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：　黄正谦

2022 年　12 月　12 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：_____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

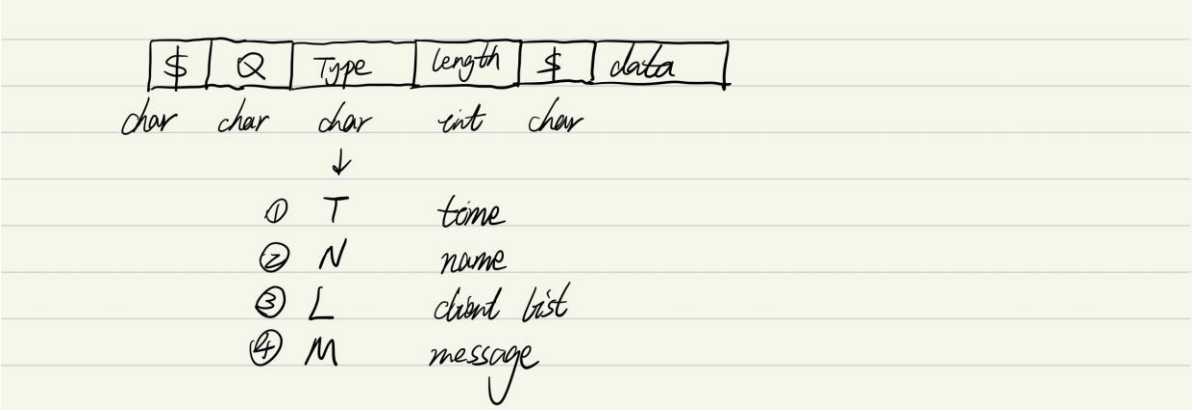
五、 实验数据记录和处理

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：

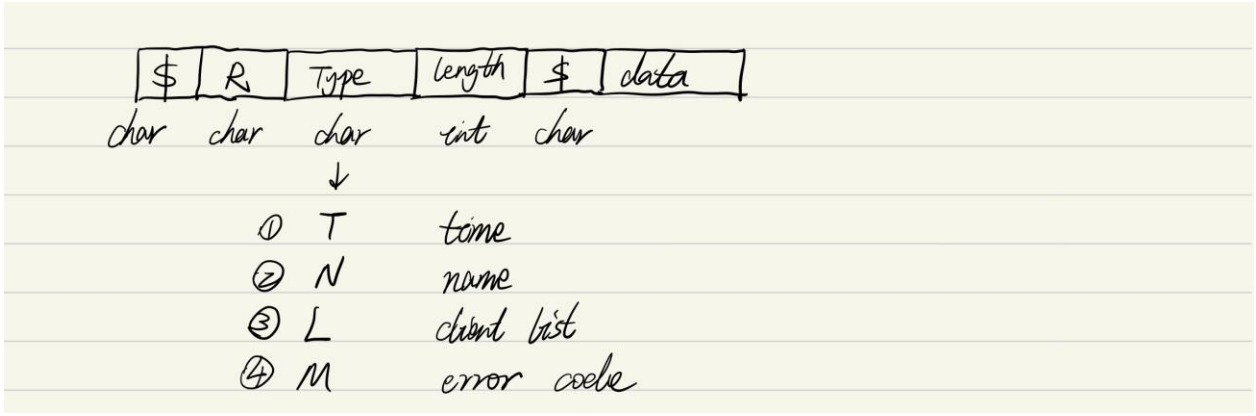
- **源代码：客户端和服务端的代码分别在一个目录**
- **可执行文件：可运行的.exe 文件或 Linux 可执行文件，客户端和服务端各一个**

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）。

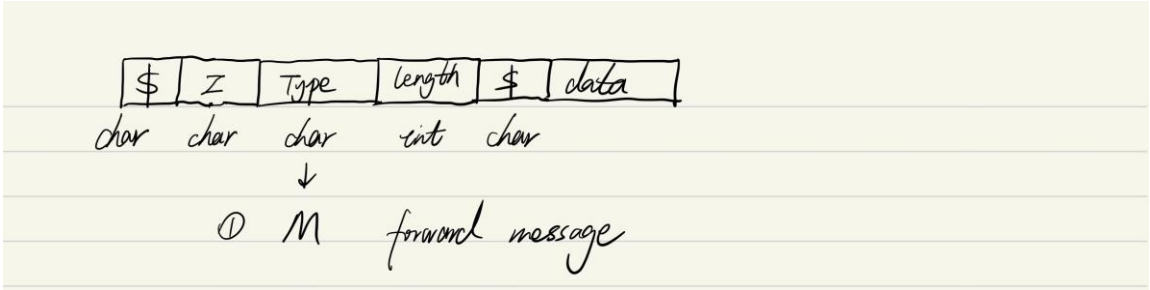
- 描述请求数据包的格式（画图说明），请求类型的定义



- 描述响应数据包的格式（画图说明），响应类型的定义



- 描述指示数据包的格式（画图说明），指示类型的定义



- 客户端初始运行后显示的菜单选项

```
Client:
[1] Connect
[2] Disconnect
[3] Get time
[4] Get host name
[5] Get activitiy link list
[6] Send message
[7] Exit
Select a number above:
```

- 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）

  选择菜单

```
    while(1) {
        printf("\n");
        printf("Client:\n");
        printf("[1] Connect\n");
        printf("[2] Disconnect\n");
        printf("[3] Get time\n");
        printf("[4] Get host name\n");
        printf("[5] Get activitiy link list\n");
        printf("[6] Send message\n");
        printf("[7] Exit\n");
        printf("Select a number above: \n");
```

  等待用户选择

  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）

```
        scanf("%s", buffer);
        if (strcmp(buffer, "1") == 0) {…
        if (strcmp(buffer, "2") == 0) {…
        if (strcmp(buffer, "3") == 0) {…
        if (strcmp(buffer, "4") == 0) {…
        if (strcmp(buffer, "5") == 0) {…
        if (strcmp(buffer, "6") == 0) {…
        if (strcmp(buffer, "7") == 0) {…

    }
```

如果选择连接，则判断是否已经连接，是则提示用户，否则进行连接并创建进程

```
if (strcmp(buffer, "1") == 0) {
    // have connected
    if (connected == 1) {…
    // have not connected
    // connect failed
    else if (connect(client_socket, (struct sockaddr*)&server_
    // connect succeed
    else {…
}
```

- 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）

  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）。

```
    // read response continue
    while (1) {
        // judge if a complete response packet
        if (recv(conn_id, response, BUFFER_SIZE, 0) > 0) {
            printf("Client: receive: %s\n", response);
            if (*response == '$' && *(response + 1) == 'R' && *((char*)((int*)(respon
                if (*(response + 2) == 'T') {…
                else if (*(response + 2) == 'N') {…
                else if (*(response + 2) == 'L') {…
                else if (*(response + 2) == 'M') {…
            }
            if (*response == '$' && *(response + 1) == 'I' && *((char*)((int*)(respon
                if (*(response + 2) == 'M') {…
            }
        }
        memset(response, 0, BUFFER_SIZE);
    }
```

- 服务器初始运行后显示的界面

```
Server: Create socket succeed
Server: bind succeed
Server: listen succeed
```

- 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()。

```
// send and receive data
char buffer[BUFFER_SIZE];
while (1) {
    new_socket = accept(server_socket, (struct sockaddr*)&new_addr, &new_addr_size);
    if (new_socket < 0) {…

    printf("Connection accepted from %s:%d\n", inet_ntoa(new_addr.sin_addr), ntohs(new

    client_list[Client_num].addr = new_addr;
    client_list[Client_num].index = Client_num;
    client_list[Client_num].socket = new_socket;
    client_list[Client_num].exist = 1;
    client_list[Client_num].connected = 1;
    Client_num++;

    pthread_t tid;
    if (pthread_create(&tid, NULL, HandleThread, &client_list[Client_num - 1]) != 0) {
    else {…
}
```

- 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）

  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。

```
    char receive_packet[BUFFER_SIZE];

    // read response continue
    while (1) {
        memset(receive_packet, 0, sizeof(receive_packet));
        if (recv(conn_id, receive_packet, BUFFER_SIZE, 0) > 0) {
            // judge if a complete response packet
            if (*receive_packet == '$' && *(receive_packet + 1) == 'Q' && *((char*)((int*)
                if (*(receive_packet + 2) == 'T') {…
                else if (*(receive_packet + 2) == 'N') {…
                else if (*(receive_packet + 2) == 'L') {…
                else if (*(receive_packet + 2) == 'M') {…
            }
        }
        else {
            connected[list_number] = 0;
        }
        bzero(receive_packet, sizeof(receive_packet));
    }
```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

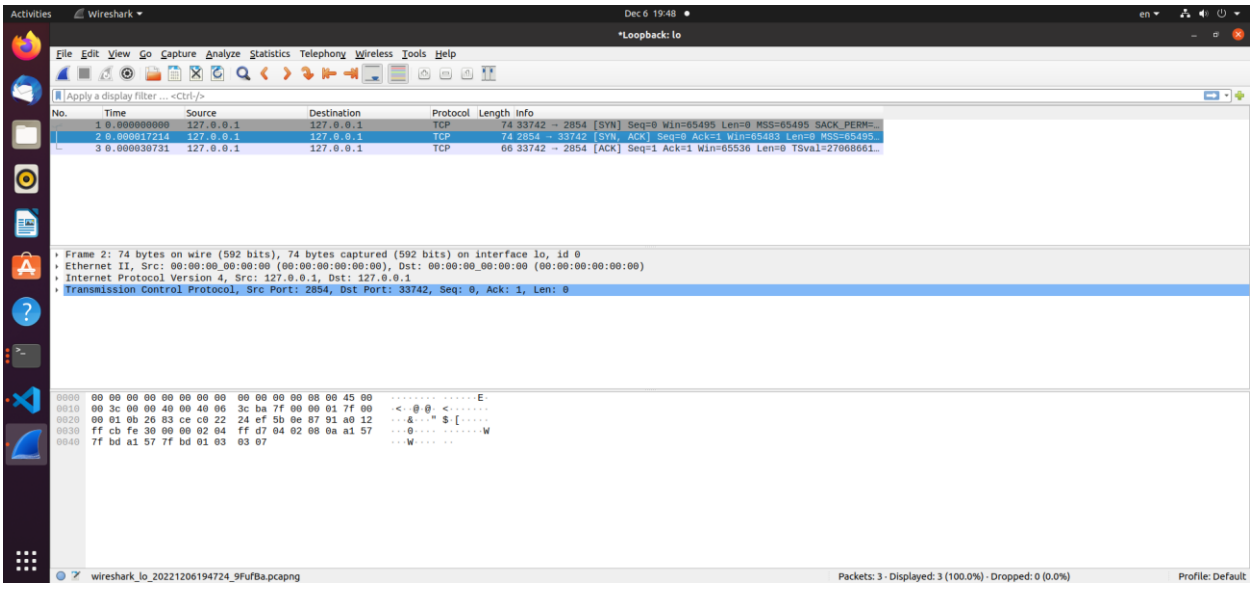
  服务器

```
Connection accepted from 127.0.0.1:51199
Server: Connection succeed
```

  客户端

```
1
Server: Connection succeed
```

  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

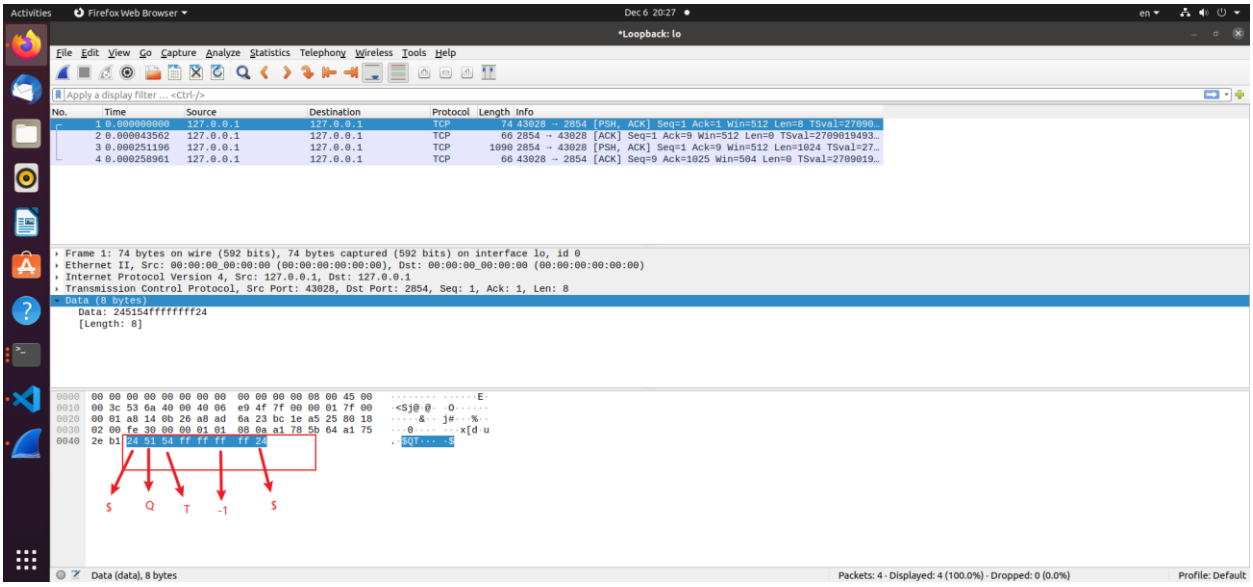
```
3
Successfully received, time is
Wed Dec 28 15:16:26 2022
```

服务器

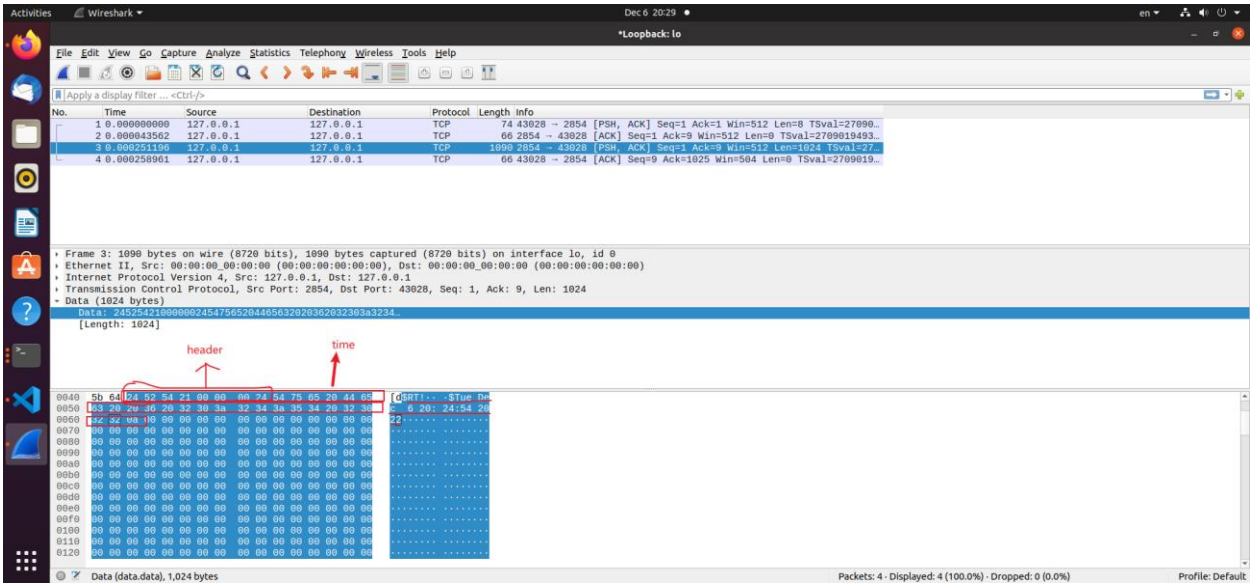
```
time_cnt = 1 Server send to Client[0] the host time
```

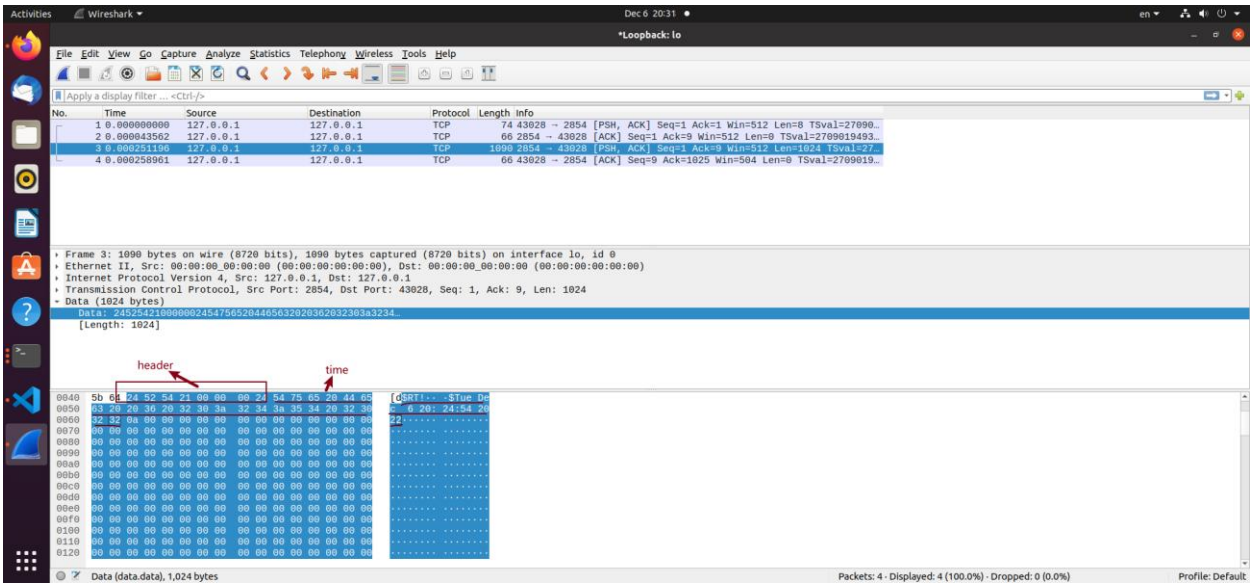
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的时间数据对应的位置）：

请求数据包



响应数据包





- 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

```
4
Name is ZaricEspana
```

服务器

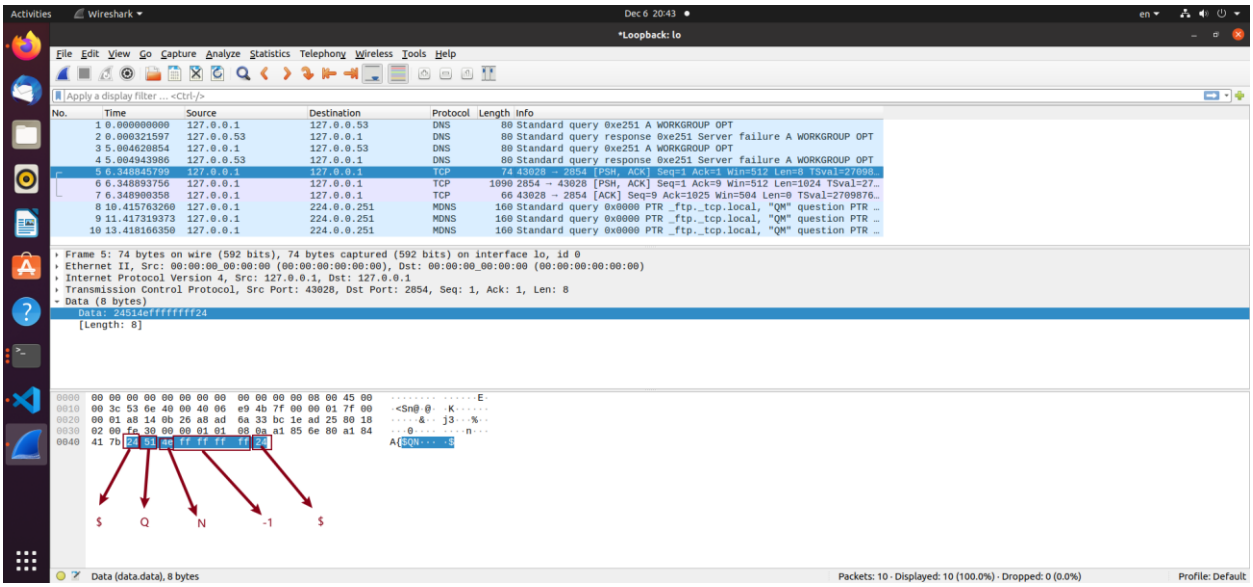
```
Server send to Client[0] the host name
```

相关代码

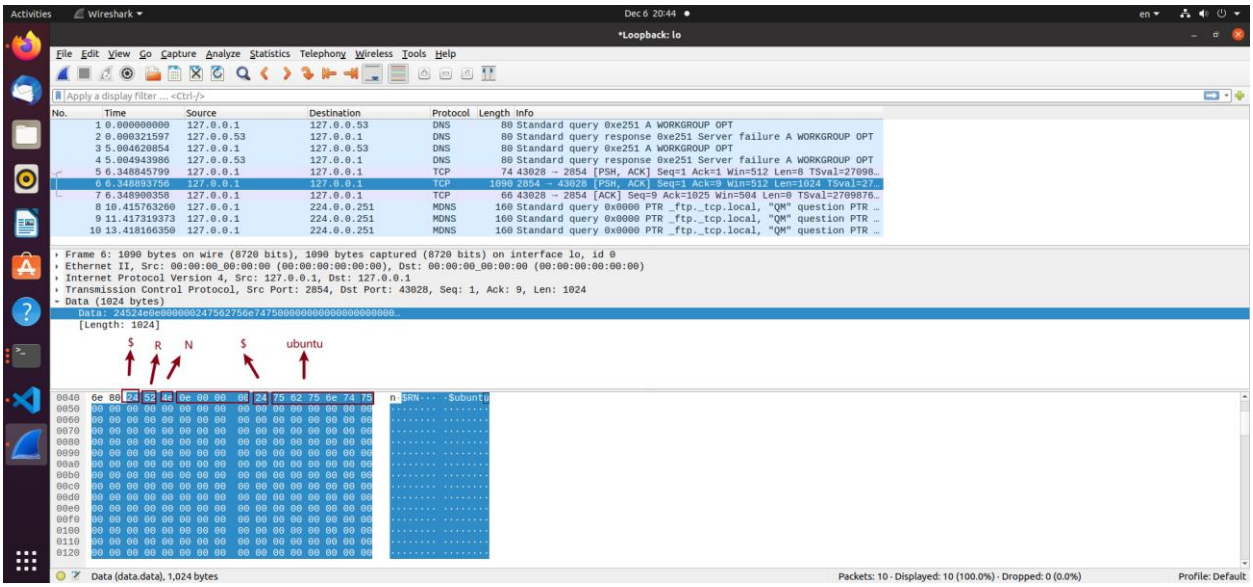
```
printf("time_cnt = %d", time_cnt);
// get host time
time_t *timep = malloc(sizeof(time_t));
time(timep);
char *host_time = ctime(timep);

// build response packet
char *response = (char*)malloc(sizeof(char) * 4 + sizeof(int) + size
memset(response, 0, sizeof(char) * 4 + sizeof(int) + sizeof(char) *
*response = '$';
*(response + 1) = 'R';
*(response + 2) = 'T';
int length = strlen(host_time) + sizeof(char) * 4 + sizeof(int);
*(int*)(response + 3) = length;
*((char*)((int *)(response + 3) + 1)) = '$';
strcat((char*)((int *)(response + 3) + 1), host_time);

// send the time packet to client
if (send(conn_id, response, BUFFER_SIZE, 0) > 0) {
    printf("Server send to Client[%d] the host time\n", list_number)
}
```

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）：

请求数据包



响应数据包



相关的服务器的处理代码片段：

相关代码以及注释

```
else if (*(receive_packet + 2) == 'N') {
    char host_name[BUFFER_SIZE];
    gethostname(host_name, sizeof(host_name));

    char *response = (char*)malloc(sizeof(char) * 4 + sizeof(int) + sizeof
    memset(response, 0, sizeof(char) * 4 + sizeof(int) + sizeof(char) * 10
    *response = '$';
    *(response + 1) = 'R';
    *(response + 2) = 'N';
    int length = (int)(strlen(host_name) + sizeof(char) * 4 + sizeof(int));

    *(int*)(response + 3) = length;
    *((char*)((int *)(response + 3) + 1)) = '$';
    strcat((char*)((int *)(response + 3) + 1), host_name);

    if (send(conn_id, response, BUFFER_SIZE, 0) > 0) {// some improvement
        printf("Server send to Client[%d] the host name\n", list_number);
    }
    // free the space allocated
    free(response);
}
```

- 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

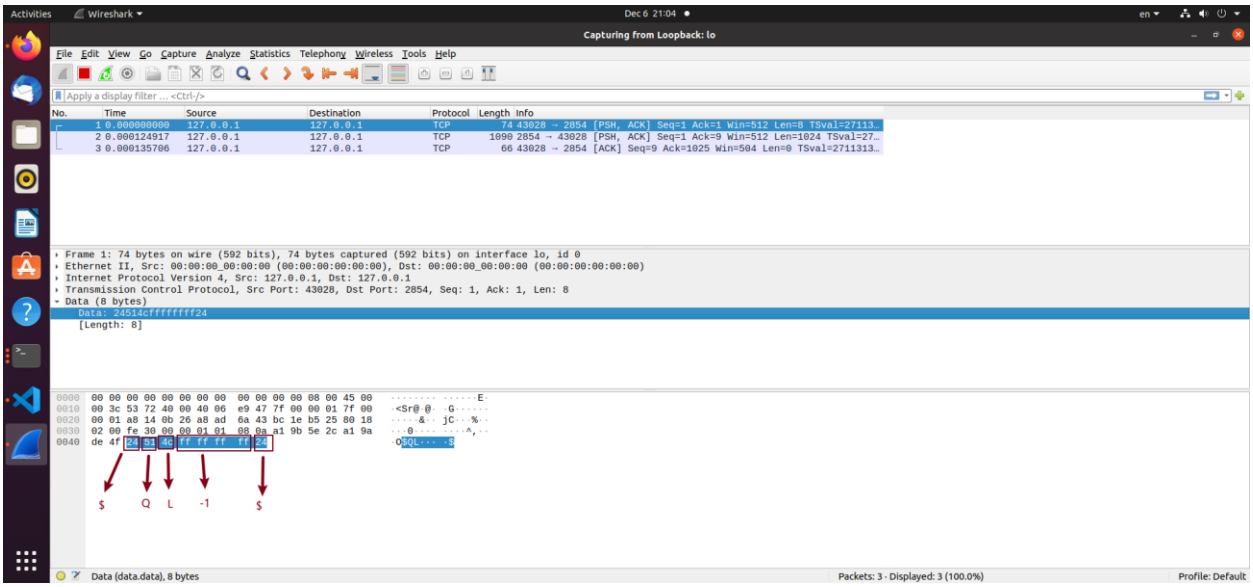
```
5
The client list received is:
[0] addr = 127.0.0.1, port = 51199
```

  服务端

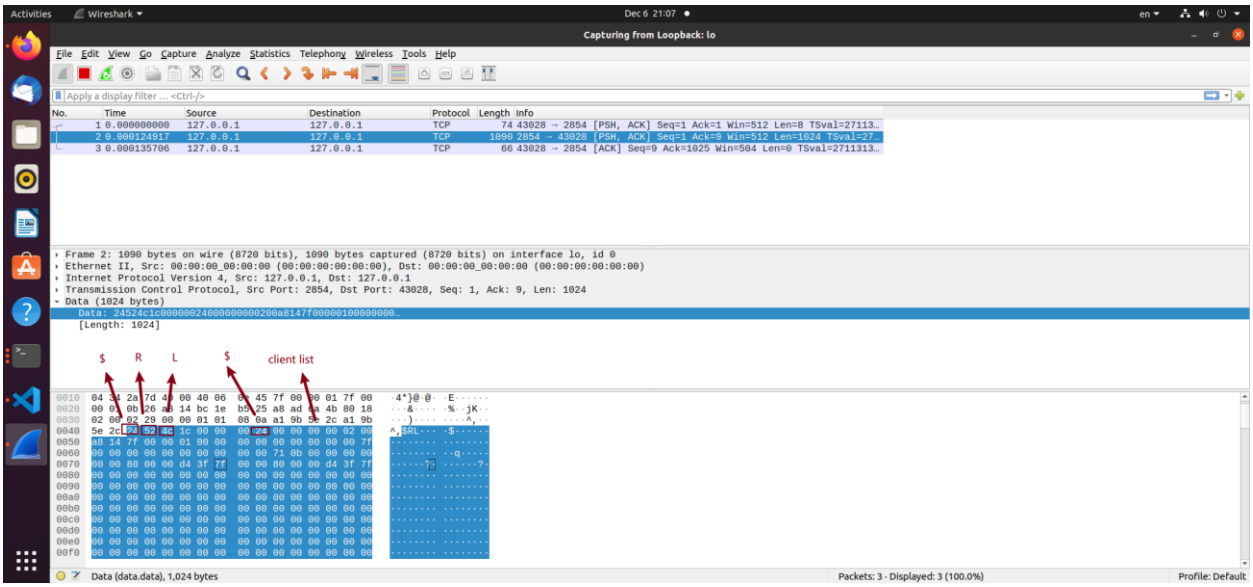
```
Server send to Client[0] the client list
```

  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）：

  请求数据包



响应数据包



相关的服务器的处理代码片段：

```
else if (*(receive_packet + 2) == 'L') {
    // build response packet
    // fill the header
    char *response = (char*)malloc(sizeof(char) * 4 + sizeof(int) + Client
    memset(response, 0, sizeof(char) * 4 + sizeof(int) + Client_num * (siz
    *response = '$';
    *(response + 1) = 'R';
    *(response + 2) = 'L';
    int length = (int)(sizeof(char) * 4 + sizeof(int) + Client_num * (size
    // printf("length = %d\n", length);// debug
    *(int*)(response + 3) = length;
    *((char*)((int *)(response + 3) + 1)) = '$';
    // fill the data
    char *p = (char*)((int *)(response + 3) + 1) + 1;
    for (int i = 0; i < Client_num; i++) {
        *((int*)p) = i;
        p = (char*)((int*)p + 1);
        *((struct sockaddr_in*)p) = client_list[i].addr;
        p = (char*)((struct sockaddr_in*)p + 1);
    }
    *p = 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：

```
6
Please enter the client list number
0
Please input what you want to send.
Hello World
target client number = 0
content = Hello World
```

服务器：

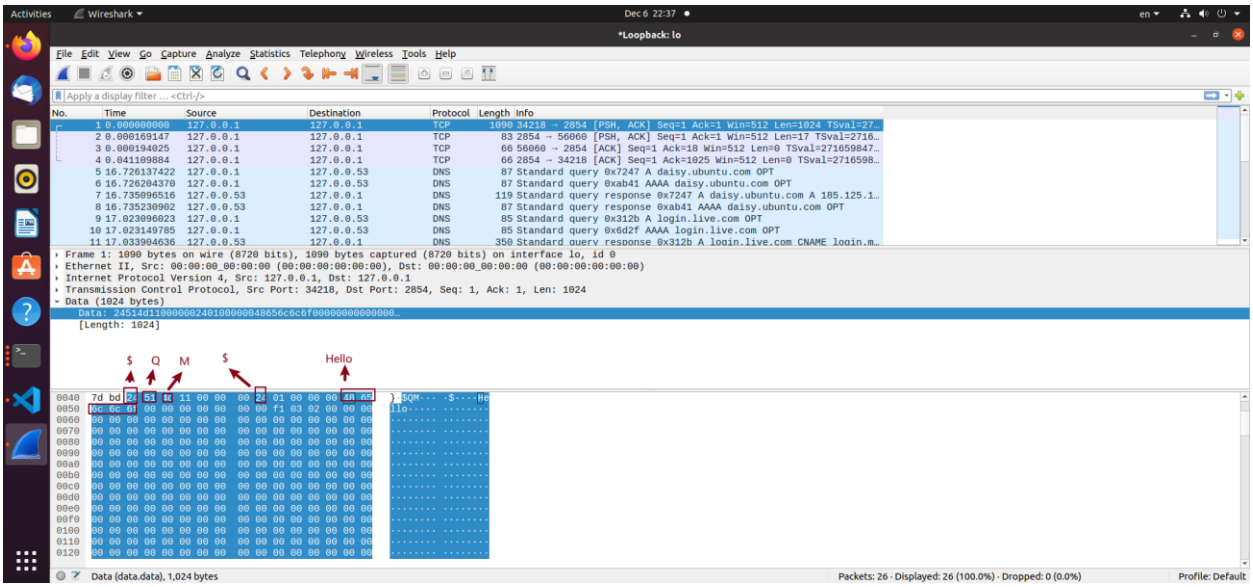
```
des_list_number = 0
content = Hello World
Client[1] send message to Client[0]
```

接收消息的客户端：

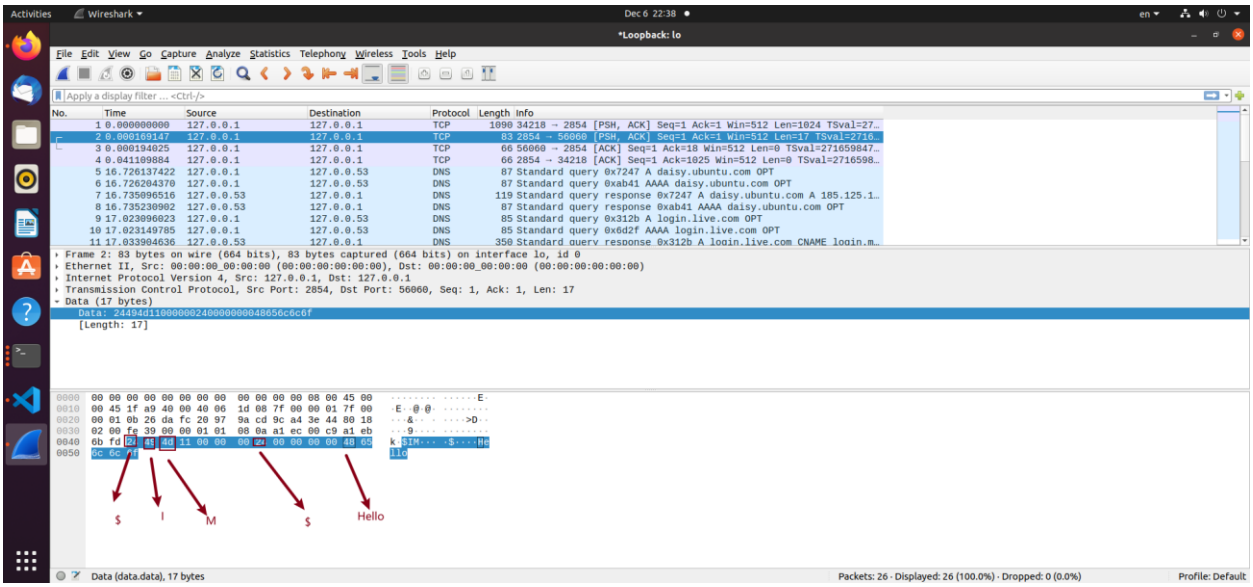
```
Client[1] send you message:
Hello World
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

请求数据包



指示数据包



相关的服务器的处理代码片段：

代码以及相关注释如下

```
else if (*(receive_packet + 2) == 'M') {
    // check some information
    // int des_list_number = *((int*)(receive_packet + 8));
    int des_list_number = *((int*)((char*)((int *)(receive_packet + 3) + 1
    printf("des_list_number = %d\n", des_list_number);

    // // build response packet
    int length = *(int*)(receive_packet + 3);
    // printf("%d\n", length);
    char *p = (char*)((int*)((char*)((int *)(receive_packet + 3) + 1) + 1)
    printf("content = %s\n", p);

    // check if the number exist
    if (des_list_number < 0 || des_list_number >= QUEUE_CONNECTION || clie
    // exist and connect
    else if (client_list[des_list_number].connected == 1 && client_list[de
    // exist but not connect
    else if (client_list[des_list_number].connected == 0 && client_list[de
    // free(response);
```

相关的客户端（发送和接收消息）处理代码片段：

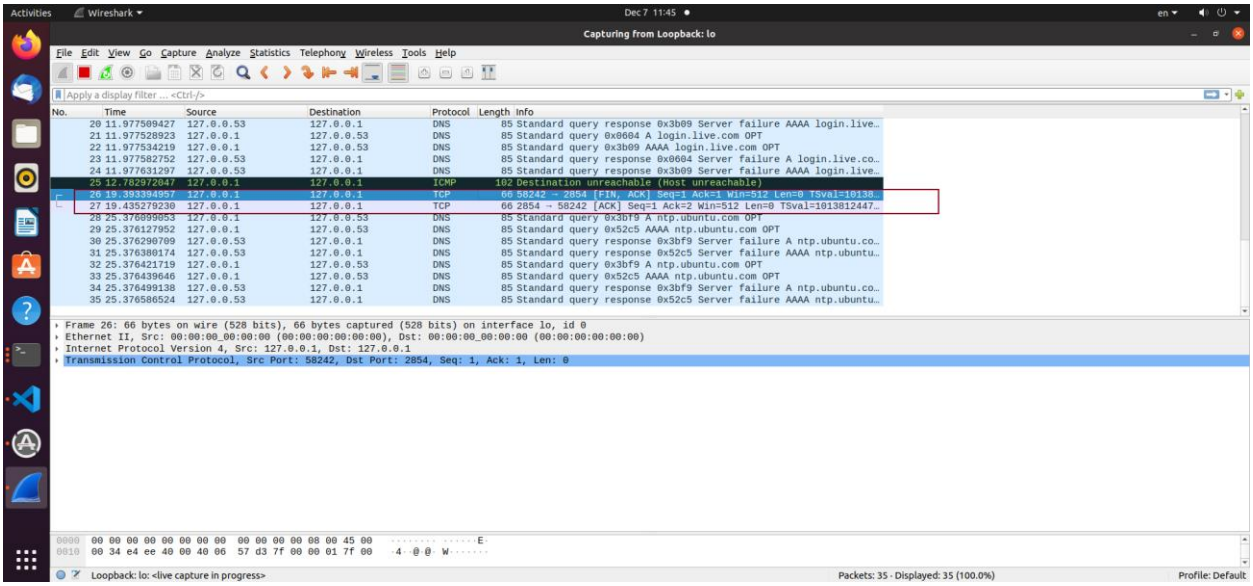
发送消息

```
    // build the packet
    int length = sizeof(int) + 4 * sizeof(char) + sizeof(int) + strlen(content
    char *request_packet = (char*)malloc(length * sizeof(char));
    *request_packet = '$';
    *(request_packet + 1) = 'Q';
    *(request_packet + 2) = 'M';// send message
    *(int*)(request_packet + 3) = length;
    *((char*)((int *)(request_packet + 3) + 1)) = '$';
    *((int*)((char*)((int *)(request_packet + 3) + 1) + 1)) = list_number;
    printf("target client number = %d\n", *((int*)((char*)((int *)(request_pac
    // printf("%d\n", *((int*)(request_packet + 8)));

    strcpy((char*)((int*)((char*)((int *)(request_packet + 3) + 1) + 1) + 1),
    printf("content = %s\n", (request_packet + 12));

    // send the message request packet
    if (send(client_socket, request_packet, BUFFER_SIZE, 0) < 0) {
        perror("Send message request failed\n");
    }
}
```

接受消息

```
// instruction packet
if (*response == '$' && *(response + 1) == 'I' && *((char*)((int*)(response
    if (*(response + 2) == 'M') {
        int list_number = *((int*)((char*)((int *)(response + 3) + 1) + 1))
        char *p = (char*)((int*)((char*)((int *)(response + 3) + 1) + 1) +
        printf("Client[%d] send you message:\n", list_number);
        // getchar();// debug
        // printf("%s", p);
        puts(p);
    }
}
```

- 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。



我通过 wireshark 抓包发现 TCP 发出连接释放消息。

服务端的 TCP 连接状态在较长时间内（10 分钟以上）没有发生发生变化，通过 wireshark 没有抓到相关包。

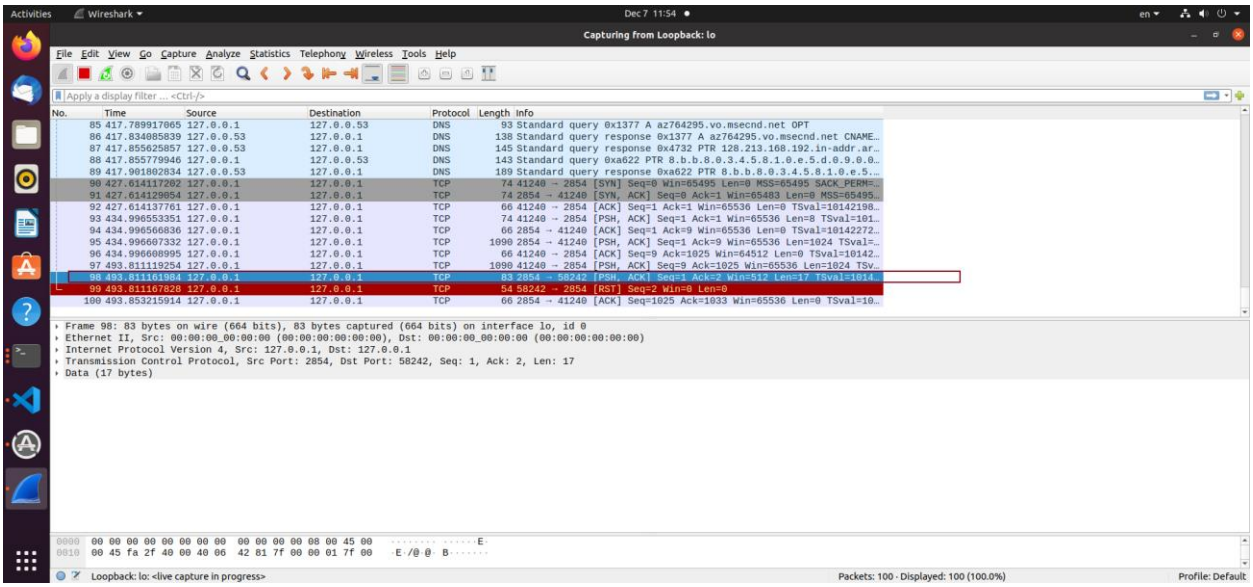
- 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

```
5
The client list received is:
[0] addr = 127.0.0.1, port = 51199
[1] addr = 127.0.0.1, port = 51217
[2] addr = 127.0.0.1, port = 51239
```

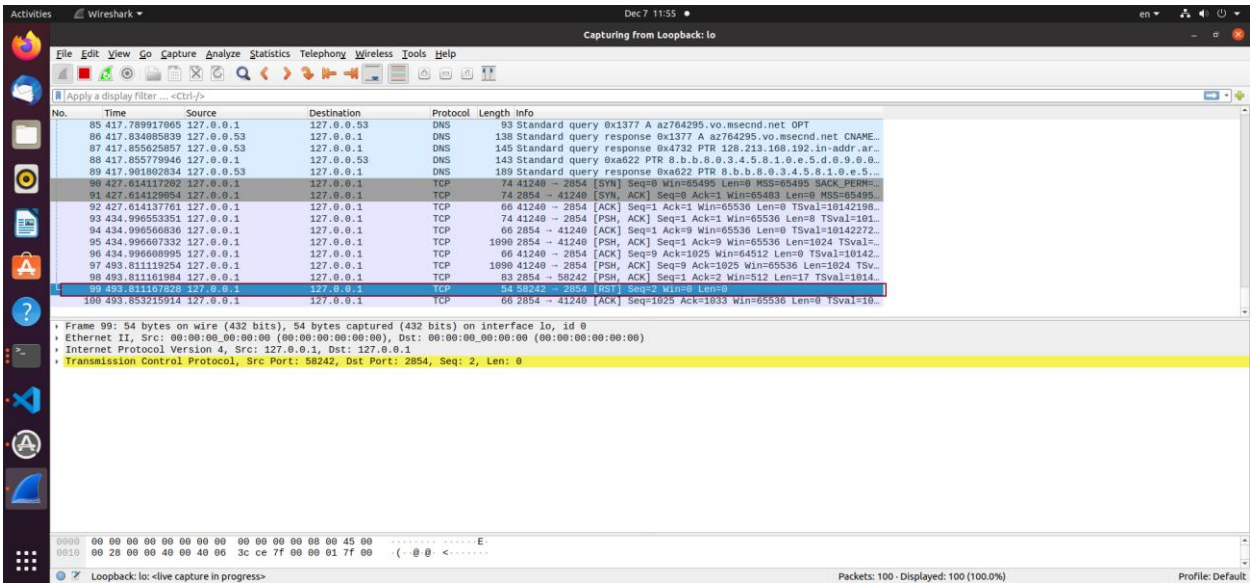
之前的连接还在

```
Connection accepted from 127.0.0.1:51239
Server: Connection succeed
Server send to Client[2] the client list
```

服务器会发送消息

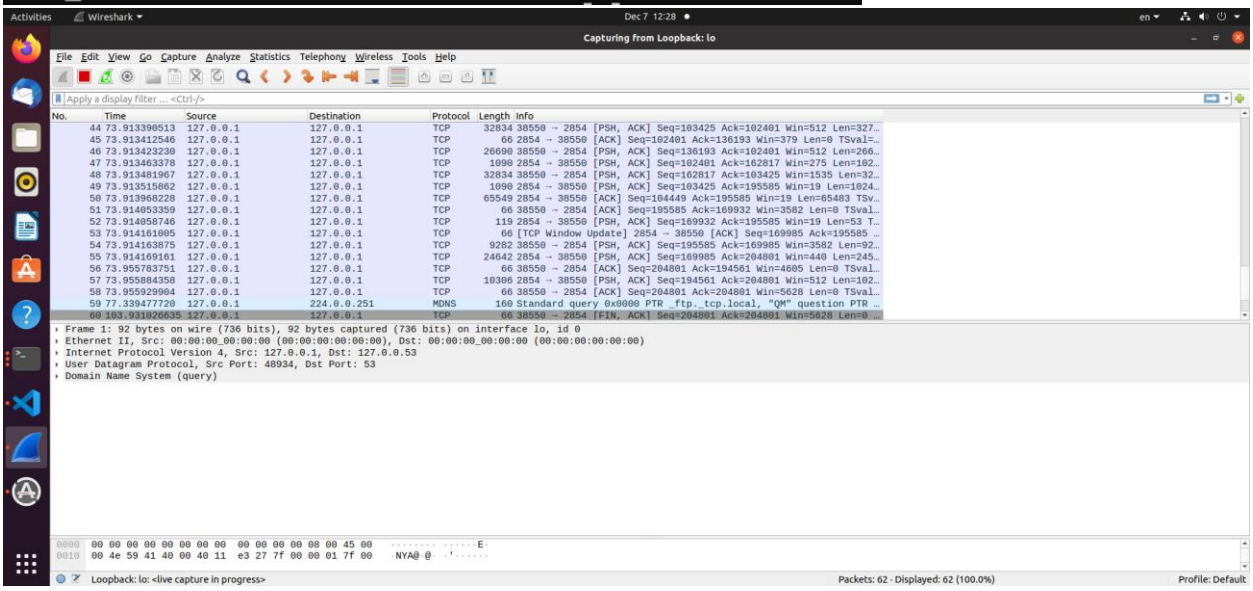


RST， 发送 RST 包关闭连接时，不必等缓冲区的包都发出去，直接就丢弃缓存区的包发送 RST. 而接收端收到 RST 包后，也不必发送 ACK 包来确认。



- 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。

```
time_cnt = 92 Server send to Client[3] the host time
time_cnt = 93 Server send to Client[3] the host time
time_cnt = 94 Server send to Client[3] the host time
time_cnt = 95 Server send to Client[3] the host time
time_cnt = 96 Server send to Client[3] the host time
time_cnt = 97 Server send to Client[3] the host time
time_cnt = 98 Server send to Client[3] the host time
time_cnt = 99 Server send to Client[3] the host time
time_cnt = 100 Server send to Client[3] the host time
```



我发现 wireshark 并没有抓到 100 个包，我观察到每个包的长度不同，可能是因为 wireshark 有的包的将几次数据拼接在了一起，并不是按照我发送的每次都发 1024 byte 大小。

在 TCP 中存在这样的机制。

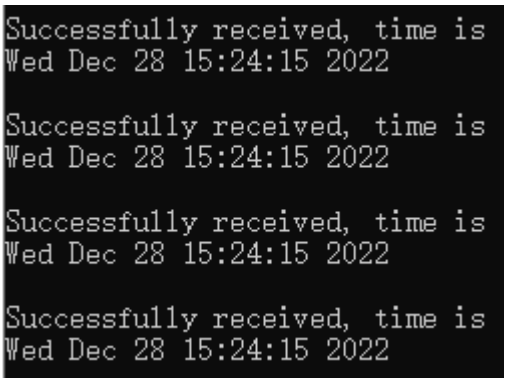
- 多个客户端同时连接服务器，同时发送时间请求（程序内自动连续调用 100 次 send），服务器和客户端的运行截图

```
time_cnt = 101 Server send to Client[4] the host time
time_cnt = 102 Server send to Client[3] the host time
```

Client0

```
Successfully received, time is
Wed Dec 28 15:24:15 2022

Successfully received, time is
Wed Dec 28 15:24:15 2022

Successfully received, time is
Wed Dec 28 15:24:15 2022

Successfully received, time is
Wed Dec 28 15:24:15 2022
```

Client1



六、 实验结果与分析

根据你编写的程序运行效果，分别解答以下问题（看完请删除本句）：

- 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？

  客户端不需要调用 bing 操作。服务器端要用 bind() 函数将套接字与特定的 IP 地址和端口绑定起来，只有这样，流经该 IP 地址和端口的数据才能交给套接字处理。类似地，客户端也要用 connect() 函数建立连接。客户端用的是 connect 而不是 bind。

  客户端的端口号是 connect()函数中产生的。

  当我们在客户端机上调用 connect 函数的时候，事实上会进入到内核的系统调用源码中进行执行。

  ```
  //进行 connect
   err = sock->ops->connect(sock, (struct sockaddr *)&address, addrlen,
       sock->file->f_flags);
  ```

  sock->ops->connect 其实调用的是 inet_stream_connect 函数。

```
//file: ipv4/af_inet.c
int inet_stream_connect(struct socket *sock, ...)
{
 ...
 __inet_stream_connect(sock, uaddr, addr_len, flags);
}

int __inet_stream_connect(struct socket *sock, ...)
{
 struct sock *sk = sock->sk;

 switch (sock->state) {
  case SS_UNCONNECTED:
   err = sk->sk_prot->connect(sk, uaddr, addr_len);
   sock->state = SS_CONNECTING;
   break;
 }
 ...
}
```

sk->sk_prot->connect 实际上对应的是 tcp_v4_connect 方法。

```
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
 //设置 socket 状态为 TCP_SYN_SENT
 tcp_set_state(sk, TCP_SYN_SENT);

 //动态选择一个端口
 err = inet_hash_connect(&tcp_death_row, sk);

 //函数用来根据 sk 中的信息，构建一个完成的 syn 报文，并将它发送出去。
 err = tcp_connect(sk);
}
```

在 tcp_v4_connect 中我们看到了选择端口的函数， inet_hash_connect。

```
int inet_hash_connect(struct inet_timewait_death_row *death_row,
        struct sock *sk)
{
 return __inet_hash_connect(death_row, sk, inet_sk_port_offset(sk),
   __inet_check_established, __inet_hash_nolisten);
}
```

inet_sk_port_offset(sk)：这个函数是根据要连接的目的 IP 和端口等信息生成一个随机数。

__inet_check_established：检查是否和现有 ESTABLISH 的连接是否冲突的时候用的函数。

如果 socket 重新申请的话，端口号是会发生改变的。

- 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？

  可以成功。

listen()之后就可以监听消息队列了，此时客户端调用 connected()是可以进入消息队列的，也就是连接成功，至于服务器什么时候将消息队列中的消息 accept()，就看服务器自己了。

- 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？

  是的。

- 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？

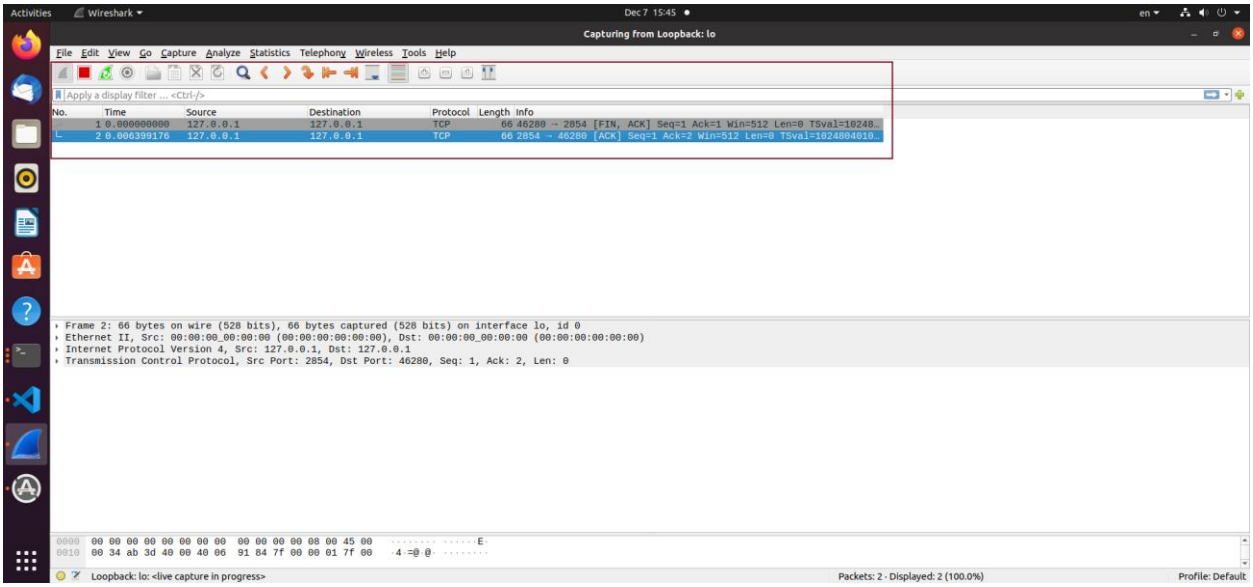
  通过不同的 sockaddr_in 来区分。

```
struct sockaddr_in {
  short int sin_family; /* AF_INET */
  unsigned short int sin_port; /* Port number */
  struct in_addr sin_addr; /* Internet address */
};
```

  sockaddr_in 对于中三个成员变量的三元组能唯一决定一个客户端。

- 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 连接断开



```
    else if (strcmp(buffer, "2") == 0) {
        if (connected == 1) {
            connected = 0;
            printf("Server: disconnected and exit\n");
            close(client_socket);
            break;
        }
        else {
            printf("No connection\n");
        }
    }
```

- 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？

Time	Source	Destination	Protocol	Length	Info
0.000000000	127.0.0.1	127.0.0.1	TCP	66	38334 → 2854 [FIN, ACK] Seq=1 Ack=1 Win=512 Len=0 TSval=10252…
0.002292147	127.0.0.1	127.0.0.1	TCP	66	2854 → 38334 [ACK] Seq=1 Ack=2 Win=512 Len=0 TSval=1025245078…

可以向客户端发送一个包来确认是否有效。

七、 讨论、心得

Winsock2.h vs winsock2.h

There is no difference between Winsock2.h and winsock2.h. Filenames are case-insensitive on typical Windows filesystems.

相关问题以及解决方案

读取消息不知道如何处理 fgets 读取的\n

Removing trailing newline character from fgets() input

https://stackoverflow.com/questions/2693776/removing-trailing-newline-character-from-fgets-input