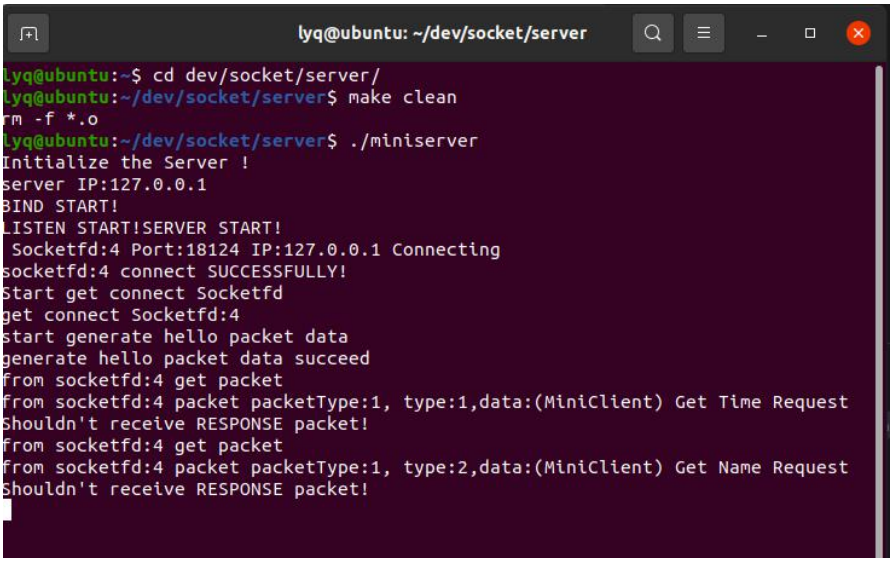
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的时间数据对应的位置）：



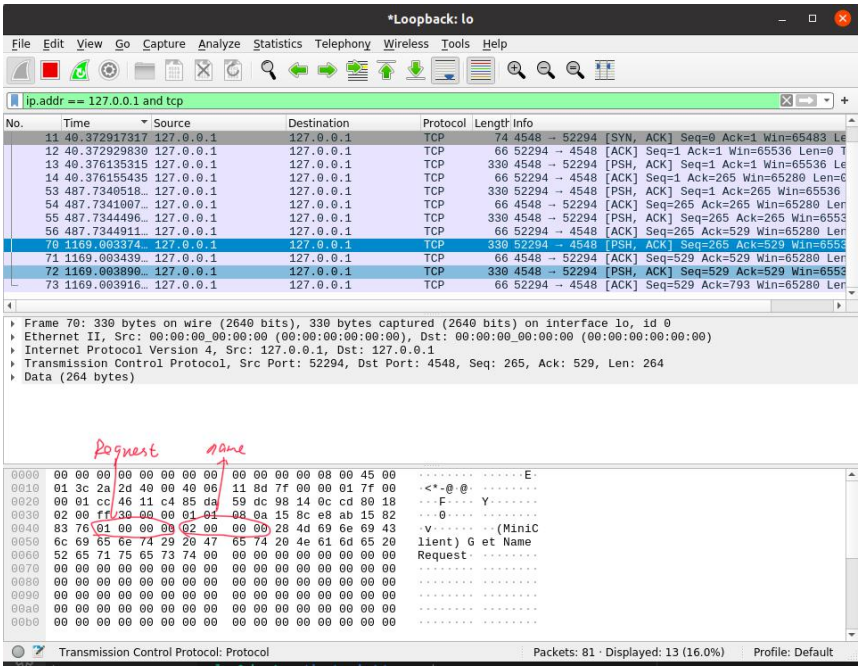


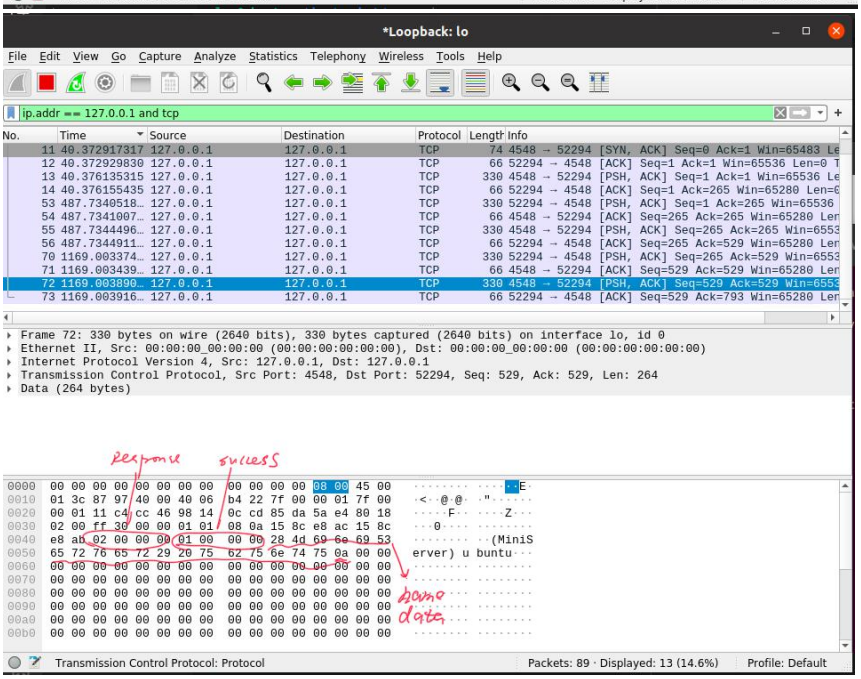
- 客户端选择获取名字功能时，客户端和服务端显示内容截图。





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）：

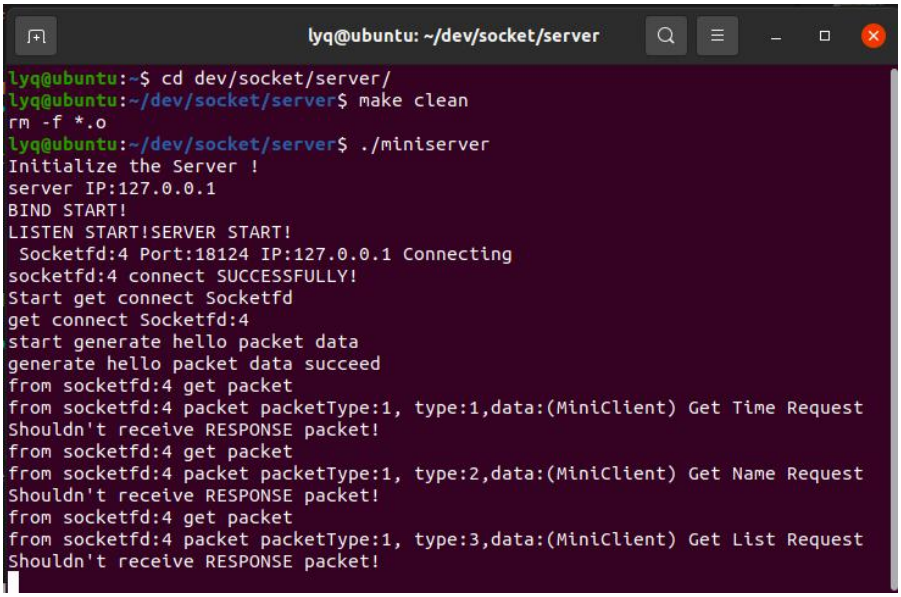




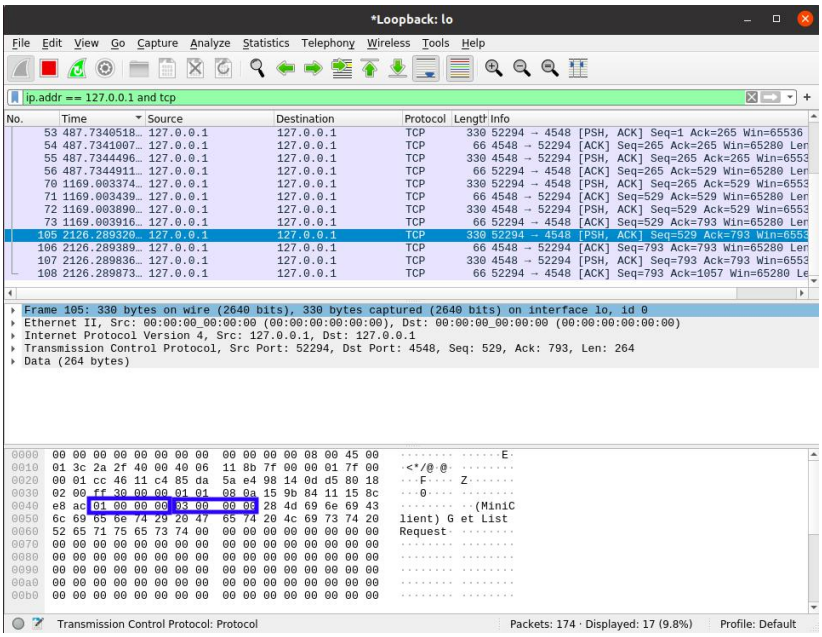
相关的服务器的处理代码片段:

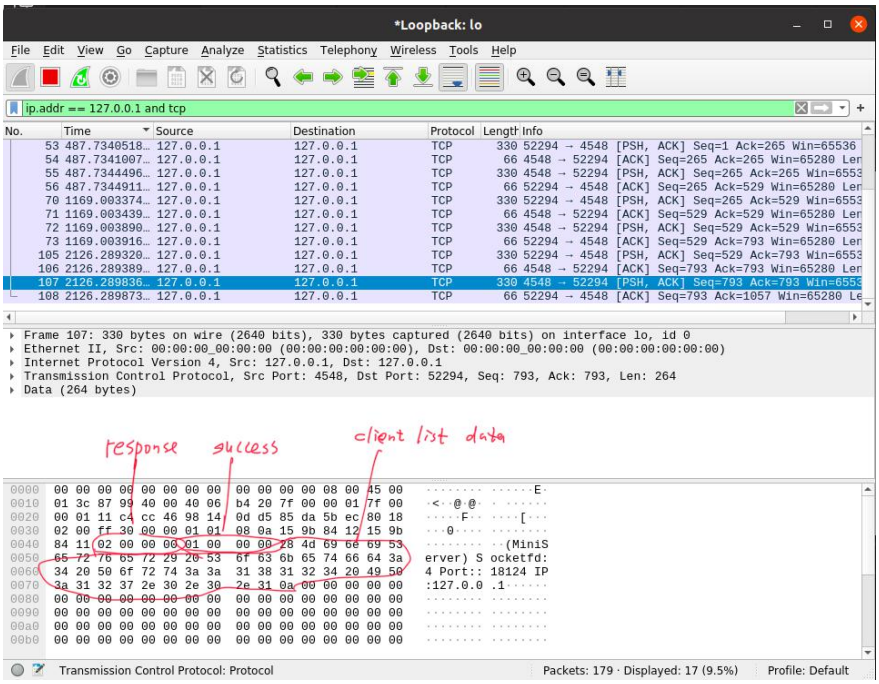
```
void handle_Name_Packet(struct packet *one_packet, int fd)
{
    struct packet socket_packet;
    char host_name[128];
    memset(socket_packet.data,0,MAXDATALEN);
    //Put the name of the current host in no more than LEN bytes of NAME
    int hostres = gethostname(host_name, sizeof(host_name));
    if(hostres == -1)
    {
        printf("ERR! GET HOSTNAME FAILED!\n");
    }
    //write host into socketpactek data
    sprintf(socket_packet.data, "(MiniServer) %s\n", host_name);
    socket_packet.packetType = RESPONSE;
    socket_packet.type = CORRECT;
    send(fd, (char *)&socket_packet, sizeof(socket_packet), 0);
}
```

- 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。



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）：



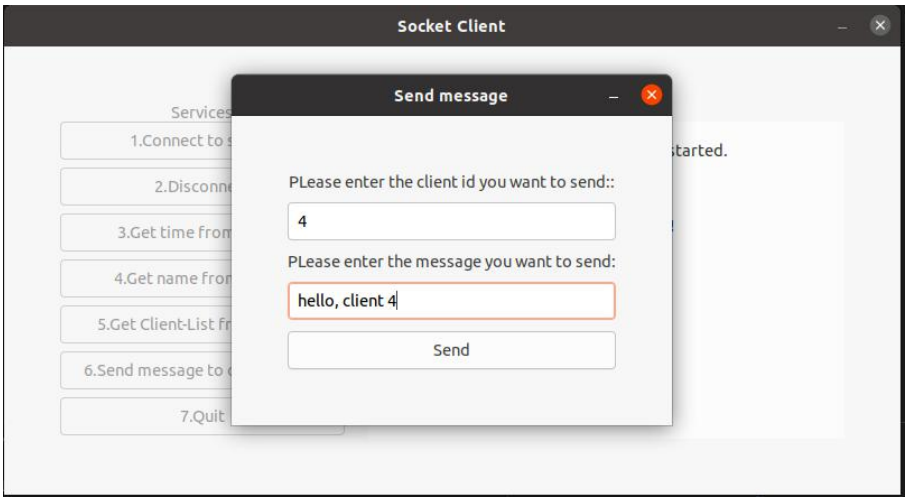


相关的服务器的处理代码片段:

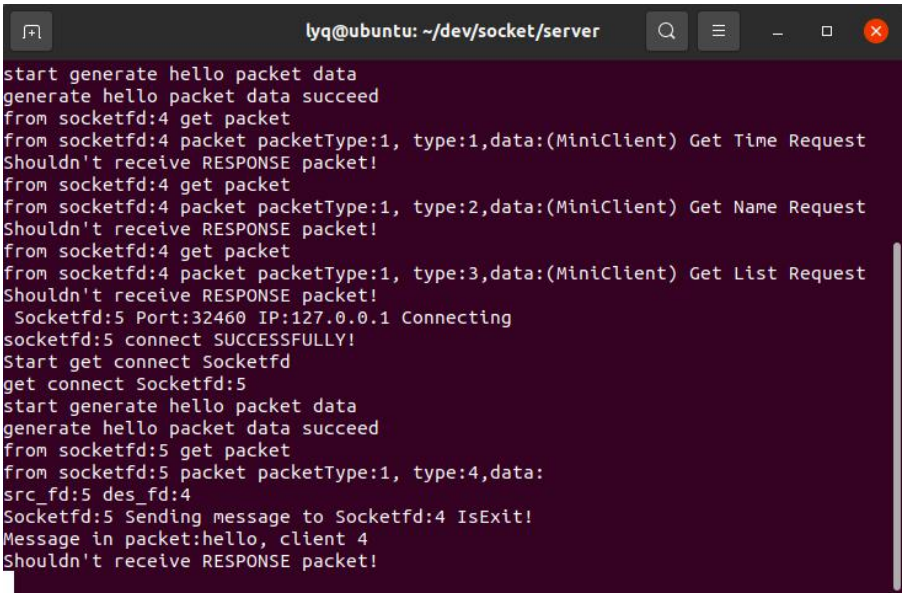
```c
void handle_List_Packet(struct packet *one_packet, int fd)
{
    struct packet socket_packet;
    socket_packet.packetType = RESPONSE;
    socket_packet.type = CORRECT;
    memset(socket_packet.data,0,MAXDATALEN);
    pthread_mutex_lock(&mutex);
    int j = 0;
    for (int i = 0; i < LISTENQUEUESIZE; ++i)
    {
        if (list_client[i].fd > 0)
        {
            /*
                include: fd + port + addr
            */
            j+=sprintf(socket_packet.data+j, "(MiniServer) Socketfd:%d ", list_client[i].fd);
            j+=sprintf(socket_packet.data+j, "Port::%hu ", list_client[i].port);
            j+=sprintf(socket_packet.data+j, "IP:%s\n", inet_ntoa(list_client[i].addr));
        }
    }
    pthread_mutex_unlock(&mutex);
    send(fd, (char *)&socket_packet, sizeof(socket_packet), 0);
}
```

- 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。

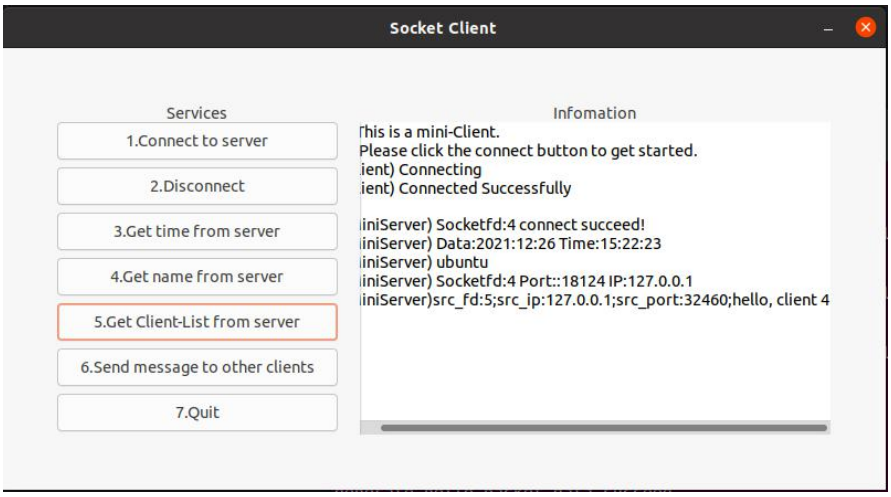
  发送消息的客户端:



  服务器:

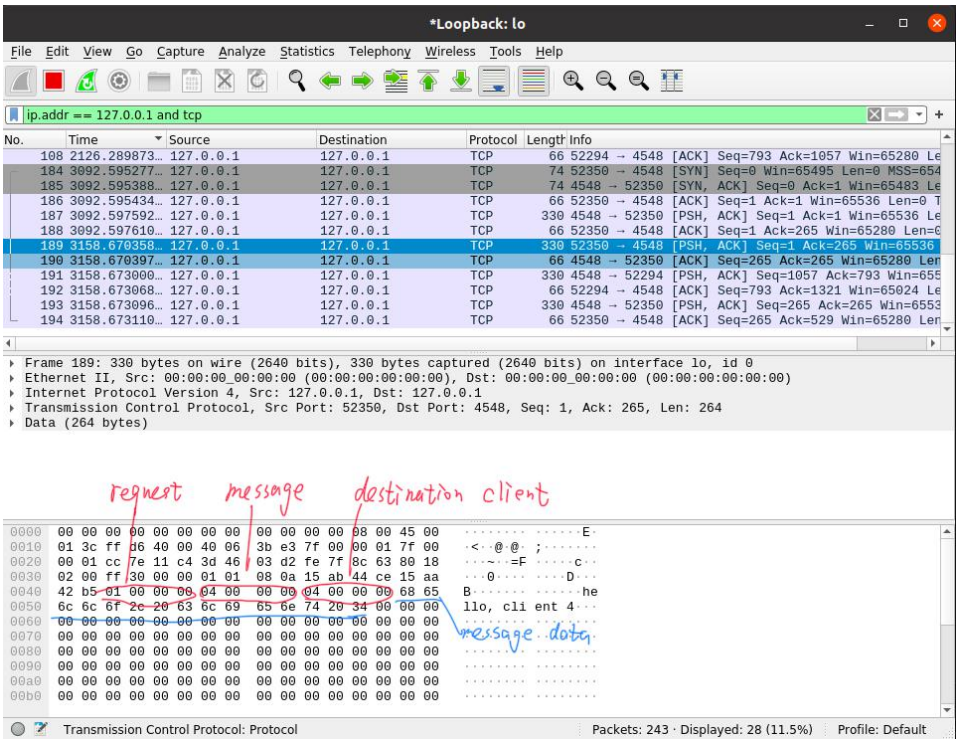


接收消息的客户端：

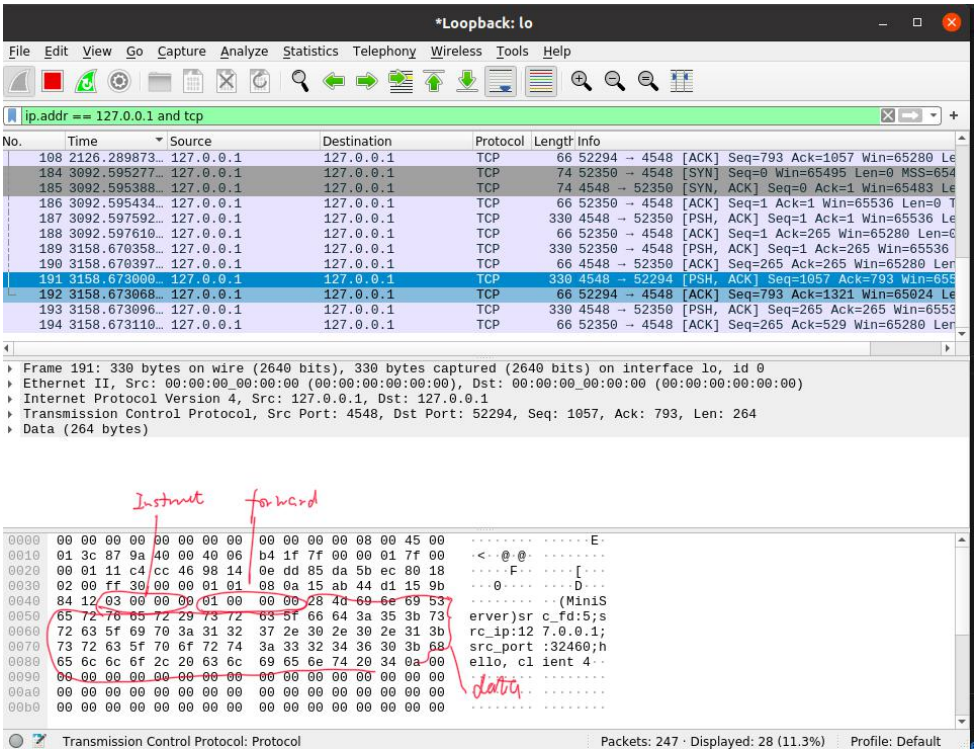


Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送：



接收：



相关的服务器的处理代码片段：

```c
void handle_Message_Packet(struct packet *one_packet, int fd)
{
    struct packet socket_packet;
    memset(socket_packet.data,0,MAXDATALEN);
    // fd of destination
    int fd_of_destination = *((int *)one_packet->data);

    int IsExit = 0;
    printf("src_fd:%d des_fd:%d\n", fd, fd_of_destination);
    pthread_mutex_lock(&mutex);
    for (int i = 0; i < LISTENQUEUESIZE; ++i)
    {
        //find and label the isExit as 1
        if (list_client[i].fd == fd_of_destination)
        {
            IsExit = 1;
            break;
        }
    }
    pthread_mutex_unlock(&mutex);
    if (IsExit == 1)
    {
        printf("Socketfd:%d Sending message to Socketfd:%d IsExit!\n", fd, fd_of_destination);
        printf("Message in packet:%s\n", one_packet->data + sizeof(int));
        socket_packet.packetType = INSTRUCT;
        socket_packet.type = FORWARD;
        //write into socketpactek data
        int thisI=0;
        for(int i=0;i<LISTENQUEUESIZE;i++){
            if(list_client[i].fd==fd){
                thisI=i;
            }
        }
        sprintf(socket_packet.data, "(MiniServer)src_fd:%d;src_ip:%s;src_port:%hu;%s\n", fd,inet_ntoa(list_clien
        send(fd_of_destination, (char *)&socket_packet, sizeof(socket_packet), 0);
        struct packet response_packet;
        memset(response_packet.data,0,MAXDATALEN);
        response_packet.packetType = RESPONSE;
        response_packet.type = CORRECT;
        //write into response socketpacket data
        sprintf(response_packet.data, "(MiniServer) Message to Socketfd:%d send successfully!\n", fd_of_destinat
        send(fd, (char *)&response_packet, sizeof(response_packet), 0);
    }
    else
    {
        socket_packet.packetType = RESPONSE;
        socket_packet.type = CORRECT;
        int num_of_bytes = sprintf(socket_packet.data, "(MiniServer) No Socketfd:%d\b", fd_of_destination);
        send(fd, (char *)&socket_packet, sizeof(socket_packet), 0);
    }
}
```

相关的客户端（发送和接收消息）处理代码片段:

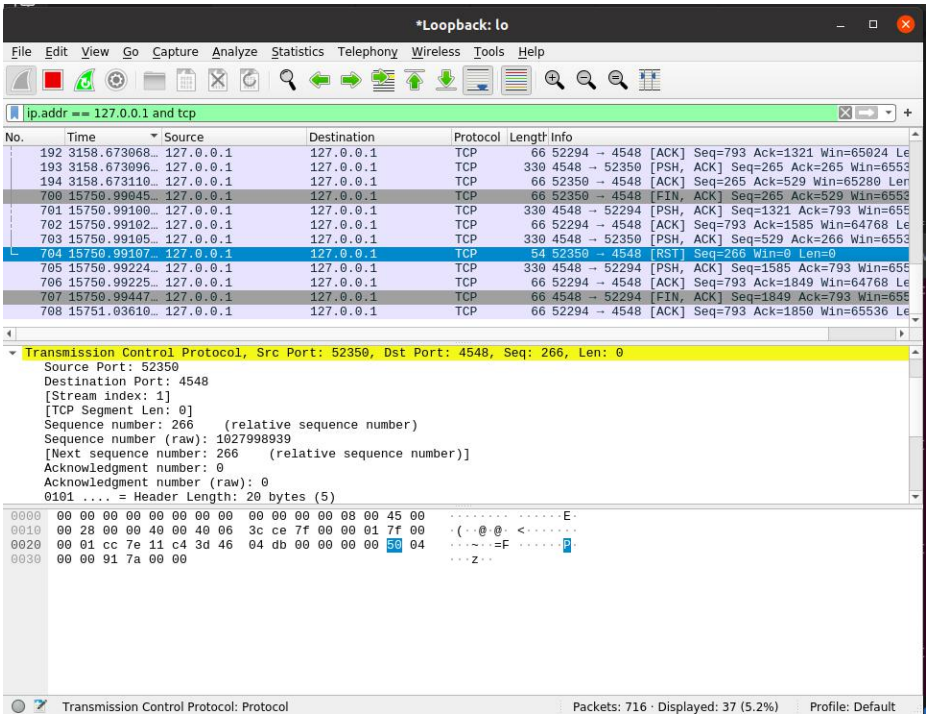
发送:

```c
void sendMessageRequestPacket(const char *client,const char *msg) {
    int destClient;
    packet pkt;
    pkt.pType = REQUEST;
    pkt.type = (int)MESSAGE;
    memset(pkt.data, 0, sizeof(pkt.data));

    sscanf(client,"%d", &destClient);
    //printf("client id:%d\n",destClient);
    memcpy(pkt.data, &destClient, sizeof(int));

    strcat(pkt.data + sizeof(int),msg);


    send(socketfd, (char *)&pkt, sizeof(pkt), 0);

}
```

接收：

```c
void *waitServer(void* para){
    packet pkt;
    int ret = 0;
    while(1) {
        memset(pkt.data, 0, sizeof(pkt.data));
        ret = recv( *(int*)para, (char *)&pkt, sizeof(pkt), 0);
        if(pkt.type == TERMINATE && pkt.pType == INSTRUCT) {
            print_info("(Client) Server connection terminated!\n");
            pthread_exit(0);
        }

        print_info(pkt.data);
    }
}
```

- 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。



通过 wireshark 可以发现并没有发出 TCP 释放连接的信号,但是发送了一个别的消息(见上图),导致服务端退出了,使用 echo $?查看返回信息,发现$?是 141,因为 141-128=13 表示错误原因是 SIGPIPE:管道破裂。(这个信号一般在进程间通讯产生，好比采用 FIFO(管道)通讯的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到 SIGPIPE 信号。此外用 Socket 通讯的两个进程，写进程在写 Socket 的时候，读进程已经终止。)

我认为是 client 的 gtk_main()异常退出导致的异常中断,也是因为这个原因,我没有观察到 TCP 连接状态的变化,因为服务端进程已经终止.通过 netstat -an 可以发现 4548 端口已经不再监听

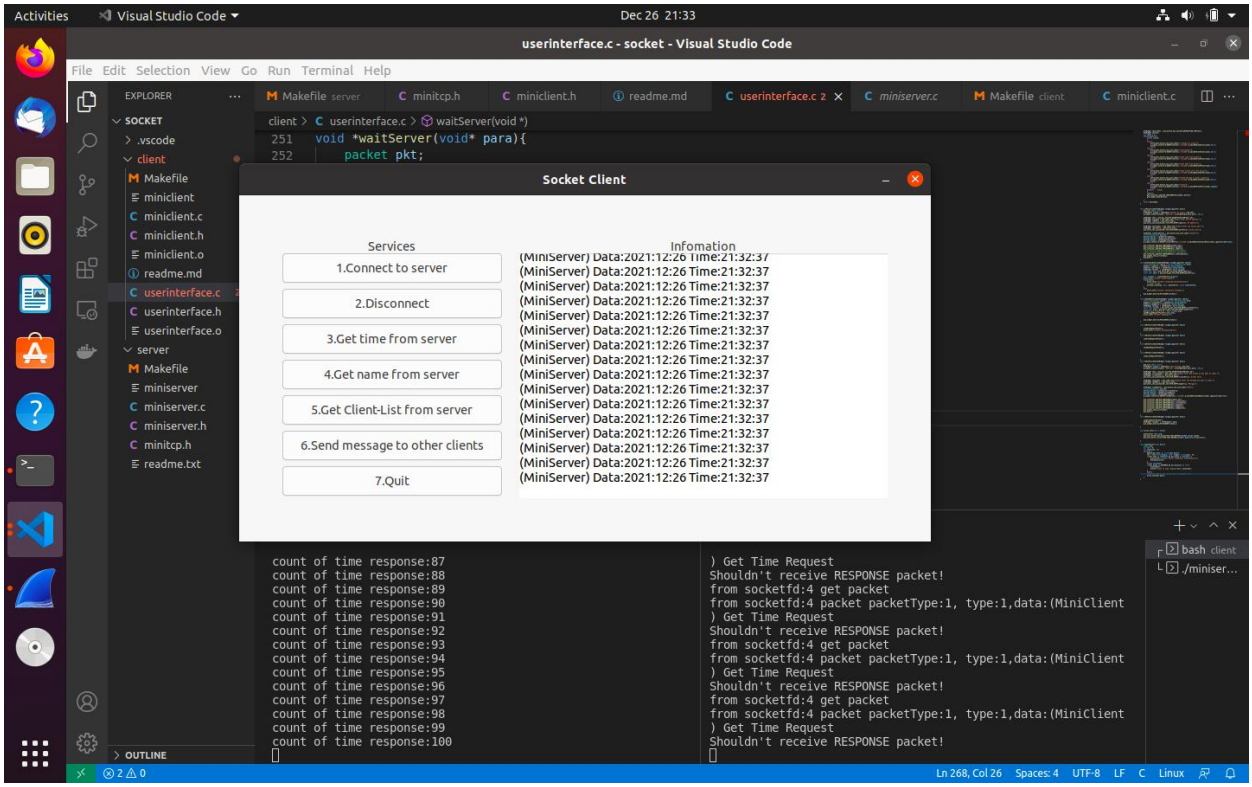
- 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

  由于服务进程已经终止,所以无法查看此处已经异常退出的连接是否还在.

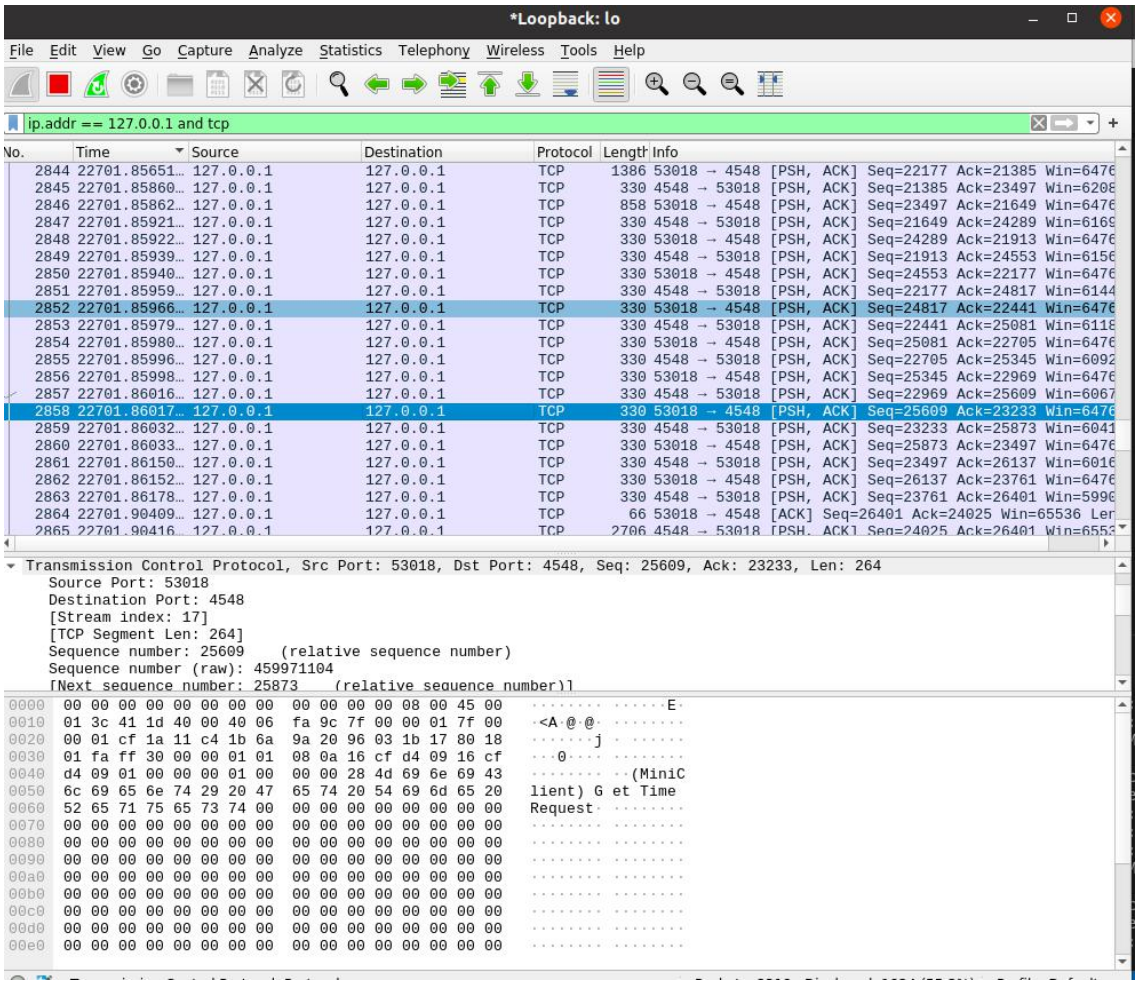
  从理论上推测,由于 client 并没有向 server 发送一个断开 tcp 连接的消息,所以如果服务端进程还在的话,之前异常退出的连接应该还在,如果给这个之前异常退出的客户端发送异常,会由于 server 发送给目的 client 消息之后无法收到确认信号,导致转发消息失败,返回一个 WRONG type 的 INSTRUCT 类型数据包.

- 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。

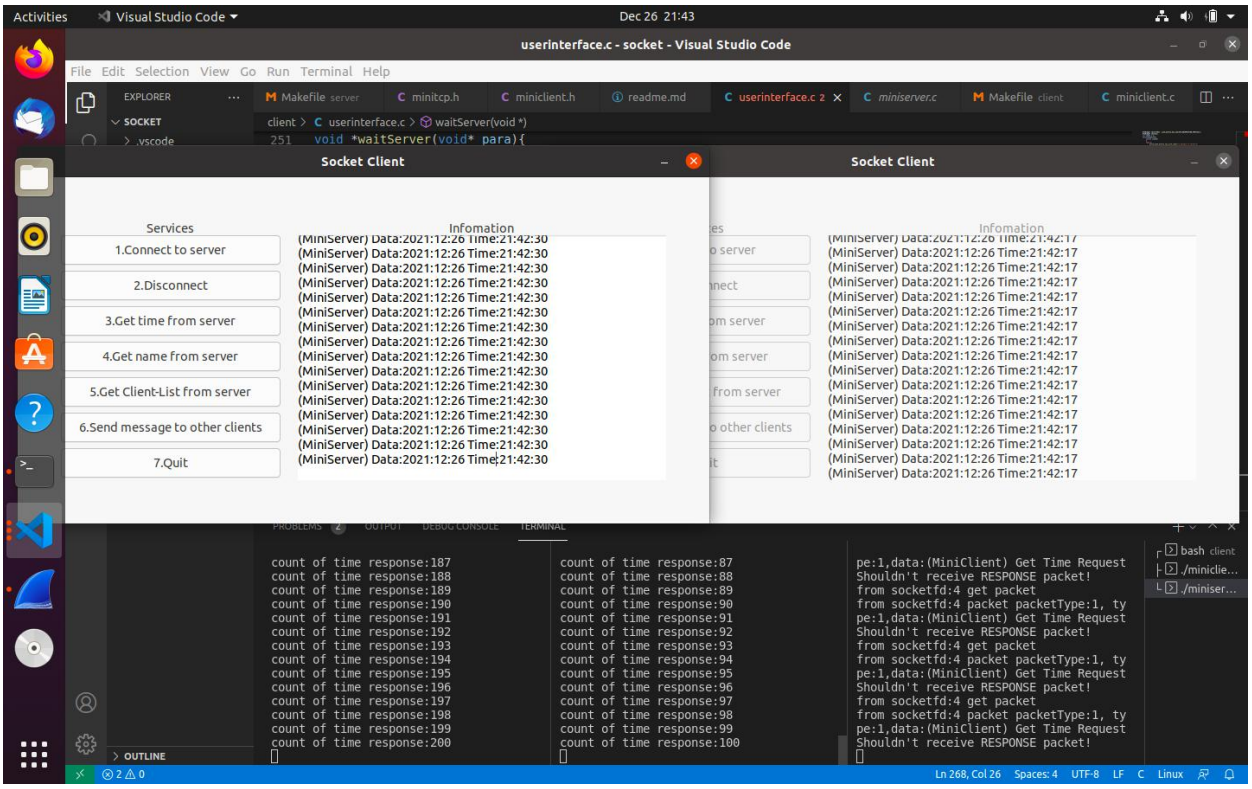
从 terminal 里可以发现，客户端确实收到了 100 个响应。



经统计，发现服务端实际发送的数据包多于 100 个。服务端发送了一些 TCP Segment Len=0 的纯通知滑动窗口数据包。



- 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



（服务器为右下角的 terminal）

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？

  是的。

  随机生成的。

  否。

- 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？

否。因为 listen 只是让服务端处于监听状态，调用 accept 后才开始接收请求。

- 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?

不一致。通过观察 WireShark 发现发送了若干次纯通知滑动窗口的数据包。

- 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?

根据 socket 套接字的 file descriptor 来区分。

- 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）

  处于 TIME_WAIT 状态

  保持了大概几十秒。查阅资料后发现持续时间是两倍的分段最大生存期。

- 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?

从理论上讲,客户端断网之后异常退出,是没有给服务器发送断开连接的请求的,但是在我们的试验中发现这个基于 gtk 的图形客户端在断电的时候,会触发一个事件给服务器发送一个信号,同时会导致服务器进程终止(这一点的解释请参见上文倒数第三第四题)。

试验的预期效果是 TCP 连接状态不会发生变化,还是 established

服务器如果想要检查连接是否继续有效,可以使用 select 函数,在源代码开头我添加了对这个函数的解释.

```
/*
select() can implement the unblock programming
int select(int nfds,  fd_set* readset,  fd_set* writeset,  fe_set* exceptset,  struct timeval*
timeout);
      nfds            需要检查的文件描述字个数
```

```
        readset      用来检查可读性的一组文件描述字。
        writeset     用来检查可写性的一组文件描述字。
        exceptset    用来检查是否有异常条件出现的文件描述字。(注：错误不包括在异常条件之内)
        timeout      超时，填 NULL 为阻塞，填 0 为非阻塞，其他为一段超时时间
return value: 返回 fd 的总数，错误时返回 SOCKET_ERROR
*/
```

它的原理就是服务器定时向各个client发送确认信息,如果一定时间内没有收到该client的回答,就可以认为它出现了异常,连接取消.

七、 讨论、心得

实验中我和我的队友对于数据包的首部的定义没有很好的统一, 这导致客户端在判断响应包类型时出现了一些 bug。排查问题后，我修改了判定条件，成功解决了这个问题。