

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： 计算机科学与技术

专 业： 计算机科学与技术

学 号：

指导教师： 黄正谦

2021 年 12 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： _____ 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

请求数据包	
数据包类型 4B REQUEST	命令类型 4B
数据包内容 最大1024B	

- 描述响应数据包的格式（画图说明），响应类型的定义

D	E
响应数据包	
数据包类型 4B RESPONSE	命令类型 4B
数据包内容 最大1024B	

- 描述指示数据包的格式（画图说明），指示类型的定义

指示数据包	
数据包类型 4B INSTRUCT	命令类型 4B
数据包内容 最大1024B	

- 客户端初始运行后显示的菜单选项

```
cdh@cdh-virtual-machine:~/network_repository/enigma/lab7_copy$ ./client
Welcome to Socket client program!
1 -> Connect to server
2 -> Quit
Enter(number):
```

- 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）

```
int main() {
    int choice;

    maincmd();
    while (1) {
        cout << BLUE "client> " CLR;
        cin >> choice;
        // check if it is valid
        while (choice != 1 && choice != 2) {
            cout << RED "Invalid input and please enter your choice again!\n" CLR;
            cout << BLUE "client> " CLR;
            cin >> choice;
        }

        // analyze input choice
        if (choice == 2)
        {
            cout << "Exit\n";
            exit(0);
        }
        else if(choice == 1)
        {
            connecting();
        }
    }
}
```

首先调用 maincmd 函数，打印命令界面，然后根据用户选择连接服务器

```

// cmd
commandline();
while (1) {
    cout << BLUE "client> " CLR;
    cin >> choice;
    switch (choice) {
        case 1:
            pthread_cancel(p);
            request_command(sockfd, DISCONNECT);
            break;
        case 2:
        case 3:
        case 4:
            request_command(sockfd, choice);
            break;
        case 5:
            send_message(sockfd);
            break;
        case 6:
            pthread_cancel(p);
            request_command(sockfd, DISCONNECT);
            printf("Quit!\n");
            return;
        default:
            printf("Invalid Input!\n");
            continue;
    }
}

```

服务器连接后进入命令行界面循环，根据用户输入选择向服务器发送的数据包

- 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）

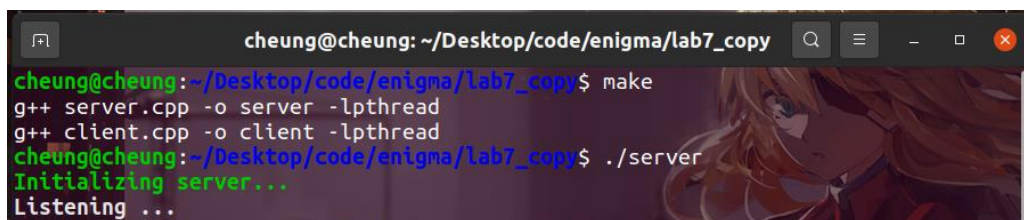
注册该函数作为一个监听线程，收到数据包后就打印出来

```

5 // listen to server
5 void *server_confirm(void *sockfd) {
7     packet p;
3     while (1) {
3         recv(*(int *) sockfd, (char *) &p, sizeof(p), 0); // get packet
3
1         // if there is disconnect
2         if (p.type == DISCONNECT && p.packet_type == INSTRUCT) {
3             printf("\nServer is offline!\n");
4             pthread_exit(0); // exit this thread
5         }
5
7         // otherwise print the content
3         printf("%s\n", p.content);
3     }
3 }
.

```

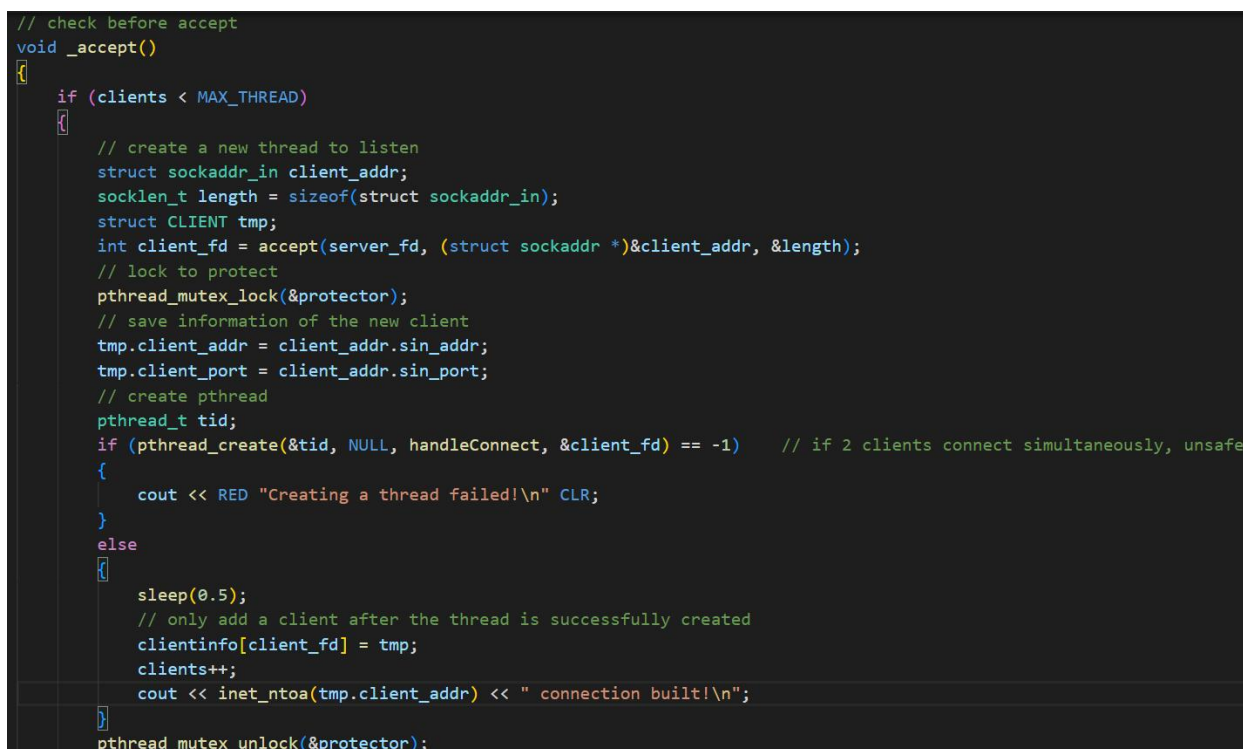

- 服务器初始运行后显示的界面



```
cheung@cheung: ~/Desktop/code/enigma/lab7_copy
cheung@cheung:~/Desktop/code/enigma/lab7_copy$ make
g++ server.cpp -o server -lpthread
g++ client.cpp -o client -lpthread
cheung@cheung:~/Desktop/code/enigma/lab7_copy$ ./server
Initializing server...
Listening ...
```

- 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）

主线程循环调用该函数，如果有客户发起连接请求，则为其创建子线程。创建成功后再将客户加入客户池。



```
// check before accept
void _accept()
{
    if (clients < MAX_THREAD)
    {
        // create a new thread to listen
        struct sockaddr_in client_addr;
        socklen_t length = sizeof(struct sockaddr_in);
        struct CLIENT tmp;
        int client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &length);
        // lock to protect
        pthread_mutex_lock(&protector);
        // save information of the new client
        tmp.client_addr = client_addr.sin_addr;
        tmp.client_port = client_addr.sin_port;
        // create pthread
        pthread_t tid;
        if (pthread_create(&tid, NULL, handleConnect, &client_fd) == -1) // if 2 clients connect simultaneously, unsafe
        {
            cout << RED "Creating a thread failed!\n" CLR;
        }
        else
        {
            sleep(0.5);
            // only add a client after the thread is successfully created
            clientinfo[client_fd] = tmp;
            clients++;
            cout << inet_ntoa(tmp.client_addr) << " connection built!\n";
        }
        pthread_mutex_unlock(&protector);
    }
}
```

- 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）

连接建立成功后，首先发送一个 hello。然后循环调用接受包函数 rscv()。如果接受失败则输出，否则交由另外一个函数分析包的内容并做处理。

```

// the entry of a new thread
void *handleConnect(void *clifd)
{
    int fd = *(int *)clifd;
    packet p;

    // send hello
    send_hello(fd);

    while (1)
    {
        if (recv(fd, (char *)&p, sizeof(p), 0) < 0)
        {
            cout << RED "Receiving from " << inet_ntoa(clientinfo[fd].client_addr) << " failed!\n" CLR;
            break;
        }
        if (handlePacket(&p, fd) == -1)
        {
            return NULL;
        }
    }
    return NULL;
}

```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

客户端：

```

Welcome to Socket client program!
1 -> Connect to server
2 -> Quit
Enter(number):
client> 1
Please enter server IP: 10.192.181.22
Please enter server port: 2203
Connection success!
Select your command!
1 -> Disconnect
2 -> Get server time
3 -> Get server user name
4 -> Get client list
5 -> Send message to other clients
6 -> Quit
Enter(number):
client> hello from server 0.0.0.0!

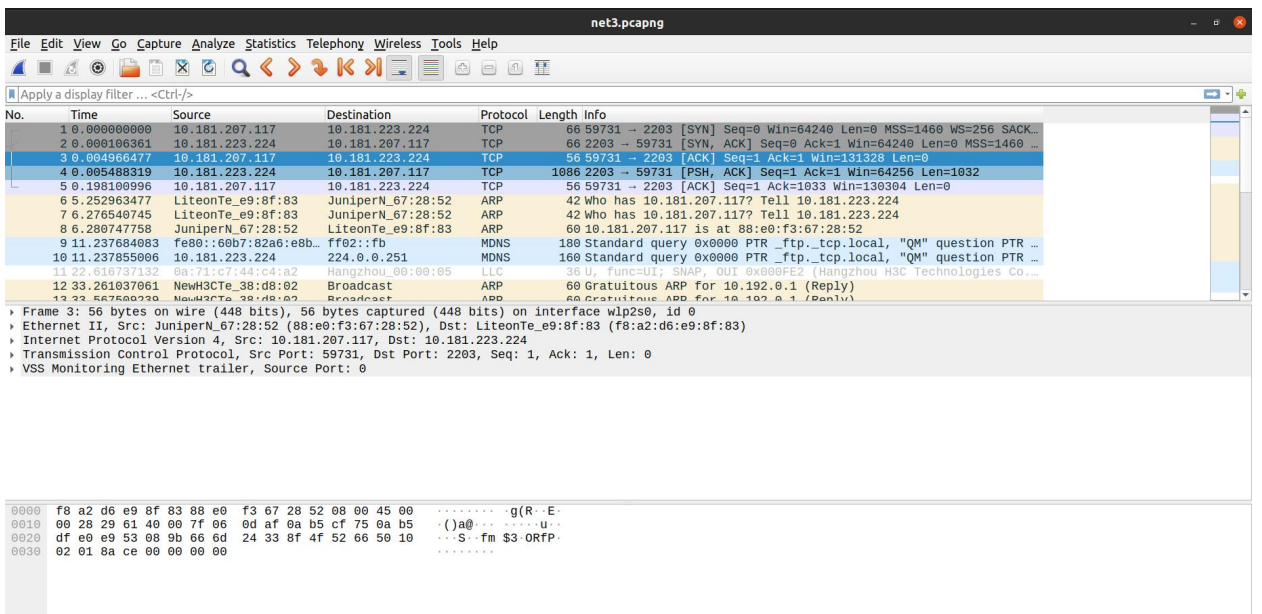
```

服务端：

（因为写代码的时候没有发现这里的实验要求看服务端的显示内容且我们当时不认为服务器有必要输出内容，我们没有给服务端编写任何输出的代码。虽然没有输出，但是从客户端和wireshark抓包的内容可以看出我们的代码是正确的。）


```
cheung@cheung: ~/Desktop/code/enigma/lab7_copy
cheung@cheung:~/Desktop/code/enigma/lab7_copy$ make
g++ server.cpp -o server -lpthread
g++ client.cpp -o client -lpthread
cheung@cheung:~/Desktop/code/enigma/lab7_copy$ ./server
Initializing server...
Listening ...
10.181.207.117 connection built!
```

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：

```
2
client> Date:2021/12/30 Time:21:55:50
```

服务端：

- 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：

```

3
client> cheung

```

服务端：

```

cheung@cheung: ~/Desktop/code/enigma/lab7_copy
cheung@cheung: ~/Desktop/code/enigma/lab7_copy$ ./server
Initializing Server...
Listening ...
10.181.207.117 connection built!

```

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）：

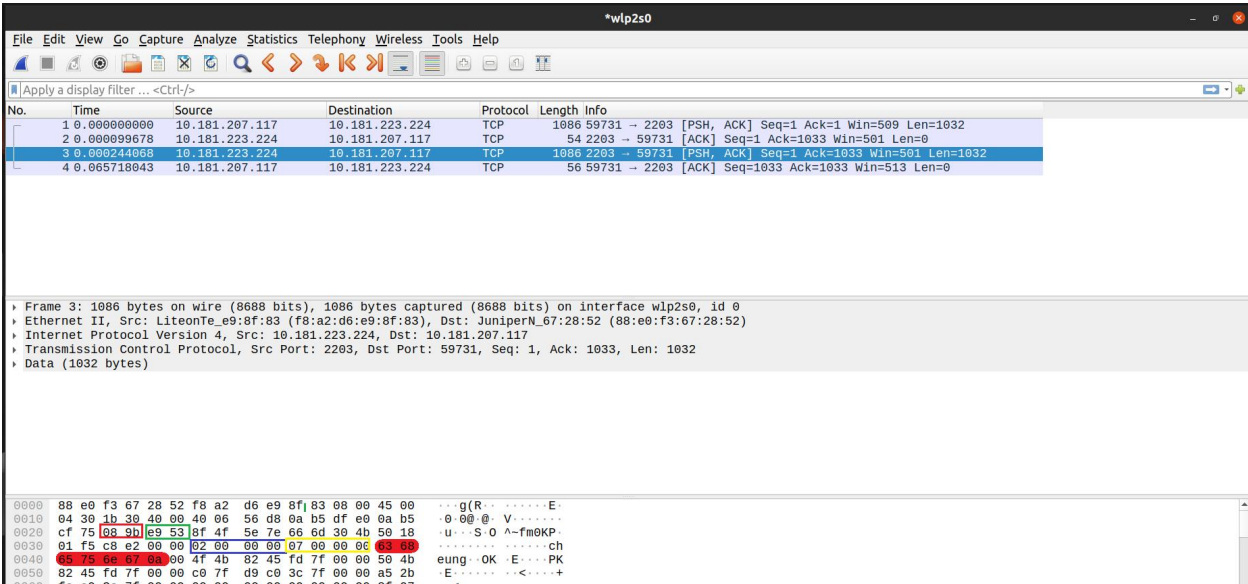
红框：源端口

绿框：目的端口

蓝框：自定义的数据包类型是 2，对应回复

黄框：自定义的命令类型是 7，对应成功

红底：名字数据字符串



相关的服务器的处理代码片段：

利用函数获取主机名，然后设置好数据包并发送

```

void send_hostname(int fd)
{
    packet p;
    char hostname[HOSTNAME_LEN];
    gethostname(hostname, sizeof(hostname));
    sprintf(p.content, "%s\n", hostname);
    p.packet_type = RESPONSE;
    p.type = SUCCESS;
    send(fd, (char *)&p, sizeof(p), 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。

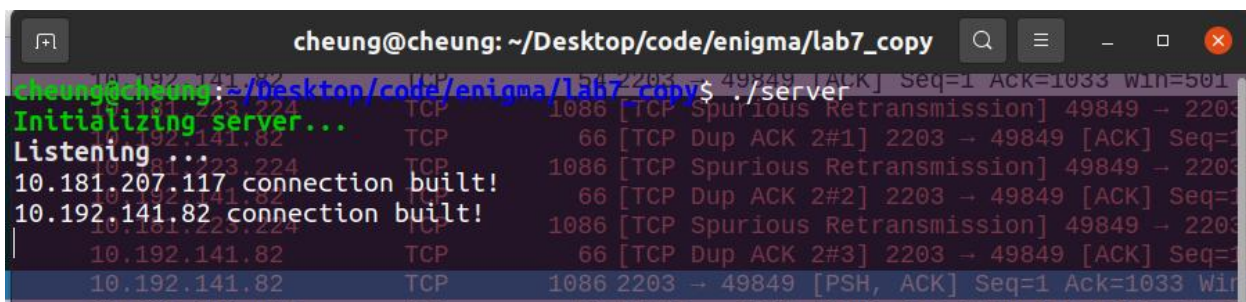
客户端：

```

Client sockfd:4 Port:21481 IP addr:10.181.207.117
Client sockfd:5 Port:47554 IP addr:10.192.141.82

```

服务端：



The terminal window shows the server being initialized and listening. It then receives two connections from 10.181.207.117 and 10.192.141.82. The background shows Wireshark capturing network traffic.

```

cheung@cheung: ~/Desktop/code/enigma/lab7_copy
cheung@cheung: ~/Desktop/code/enigma/lab7_copy$ ./server
Initializing server...
Listening...
10.181.207.117 connection built!
10.192.141.82 connection built!

```

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）：

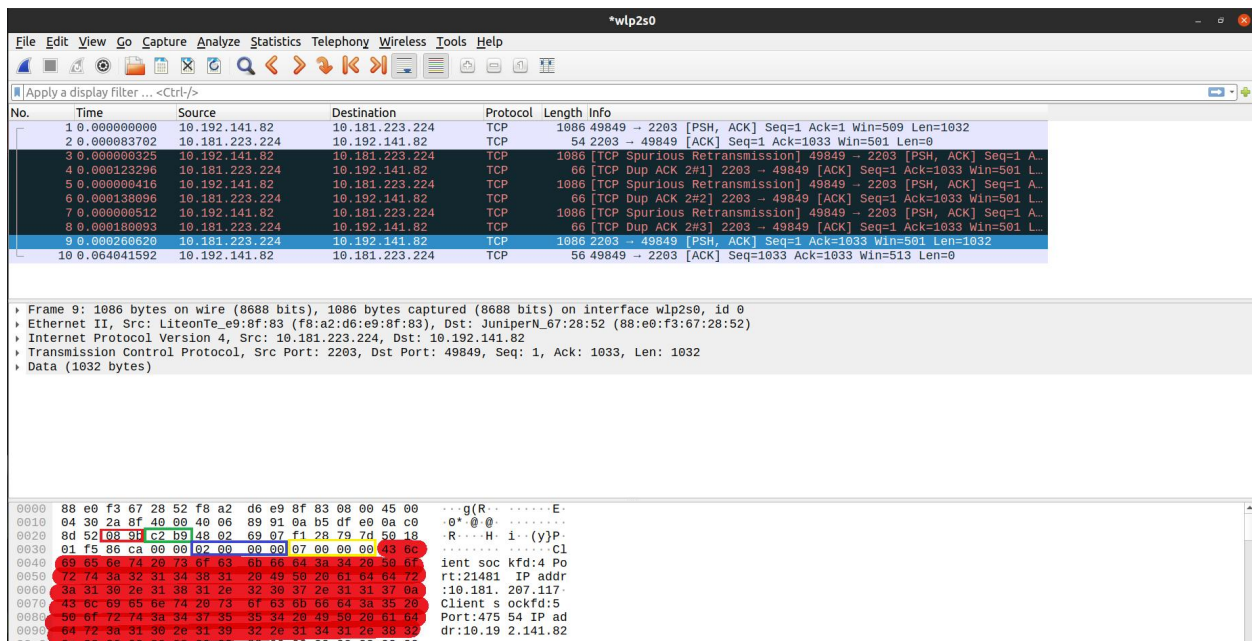
红框：源端口

绿框：目的端口

蓝框：自定义的数据包类型是 2，对应回复

黄框：自定义的命令类型是 7，对应成功

红底：客户端列表字符串



相关的服务器的处理代码片段：

首先设置数据包的属性，然后获取所有客户端的信息输出到数据包中再发送

```
void send_client_list(int fd)
{
    packet p;
    p.packet_type = RESPONSE;
    p.type = SUCCESS;

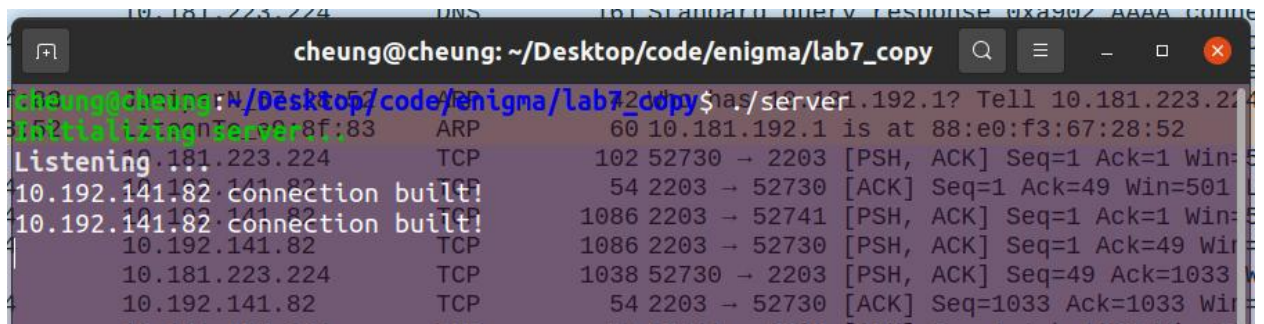
    pthread_mutex_lock(&protector);
    int j = 0;
    for (map<int, struct CLIENT>::iterator it = clientinfo.begin(); it != clientinfo.end(); it++)
    {
        j += sprintf(p.content + j, "Client sockfd:%d ", it->first);
        j += sprintf(p.content + j, "Port:%d ", it->second.client_port);
        j += sprintf(p.content + j, "IP addr:%s\n", inet_ntoa(it->second.client_addr));
    }
    pthread_mutex_unlock(&protector);
    send(fd, (char *)&p, sizeof(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：

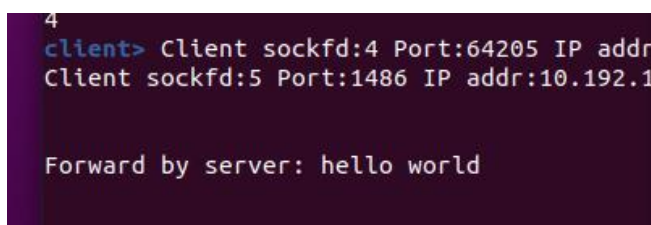
```
5
Enter Client ID: 5
Enter message: hello world
Sending message to client 5 ...Success!
client>
```

服务器:



A terminal window titled 'cheung@cheung: ~/Desktop/code/enigma/lab7_copy'. The prompt is 'cheung@cheung: ~/Desktop/code/enigma/lab7_copy\$'. The user enters './server'. The output shows 'Starting Server...', 'Listening...', and two 'connection built!' messages from 10.192.141.82. In the background, Wireshark network traffic is visible, including ARP requests and TCP connections between 10.181.223.224 and 10.192.141.82.

接收消息的客户端:



A terminal window showing client output. It displays 'Client sockfd:4 Port:64205 IP addr:' and 'Client sockfd:5 Port:1486 IP addr:10.192.1'. Below this, it says 'Forward by server: hello world'.

Wireshark 抓取的数据包截图（发送和接收分别标记）:

接收:

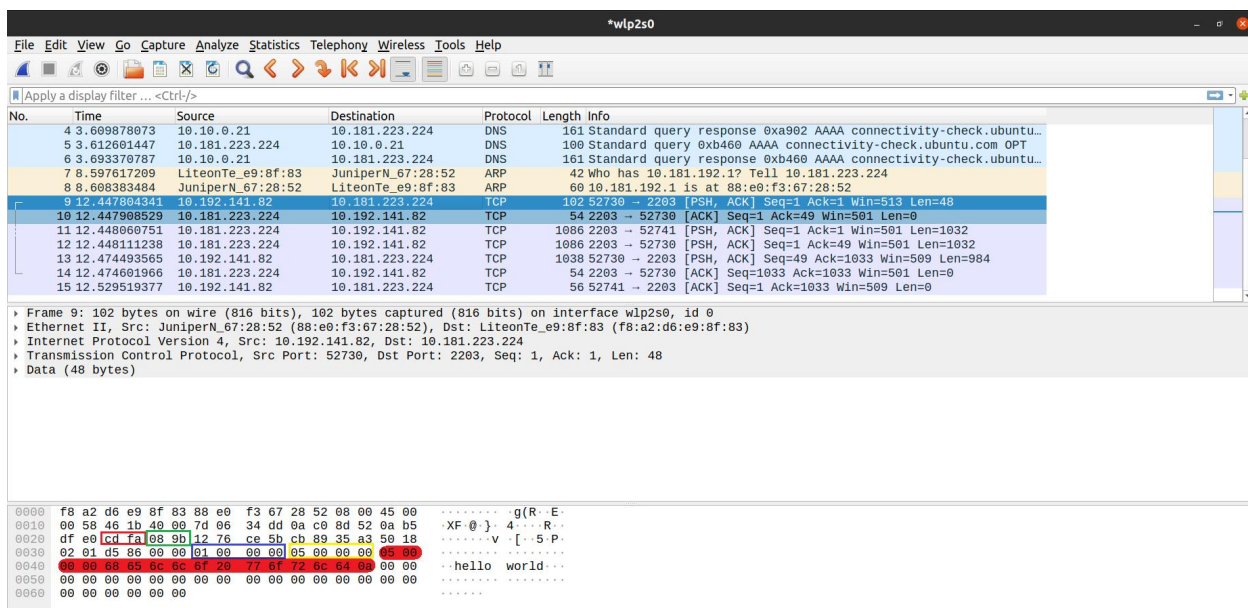
红框: 源端口

绿框: 目的端口

蓝框: 自定义的数据包类型是 1, 对应请求

黄框: 自定义的命令类型是 5, 对应发送给客户端

红底: 接收方的 ID（文件描述符）和通信内容字符串



发送：

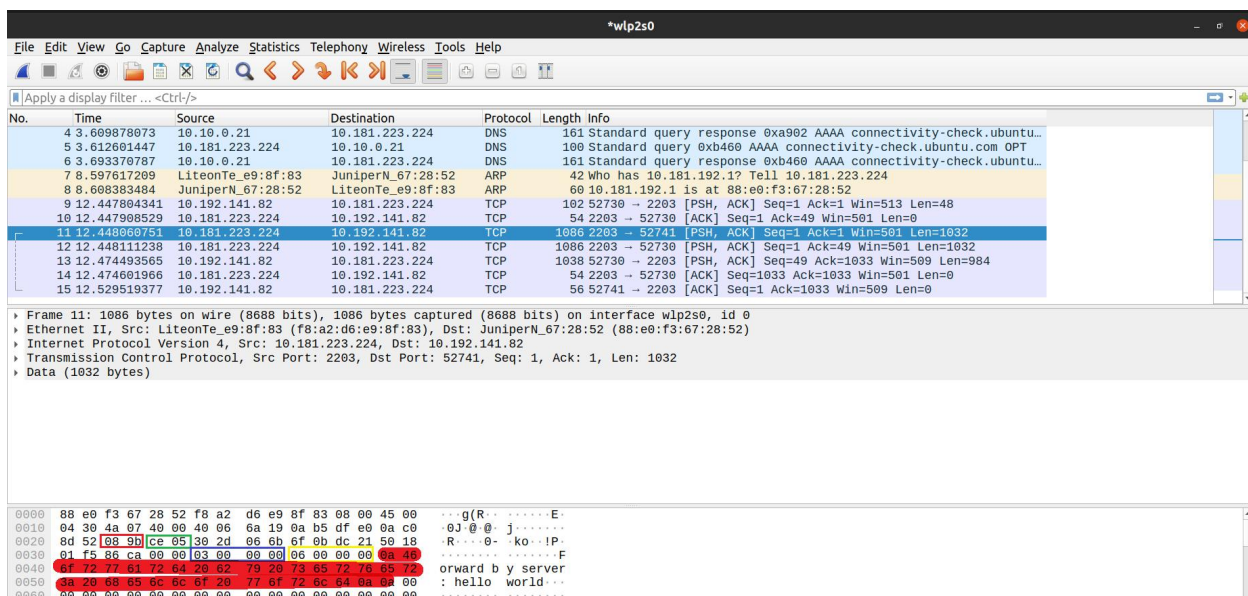
红框：源端口

绿框：目的端口

蓝框：自定义的数据包类型是 3，对应指示

黄框：自定义的命令类型是 6，对应转发

红底：通信内容字符串



相关的服务器的处理代码片段：

首先检查该信息是否发送给发送者的，如果是，则直接回发指示错误的数据包。

```

void send_message(packet *p, int fd)
{
    packet sp;

    // obtain the destination
    int des_fd = *((int *)p->content);
    pthread_mutex_lock(&protector);
    if(des_fd == fd){
        // cannot send message to itself
        sp.packet_type = RESPONSE;
        sp.type = ERROR;
        send(fd, (char *)&sp, sizeof(sp), 0);
    }
    pthread_mutex_unlock(&protector);
    // destination is connected
    if (clientinfo.count(des_fd))
    {
        sp.packet_type = INSTRUCT;
        sp.type = FORWARD;
        sprintf(sp.content, "\nForward by server: %s", p->content);
    }
}

```

如果接收方是已连接上的客户端，则转发，否则回发一条指示错误的数据包。

```

204     send(fd, (char *)&sp, sizeof(sp), 0);
205 }
206 pthread_mutex_unlock(&protector);
207 // destination is connected
208 if (clientinfo.count(des_fd))
209 {
210     sp.packet_type = INSTRUCT;
211     sp.type = FORWARD;
212     sprintf(sp.content, "\nForward by server: %s\n", p->content + sizeof(int));
213     send(des_fd, (char *)&sp, sizeof(sp), 0);
214     packet r_packet;
215     r_packet.packet_type = RESPONSE;
216     r_packet.type = SUCCESS;
217     send(fd, (char *)&r_packet, sizeof(r_packet), 0);
218 }
219 else
220 {
221     sp.packet_type = RESPONSE;
222     sp.type = ERROR;
223     sprintf(sp.content, "Destination is not connected\n");
224     send(fd, (char *)&sp, sizeof(sp), 0);
225 }
226 }

```

相关的客户端（发送和接收消息）处理代码片段：

发送消息：

```

// send msg to other client
void send_message(int sockfd) {
    // TODO: use IP instead of fd?
    int des_client_id;
    packet p;
    p.packet_type = REQUEST;
    p.type = SEND_TO_CLIENT;
    // input ID
    printf("Enter Client ID: ");
    scanf("%d", &des_client_id);
    memcpy(p.content, &des_client_id, sizeof(int));
    // input msg
    printf("Enter message: ");
    getchar(); // clear buffer
    fgets(p.content + sizeof(des_client_id), MAX_MESSAGE_SIZE - sizeof(des_client_id), stdin); // 注意这里后移一个 int
    // no \0
    printf("Sending message to client %d ...", des_client_id);
    send(sockfd, (char *) &p, sizeof(p), 0);
    printf("Success!\n");
}

```

接收消息：

```

// listen to server
void *server_confirm(void *sockfd) {
    packet p;
    while (1) {
        recv(*(int *) sockfd, (char *) &p, sizeof(p), 0); // get packet

        // if there is disconnect
        if (p.type == DISCONNECT && p.packet_type == INSTRUCT) {
            printf("\nServer is offline!\n");
            pthread_exit(0); // exit this thread
        }

        // otherwise print the content
        printf("%s\n", p.content);
    }
}

```

- 拔掉客户端的网线，然后退出客户端程序。观察客户端的 TCP 连接状态，并使用 Wireshark 观察客户端是否发出了 TCP 连接释放的消息。同时观察服务端的 TCP 连接状态在较长时间内（10 分钟以上）是否发生变化。

客户端的 TCP 连接状态无变化，且使用 Wireshark 观察发现客户端未发送 TCP 连接释放的消息，服务端的 TCP 连接在十分钟以上也没有断开。

- 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？

重新运行连接后，查看客户端列表，显示之前异常退出的连接依然存在。给之前异常退出的客

户端发送消息时，对应客户端无法接受到消息。

- 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。

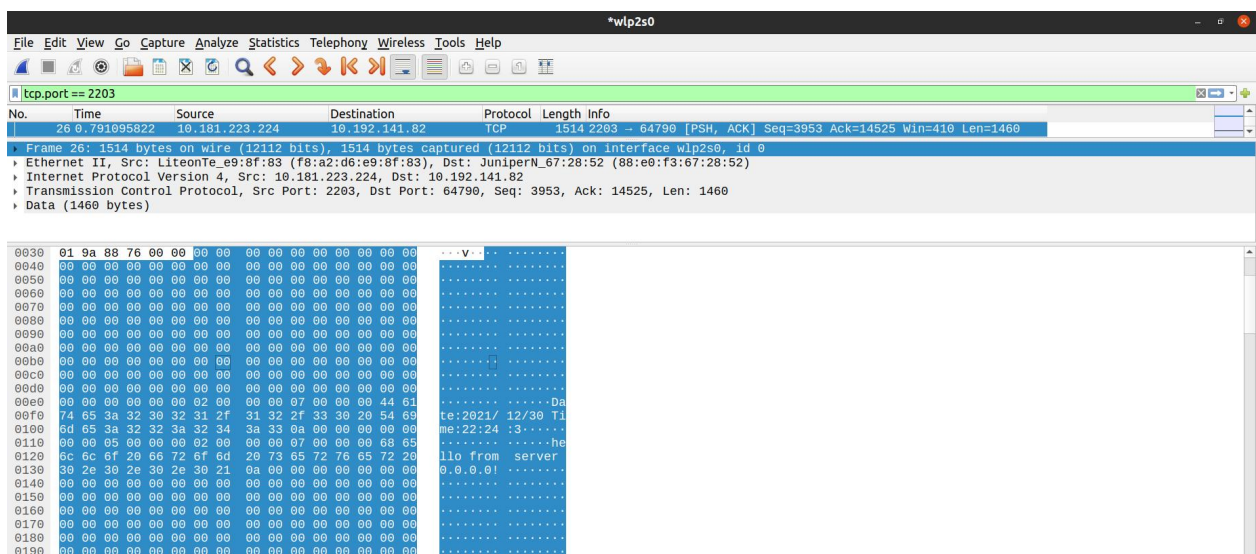
客户端收到消息：

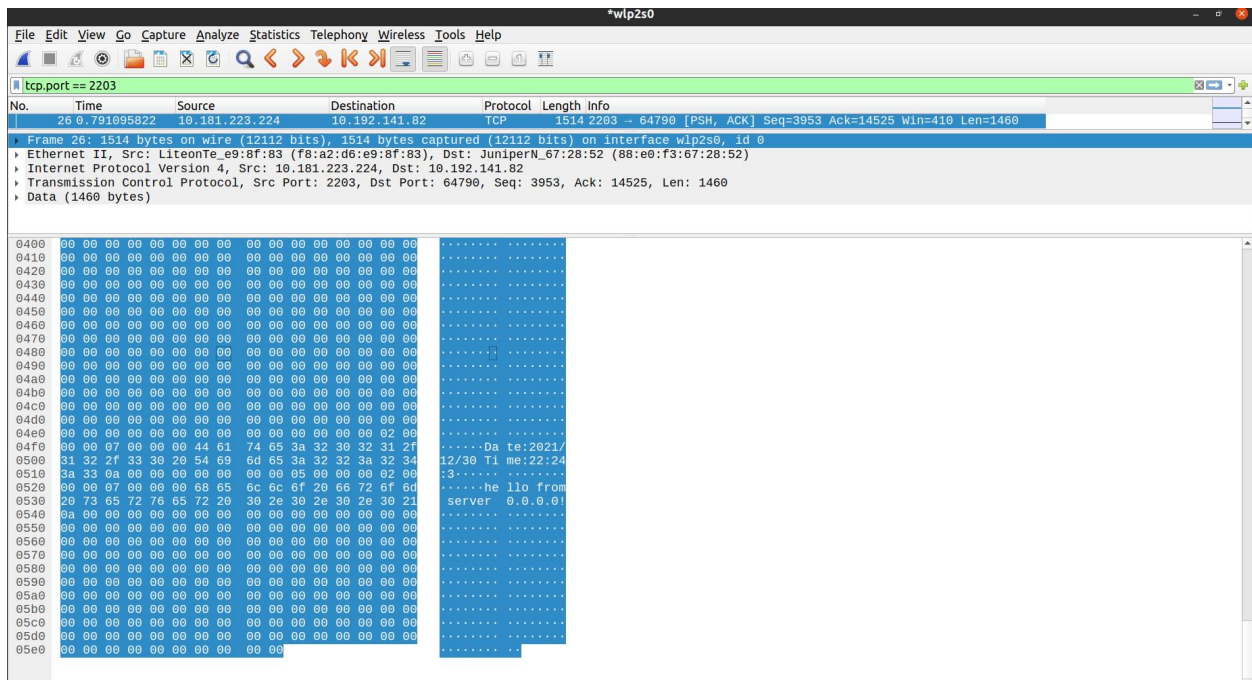
```
2
client> Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
```

```
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
Date:2021/12/30 Time:22:24:3
```

抓包：

服务器正常处理了 100 次请求，客户端实际接收的数据包少于 100 个。这一现象是 TCP 的粘包现象。查看响应数据包，可以看到里面有多于一次的相应数据。





- 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

客户端 1:

```

6 -> Quit
Enter(number):
client> hello from server 0.0.0.0!

3
client> cheung

cheung

cheung

cheung

```

客户端 2:

```
cdh@cdh-virtual-machine:~$ cat server.c
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     int fd = socket(AF_INET, SOCK_STREAM, 0);
7     if (fd < 0) {
8         perror("socket");
9         return 1;
10    }
11    struct sockaddr_in server_addr;
12    server_addr.sin_family = AF_INET;
13    server_addr.sin_port = htons(8080);
14    inet_pton(AF_INET, "0.0.0.0", &server_addr.sin_addr);
15    if (bind(fd, &server_addr, sizeof(server_addr)) < 0) {
16        perror("bind");
17        return 1;
18    }
19    if (listen(fd, 5) < 0) {
20        perror("listen");
21        return 1;
22    }
23    printf("Server is listening on 0.0.0.0:8080\n");
24    while (1) {
25        struct sockaddr_in client_addr;
26        socklen_t len = sizeof(client_addr);
27        int connfd = accept(fd, &client_addr, &len);
28        if (connfd < 0) {
29            perror("accept");
30            continue;
31        }
32        char buf[1024];
33        int n = read(connfd, buf, sizeof(buf));
34        if (n < 0) {
35            perror("read");
36            close(connfd);
37            continue;
38        }
39        printf("Received: %s\n", buf);
40        char response[1024];
41        sprintf(response, "HTTP/1.1 200 OK\nContent-Type: text/plain\n\n%s", buf);
42        n = write(connfd, response, strlen(response));
43        if (n < 0) {
44            perror("write");
45            close(connfd);
46            continue;
47        }
48        close(connfd);
49    }
50    return 0;
51 }
```

```
cdh@cdh-virtual-machine:~$ ./server
Server is listening on 0.0.0.0:8080
client> hello from server 0.0.0.0!
3
client> cheung
cheung
cheung
```

服务端:

```
cheung@cheung: ~/Desktop/code/enigma/lab7_copy
cheung@cheung:~/Desktop/code/enigma/lab7_copy$ ./server
Initializing server...
Listening ...
10.192.141.82 connection built!
10.192.141.82 connection built!
10.192.141.82 connection built!
Receiving from 10.192.141.82 failed!
^Ccheung@cheung:~/Desktop/code/enigma/lab7_copy$ ./server
Initializing server...
Listening ...
10.192.141.82 connection built!
10.192.141.82 connection built!
10.192.141.82 connection built!
10.192.141.82 connection built!
```

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？

客户端不需要调用 bind，它的端口是系统自动分配的，每次调用 connect 端口都会发生变化

- 假设在服务端调用 listen 和调用 accept 之间设了一个调试断点，暂停在此断点时，此时客户端调用 connect 后是否马上能连接成功？

可以连接成功

- 连续快速 send 多次数据后，通过 Wireshark 抓包看到的发送的 Tcp Segment 次数是否和 send 的次数完全一致？

不一致，TCP 通信时，发送端在多次发送间隔较小，数据量较小的数据包时，会自动合并为一个数据包，出现所谓的“粘包”现象

- 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？

通过客户端的 IP 和端口判断，并且不同的客户端在服务器内属于不同的监听线程

- 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）

是 TIME_WAIT 状态，该状态能保持大约一分钟

- 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？

服务器的 TCP 连接状态没有变化，仍然是 ESTABLISHED。服务器可以使用保活机制来检测连接的有效性，通过间隔一段时间发送探测报文，来检测 TCP 连接状态是否正常

七、 讨论、心得

socket 编程本身并不难，掌握一套基本的流程即可。在两台电脑通信时，要处理好虚拟机的网络问题，确保两台主机可以正常通信。