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：	南梓涵、朱镐哲
学　　院：	计算机学院
系：	信息安全
专　　业：	信息安全
学　　号：	3210106024、3210103283
指导教师：	韩劲松

2023 年 12 月 17 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：南梓涵、朱镐哲　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/C++语言形式的 Socket API**
- **本实验可组队完成，但每组最多 2 人，当然，更欢迎 solo 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：若组队的话，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。需要有较为丰富的注释。**
- **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。需要配有简易运行说明文档。**

- 描述请求数据包的格式（画图说明），请求类型的定义

每一个请求数据包的结构都是“验证标志（string）+请求类型（char）+target_id（char，只在请求类型为‘s’的时候有效）+message（string，只在请求类型为‘s’的时候有效）”：

flag(7字节)	type(1字节)	target_id(1字节)	message(n字节)
”zhz&nzh“	char	char，只在请求类型为‘s’的时候有效，默认为'1'。	string，只在请求类型为‘s’的时候有效，默认为"1"。

对应的类如下代码所示，可以看到当没有传入 target_id 和 message 时，init_packet 函数会自动进行默认的填充，to_string 函数是用于将 MyPacket 类型转化为字符流，用于传输：

```
class MyPacket
{
  private:
```

```
    char type; //消息类型
    char target; //发送目标
    std::string message; //发送内容

  public:
    //省略构造函数
    void init_packet(char type, std::string data = "1", char target = '1'); //初始化 MyPacket
    //省略接口函数
    std::string to_string();  //将 MyPacket 转换为 string
};
```

通过观察 to_string 函数和 MyPacket 类成员，我们可以清晰地看到数据包的结构：

```
std::string MyPacket::to_string()
{
  std::string res = PACKETFLAG;
  res.push_back(type);
  res.push_back(target);
  res.append(message);
  return res;
}
```

有消息的发送就有消息的接受，在消息接收时候，需要将 string 转化为 MyPacket，因此这里需要用到函数 to_MyPacket，函数先进性消息的检验，再进行字符串的解析，最后返回一个 MyPacket 类型。

```
//将收到的 string 转换为 MyPacket
std::optional<MyPacket> to_MyPacket(std::string recv_mes){
  MyPacket res;
  std::string flag = PACKETFLAG;
  std::string message;

  //首先进行消息的检验
  if(recv_mes.length() <= 9 || recv_mes.substr(0, 7) != PACKETFLAG)
    return std::nullopt;

  //然后进行消息的解析
  try{
    message = recv_mes.substr(flag.length() + 2);
  }catch(std::exception& e){
    std::cerr << "Caught exception of type: " << typeid(e).name() << std::endl;
    std::cout << "Standard exception: " << e.what() << std::endl;
    exit(1);
  }
  res.init_packet(recv_mes[flag.length()], message, recv_mes[flag.length() + 1]);
```

```
  return res;
}
```

如上即为数据包的相关描述，下面是请求数据包的相关格式与描述，看表格即可，不再赘述

请求类型	数据包格式	描述
t	't'	获取时间
n	'n'	获取服务器名字
l	'l'	获取客户端列表
s	's'+'(target_id)'+"(message)"	发送消息给指定客户端
d	'd'	断开连接

- 描述响应数据包的格式（画图说明），响应类型的定义

响应数据包的格式与请求数据包的格式相同，只是类型和相应的内容不同，因此仅展示数据包定义表：

响应类型	数据包格式	描述
d	'd'+"disconnect"	取消连接响应
e	'e'+"error request type"	未定义请求类型
t	't'+"(time)"	发送本地时间
n	'n'+"(name)"	发送服务器名字
l	'l'+"(\nid1:[IP地址,端口]\nid2:[IP地址,端口]...)"	发送客户端列表
a	'a'+"(acknoledge message)"	发送成功或者失败的消息

- 描述指示数据包的格式（画图说明），指示类型的定义

指示数据包的格式与请求数据包的格式相同，只是类型和相应的内容不同，因此仅展示数据包定义表，成功连接指示即在连接成功时发送给客户端，发送消息给指定客户端即为‘s’类型的请求下对于目标id 客户端的消息发送：

指示类型	数据包格式	描述
h	'h'+"hello! IP:xxx Port:xxx"	成功连接指示
s	's'+"(message)"	发送消息给指定客户端

- 客户端初始运行后显示的菜单选项

运行客户端程序，首先进行连接

```
❯ ./client
[System] 欢迎！
************功能菜单************
* 1. 连接                       *
* 2. 退出                       *
********************************
[System] 请输入序号选择功能：
[Client]
```

输入相关信息后，对服务端进行连接，连接成功会提示连接成功，并打印功能菜单，之后输入功能对应的序号即可，如下以[server]开头的消息是接受到的消息，如下连接成功后，收到 server 发来的成功连接相应：

```
[System] 请输入序号选择功能：
[Client] 1
[System] 请输入服务器的IP地址：(-1强制退出)
127.0.0.1
[System] 请输入服务器的端口：(-1强制退出)
13283
[System] 连接成功！
************功能菜单************
* 1. 获取时间                   *
* 2. 获取名字                   *
* 3. 获取客户端列表             *
* 4. 发送消息                   *
* 5. 断开连接                   *
* 6. 退出                       *
********************************
[System] 请输入序号选择功能：
[Client] [Server] hello! IP:127.0.0.1 Port:58266
```

- 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){
    …
    /* 打印菜单 */
    int op = print_menu_list();

    /* 退出程序 */
    if(op == 2 && !is_connected){
        break;
    }

    /* 连接服务端 */
```

```
        if(op == 1 && !is_connected){
            tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
            auto dst = input_ip_and_port();
            if(!dst.has_value()) break;

            struct sockaddr_in serv_addr;
            memset(&serv_addr, 0, sizeof(serv_addr));
            serv_addr.sin_family = AF_INET;
            serv_addr.sin_port = htons(dst.value().second); //端口号需要转为网络
序
            serv_addr.sin_addr.s_addr = inet_addr(dst.value().first.c_str());

            if(connect(tcp_socket, (struct sockaddr*)&serv_addr,
sizeof(serv_addr)) != -1){
                std::cout << "\033[32m[System]\033[0m 连接成功!" << std::endl;
                is_connected = true;
                std::thread t(recv_msg_thread);
                t.detach();
            }
            else{
                std::cout << "\033[31m[System]\033[0m 连接失败!" << std::endl;
                is_connected = false;
            }
            continue;
        }

        /* 向服务端发送服务请求 */
        /* 分别对六种情况的服务进行处理，即将请求封装成为格式要求的数据包，然后发送至
服务器 */
        if(is_connected){
            MyPacket msg_s;
            std::optional<std::pair<char, std::string>> dst;
            switch(op){
                …
            }
            …
        `
```

- 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 recv_msg_thread(){
    std::string buf_str;
    while(is_connected){
    /* 建立缓冲区并清空 */
        char buf[MAXSIZE];
        memset(buf, 0, sizeof(buf));
    /* 接收数据 */
        recv(tcp_socket, buf, sizeof(buf) - 1, 0);
    /* 解包得到信息部分 */
        auto p = to_MyPacket(buf);
        memset(buf, 0, sizeof(buf));
        MyPacket recv_packet;
    /* 检查数据合法性 */
        if(!p.has_value()) continue;
        else recv_packet = p.value();
    /* 输出服务器端响应 */
        std::cout << "\033[35m[Server]\033[0m " << recv_packet.message <<
std::endl;
    }
}
```

- 服务器初始运行后显示的界面

如下图所示，服务器将显示初始化信息与状态，然后持续监听用户请求

```
❯ ./server
Bind PORT13283 succeed!
Server successfully initialized!
Server starts working!
Waiting for client to connect...
```

若有客户端请求连接，则会进行相应，如下，send message 之后的内容将发送给客户端

```
❯ ./server
Bind PORT13283 succeed!
Server successfully initialized!
Server starts working!
Waiting for client to connect...

A client connected!
   Client IP: 127.0.0.1
   Client port: 58266
   send messsage: hello! IP:127.0.0.1 Port:58266
```

- 服务器的主线程循环关键代码截图

如下所示，整个主线程在一个无限的循环中，通过调用 accept 函数持续等待来自客户进程的实际连接，若收到连接，则输出客户端相关的信息，将客户端的信息加入到主线程维护的一个客户端的信息表中，然后通过 pthread_create 函数来进行子线程的创建。

```
void MyServer::on(){
  cout << "Server starts working!" << endl;
  cout << "Waiting for client to connect..." << endl;
  while(true){
    //接受客户端连接
    sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int client_sockfd = accept(server_sockfd, (sockaddr*)&client_addr,
&client_addr_len);
    if(client_sockfd == -1){
      cout << "Accept error!" << endl;
      continue;
    }

    //输出相关消息
    cout << endl << "A client connected!" << endl;
    cout << "   Client IP: " << inet_ntoa(client_addr.sin_addr) << endl;
    cout << "   Client port: " << ntohs(client_addr.sin_port) << endl;

    //将客户端信息加入客户端列表
    int list_num = get_list_num();
    client_list[list_num].client_sockfd = client_sockfd;
    client_list[list_num].client_addr = client_addr;
    set_client_list(list_num);

    //创建线程处理客户端请求
    pthread_t thread;
    struct thread_info info = {list_num, client_sockfd, client_addr, this};
    pthread_create(&thread, NULL, handle_client, (void *)&info);
  }
}
```

如果所有已连接的客户端退出，需要服务端也结束服务，这一逻辑在子线程当中调用 exit(0)来实现整体的退出，因此放在下一点进行解释。

- 服务器的客户端处理子线程循环关键代码截图

如下代码所示，此函数即为面向每个客户端的单独的子线程，开启后，会立即先调用 type_h 函数，也就是发送类型为'h'的成功连接指示消息来告诉客户端成功连接。之后进入一个循环来进行消息处理，调用 recv 来接受消息，然后通过 to_MyPacket 函数将接收到的字符流转换为 MyPacket 类型，然后调用 handle_request 函数来处理单条请求，若为断开连接的请求，则返回 1，那么退出循环，退出循环后，需

要将信息从服务列表中删去，如果服务列表空了，那么就调用 exit(0)退出，也就是整个服务端就结束运行。

```
void *handle_client(void* thread_info){
  //接受参数
  struct thread_info info = *((struct thread_info*)thread_info);
  int list_num = info.list_num;
  int client_sockfd = info.client_sockfd;
  sockaddr_in client_addr = info.client_addr;
  MyServer server = *(info.server);

  //发送成功连接响应
  type_h(client_sockfd, client_addr);

  char buffer[MAX_BUFFER];
  //循环进行消息处理
  while(true){
    //调用 recv 函数接收数据
    ssize_t res = recv(client_sockfd, buffer, MAX_BUFFER, 0);
    //若 recv 返回 0，则连接已经关闭，直接 break
    if(res == 0) break;

    //将收到的数组转换为数据包
    string recv_str(buffer);
    MyPacket recv_packet;
    auto recv = to_MyPacket(recv_str);
    if(!recv.has_value()) continue;
    else recv_packet = recv.value();

    //处理请求
    cout << endl << "[" << list_num + 1 << "]handle request..." << endl;
    res = handle_request(recv_packet, info, list_num);

    //如果收到退出请求，则返回 0
    if(res == 1){
      primary_server.set_is_end_true();
      break;
    }
  }

  close(client_sockfd); //关闭客户端套接字
  std::lock_guard<std::mutex> lock(mutex);
  primary_server.rst_client_list(list_num); //释放
```

```
  //如果所有客户端都已经退出，则服务器退出
  if(primary_server.is_end() && !primary_server.check_client_list()){
    cout << "All the client disconnected! Server timinate!" << endl;
    close(primary_server.get_server_sockfd());
    exit(0);
  }

  return 0;
}
```

- 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：

```
❯ ./client
[System] 欢迎！
************功能菜单************
* 1. 连接                       *
* 2. 退出                       *
********************************
[System] 请输入序号选择功能：
[Client] 1
[System] 请输入服务器的IP地址：(-1强制退出)
127.0.0.1
[System] 请输入服务器的端口：(-1强制退出)
13283
[System] 连接成功！
************功能菜单************
* 1. 获取时间                   *
* 2. 获取名字                   *
* 3. 获取客户端列表             *
* 4. 发送消息                   *
* 5. 断开连接                   *
* 6. 退出                       *
********************************
[System] 请输入序号选择功能：
[Client] [Server] hello! IP:127.0.0.1 Port:43972
```

服务端：

```
❯ ./server
Bind PORT13283 succeed!
Server successfully initialized!
Server starts working!
Waiting for client to connect...

A client connected!
   Client IP: 127.0.0.1
   Client port: 43972
   send messsage: hello! IP:127.0.0.1 Port:43972
```

Wireshark 抓取的数据包截图：

```
▸ Frame 4: 105 bytes on wire (840 bits), 105 bytes captured (840 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 13283, Dst Port: 46582, Seq: 1, Ack: 1, Len: 39
▾ Data (39 bytes)
    Data: 7a687a266e7a68683168656c6c6f212049503a3132372e302e302e3120506f72743a3436…
    [Length: 39]
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 5b 33 0f 40 00 40 06  09 8c 7f 00 00 01 7f 00   ·[3·@·@· ........
0020  00 01 33 e3 b5 f6 01 df  cb 12 52 c0 fe ae 80 18   ··3····· ··R·····
0030  02 00 fe 4f 00 00 01 01  08 0a a4 d1 67 1a a4 d1   ···O···· ····g···
0040  67 19 7a 68 7a 26 6e 7a  68 68 31 68 65 6c 6c 6f   g·zhz&nz hh1hello
0050  21 20 49 50 3a 31 32 37  2e 30 2e 30 2e 31 20 50   ! IP:127 .0.0.1 P
0060  6f 72 74 3a 34 36 35 38  32                       ort:4658 2
```

- 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：

```
************功能菜单************
* 1. 获取时间                    *
* 2. 获取名字                    *
* 3. 获取客户端列表              *
* 4. 发送消息                    *
* 5. 断开连接                    *
* 6. 退出                        *
*********************************
[System] 请输入序号选择功能：
[Client] 1
[System] 请求成功发送，请稍等...
[Server] 2023年12月16日 23:47:15
```

服务端：

```
[1]handle request...
   send messsage: 2023年12月16日 23:47:15
```

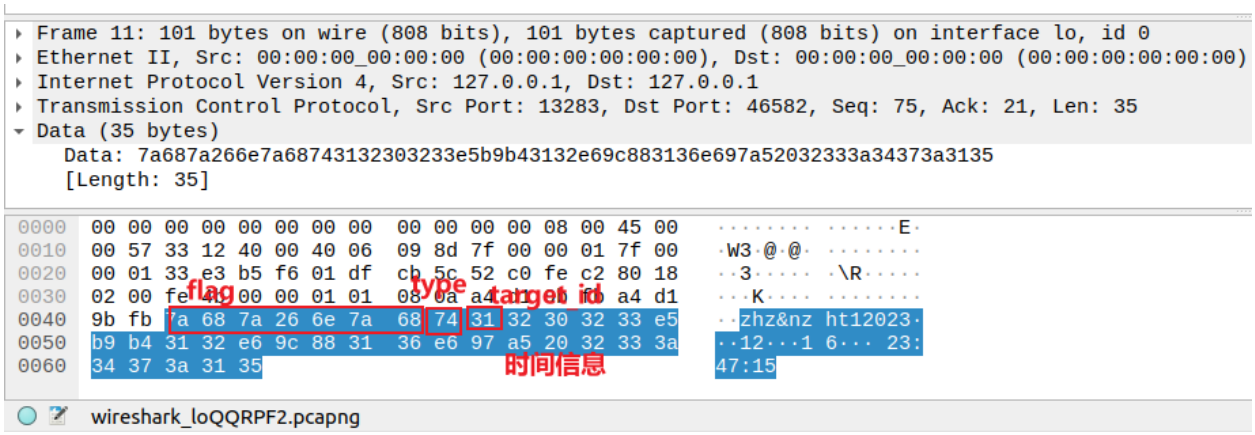
Wireshark 抓取的数据包截图：

请求：

```
▸ Frame 10: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 46582, Dst Port: 13283, Seq: 11, Ack: 75, Len: 10
▾ Data (10 bytes)
    Data: 7a687a266e7a68743131
    [Length: 10]
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 3e 93 cd 40 00 40 06  a8 ea 7f 00 00 01 7f 00   ·>··@·@· ........
0020  00 01 b5 f6 33 e3 52 c0  fe b8 01 df cb 5c 80 18   ····3·R· ·····\··
0030  02 00 fe 32 00 00 01 01  08 0a a4 d1 9b fb a4 d1   ···2···· ........
0040  89 ff 7a 68 7a 26 6e 7a  68 74 31 31               ··zhz&nz ht11
```

响应：



- 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：

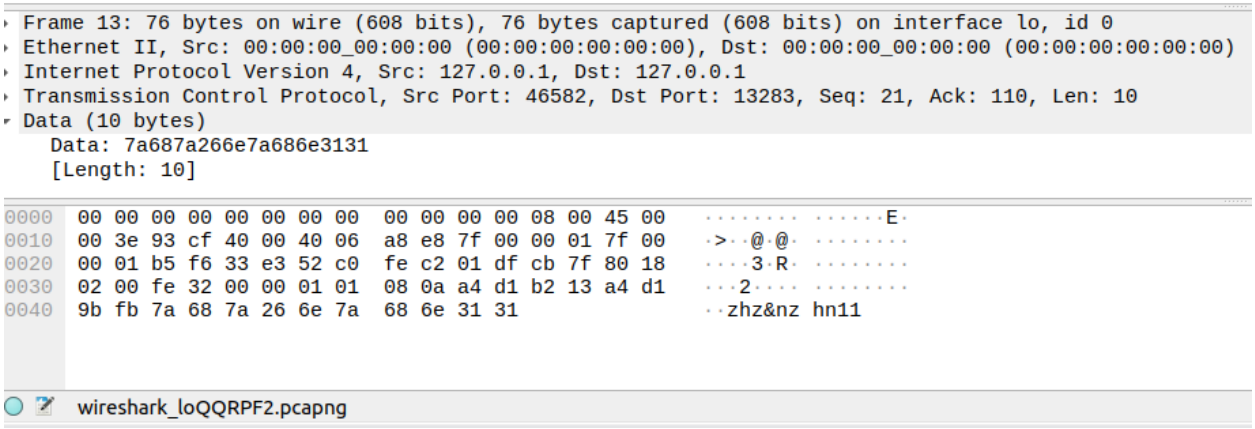
```
[Server] 2023年12月10日 23:47:15
************功能菜单************
* 1. 获取时间                    *
* 2. 获取名字                    *
* 3. 获取客户端列表              *
* 4. 发送消息                    *
* 5. 断开连接                    *
* 6. 退出                        *
**********************************
[System] 请输入序号选择功能：
[Client] 2
[System] 请求成功发送，请稍等...
[Server] ubuntu
************功能菜单************
```

服务端：

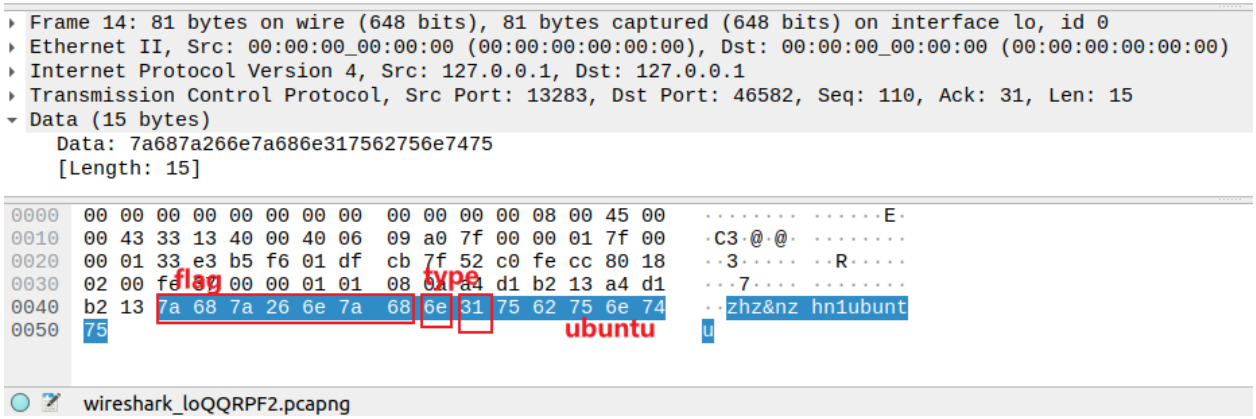
```
[1]handle request...
   send messsage: ubuntu
```

Wireshark 抓取的数据包截图：

请求：



响应：



相关的服务器的处理代码片段：

```
//处理单个请求
int handle_request(MyPacket request, struct thread_info info, int request_id){
  //拆开数据包
  char type = request.get_type();
  char targetid = request.get_target();
  string message = request.get_message();

  switch (type)
  {
    case 't': type_t(info.client_sockfd); break;
    case 'n': type_n(info.client_sockfd); break;
    case 'd': type_d(info.client_sockfd); return 1;
    case 'l': type_l(info.client_sockfd, *(info.server)); break;
    case 's': type_s(info.client_sockfd, request_id, targetid, message,
*(info.server)); break;
    default: type_e(info.client_sockfd); break;
  }

  return 0;
}
```

```
//名字请求
void type_n(int client_sockfd){
  //构造消息
  char name[256];
  gethostname(name, sizeof(name));
  string hostname(name);
```

```
  //发送消息
  MyPacket send_packet('n', hostname);
  cout << "   send messsage: " << hostname << endl;
  send(client_sockfd, send_packet.to_string().c_str(),
send_packet.to_string().size(), 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：

```
************功能菜单************
* 1. 获取时间                   *
* 2. 获取名字                   *
* 3. 获取客户端列表             *
* 4. 发送消息                   *
* 5. 断开连接                   *
* 6. 退出                       *
********************************
[System] 请输入序号选择功能：
[Client] 3
[System] 请求成功发送，请稍等...
[Server]
id1: [127.0.0.1 37626]
id2: [127.0.0.1 55182]
```

服务端：

```
[1]handle request...
   send messsage:
id1: [127.0.0.1 37626]
id2: [127.0.0.1 55182]
```

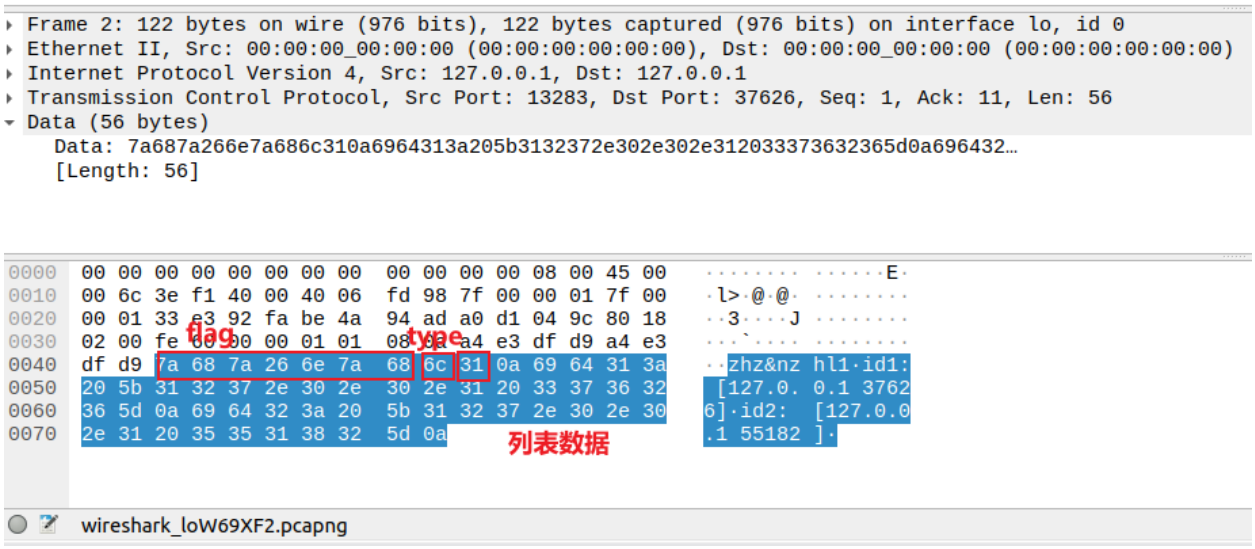
Wireshark 抓取的数据包截图：

请求：

```
▸ Frame 1: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 37626, Dst Port: 13283, Seq: 1, Ack: 1, Len: 10
▾ Data (10 bytes)
    Data: 7a687a266e7a686c3131
    [Length: 10]
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 3e ac d2 40 00 40 06  8f e5 7f 00 00 01 7f 00   ·>··@·@· ........
0020  00 01 92 fa 33 e3 a0 d1  04 92 be 4a 94 ad 80 18   ····3··· ···J····
0030  02 00 fe 32 00 00 01 01  08 0a a4 e3 df d9 a4 e2   ···2···· ........
0040  f6 3d 7a 68 7a 26 6e 7a  68 6c 31 31               ·=zhz&nz hl11
```

响应：



相关的服务器的处理代码片段：

```
//列表读取请求
void type_l(int client_sockfd, MyServer server){
  //使用 get_list 读取并发送消息
  MyPacket send_packet('l', server.get_list());
  cout << "   send messsage: " << server.get_list();
  send(client_sockfd, send_packet.to_string().c_str(),
send_packet.to_string().size(), 0);
}
```

```
string MyServer::get_list(){
  stringstream ss;
  ss << endl;
  map<int, struct client_info>::iterator it;
  for (it = client_list.begin(); it != client_list.end(); it++) {
    ss << "id" << it->first + 1 << ": [";
    ss << inet_ntoa((it->second).client_addr.sin_addr) << " ";
    ss << ntohs((it->second).client_addr.sin_port);
    ss << "]" << endl;
  }
  return ss.str();
}
```

- 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：

```
************功能菜单************
* 1. 获取时间                    *
* 2. 获取名字                    *
* 3. 获取客户端列表              *
* 4. 发送消息                    *
* 5. 断开连接                    *
* 6. 退出                        *
*********************************
[System] 请输入序号选择功能：
[Client] 4
[System] 请输入发送客户端的列表编号:(-1强制退出)
2
[System] 请输入发送内容:(-1强制退出)
hello
[System] 请求成功发送，请稍等...
[Server] already send the message!
```

服务器：

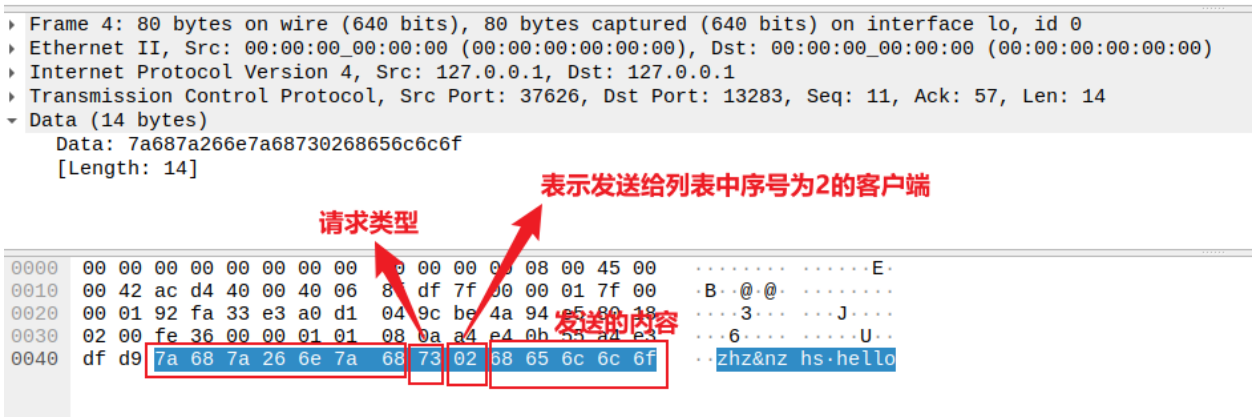
```
[1]handle request...
   send messsage to [2]: From [1]: hello
   send messsage: already send the message!
```

接收消息的客户端：

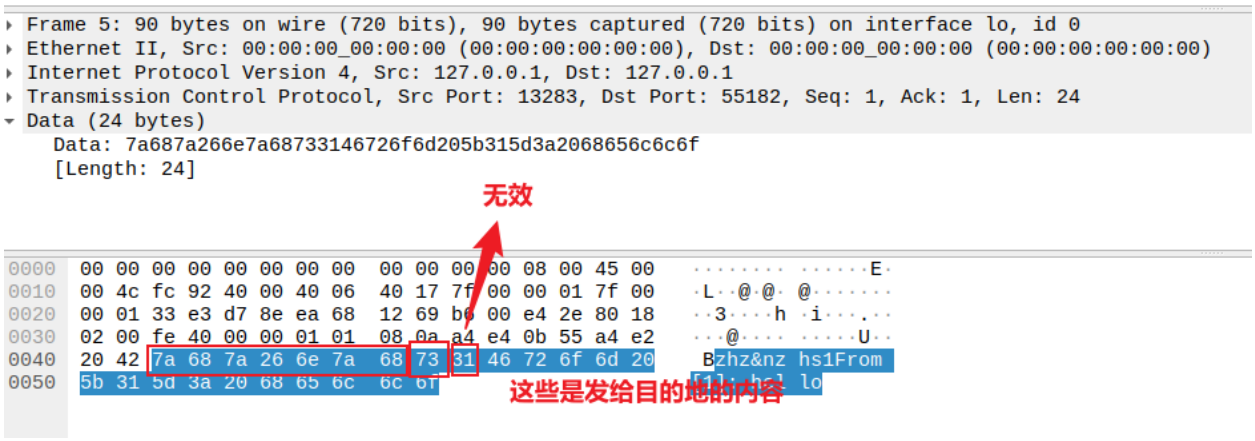
```
[Server] From [1]: hello
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

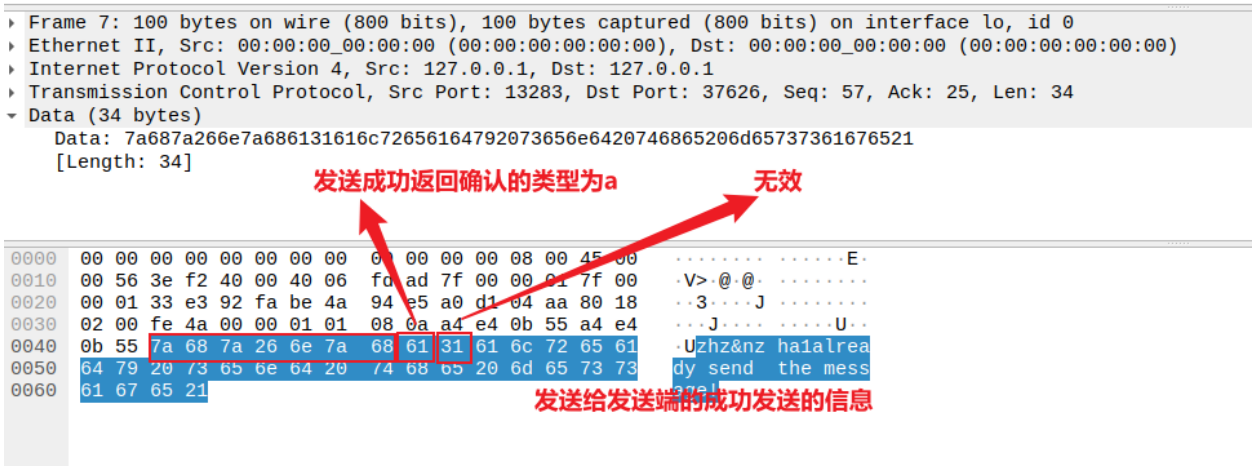
发送请求：



服务端发送：



服务端响应成功通知：



相关的服务器的处理代码片段：

```
//转发请求
void type_s(int client_sockfd, int client_id, char targetid, string message,
MyServer server){
  //首先检查目标 id 是否存在
  int target_sockfd = server.find_in_list(targetid - 1);
  if(target_sockfd == -1){
    type_a(client_sockfd, "target id not exist!");
    return;
  }

  //构造消息
  string mes = "From [";
  mes.push_back(client_id + 1 + '0');
  mes.append("]: ");
  mes.append(message);

  //发送消息
```

```
  MyPacket send_packet('s', mes);
  cout << "   send messsage to ["<< (int)targetid  << "]: " << mes << endl;
  send(target_sockfd, send_packet.to_string().c_str(),
send_packet.to_string().size(), 0);

  //发送确认回复
  type_a(client_sockfd, "already send the message!");
}
```

```
//转发确认回复
void type_a(int client_sockfd, string message){
  //发送传入的 messaeg
  MyPacket send_packet('a', message);
  cout << "   send messsage: " << message << endl;
  send(client_sockfd, send_packet.to_string().c_str(),
send_packet.to_string().size(), 0);
}
```

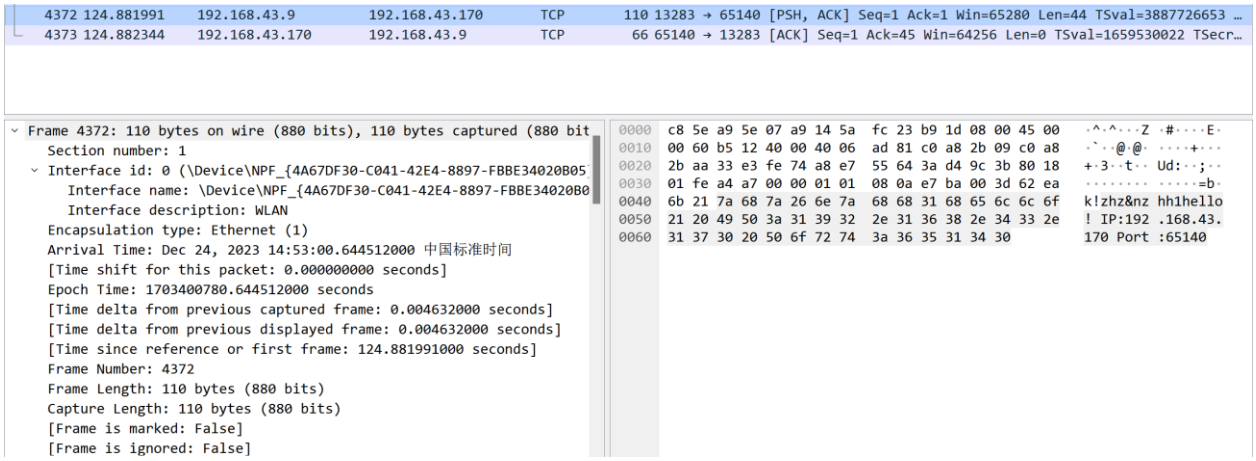
相关的客户端（发送和接收消息）处理代码片段：

```
/* 向服务端发送服务请求 */
if(is_connected){
    MyPacket msg_s;
    std::optional<std::pair<char, std::string>> dst;
    switch(op){
        case 1: /* 处理查询日期请求 */
            msg_s.init_packet('t');
            break;

        case 2: /* 处理查询名字请求 */
            msg_s.init_packet('n');
            break;

        case 3: /* 处理查询客户端列表请求 */
            msg_s.init_packet('l');
            break;

        case 4: /* 处理发送信息请求 */
            dst = input_id_and_msg();
            if(dst.has_value())
                msg_s.init_packet('s', dst.value().second, dst.value().first);
            else
```

```
                throw("无效id, 请查询正确的客户端id后再发送!");
            break;

        case 5: /* 处理断开连接请求 */
            msg_s.init_packet('d');
            is_connected = false;
            break;

        case 6: /* 处理退出请求 */
            msg_s.init_packet('d');
            is_connected = false;
            break;

        default:
            break;

    }
    try{
        std::string str = msg_s.to_string();
        const char* msg = str.c_str();
        if(send(tcp_socket, msg, str.size(), 0) == -1)
            throw("请求发送失败!");
        else{
            std::cout << "\033[32m[System]\033[0m 请求成功发送, 请稍等..." <<
std::endl;
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
        if(!is_connected)
            close(tcp_socket);
    } catch (const std::exception& e) {
        std::cout << "\033[31m[System]\033[0m 发生异常: " << e.what() <<
std::endl;
    }
```

- 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。

拔掉客户端网线，退出客户端程序，观察客户端 TCP 连接状态仍为 ESTABLISHED。如下图所示，wireshark 捕获的最后两条消息是 server 发送给 client 的 hello 和 client 给 server 的确认应答，并没有发送 TCP 连接释放的消息。



等待 15 分钟之后，通过 netstat 指令查看，发现该客户端的 TCP 连接仍然是 ESTABLISHED，并没有发生变化。

```
A client connected!
   Client IP: 192.168.43.87
   Client port: 56099
   send messsage: hello! IP:192.168.43.87 Port:56099
```

```
❯ netstat -an | head -20
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:3283            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:33060         0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 192.168.43.9:3283       192.168.43.87:56099     ESTABLISHED
tcp6       0      0 ::1:631                 :::*                    LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
tcp6       0      0 240e:473:8a0:4a9::36718 2620:2d:4000:1::23:80   TIME_WAIT
udp        0      0 127.0.0.53:53           0.0.0.0:*
udp        0      0 192.168.43.9:68         192.168.43.1:67         ESTABLISHED
udp        0      0 0.0.0.0:631             0.0.0.0:*
```

- 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？

首先客户端发送了列表请求，发现上一次因为断网而没有断开的 tcp 连接仍然在 server 列表中。

```
> ./client
[System] 欢迎!
************功能菜单************
* 1. 连接                        *
* 2. 退出                        *
********************************
[System] 请输入序号选择功能:
[Client] 1
[System] 请输入服务器的IP地址:(-1强制退出)
192.168.43.9
[System] 请输入服务器的端口:(-1强制退出)
3283
[System] 连接成功!
************功能菜单************
* 1. 获取时间                    *
* 2. 获取名字                    *
* 3. 获取客户端列表              *
* 4. 发送消息                    *
* 5. 断开连接                    *
* 6. 退出                        *
********************************
[System] 请输入序号选择功能:
[Client] [Server] hello! IP:192.168.43.87 Port:56234
3
[Server]
id1: [192.168.43.87 56099]
id2: [192.168.43.87 56234]
```

然后使用发送功能，返回的消息是发送失败

```
* 6. 退出                        *
********************************
[System] 请输入序号选择功能:
[Client] 4
[System] 请输入发送客户端的列表编号:(-1强制退出)
1
[System] 请输入发送内容:(-1强制退出)
hello
[Server] Send failed!
```

然后这之后再发送一次，返回的是那个 id 已经没有了，也就是说此时 server 经过上一次的发送失败知道了 id 为 1 的客户端已经断开，所以列表已经更新，因此这一次发送就直接返回 id 错误了。

```
************功能菜单************
* 1. 获取时间                    *
* 2. 获取名字                    *
* 3. 获取客户端列表              *
* 4. 发送消息                    *
* 5. 断开连接                    *
* 6. 退出                        *
********************************
[System] 请输入序号选择功能:
[Client] 4
[System] 请输入发送客户端的列表编号:(-1强制退出)
1
[System] 请输入发送内容:(-1强制退出)
hello
[Server] target id not exist!
```

如下是经过操作：连接第一个客户端，然后拔网线退出；连接网线，连接第二个客户端；第二个客户端发送列表功能；第二个客户端发送两次发送消息功能之后的服务端的截图，作为印证。

```
A client connected!
   Client IP: 192.168.43.87
   Client port: 56099
   send messsage: hello! IP:192.168.43.87 Port:56099

A client connected!
   Client IP: 192.168.43.87
   Client port: 56234
   send messsage: hello! IP:192.168.43.87 Port:56234

[2]handle request...
   send messsage:
id1: [192.168.43.87 56099]
id2: [192.168.43.87 56234]

[2]handle request...
   send messsage to [1]: From [2]: hello
   send messsage: Send failed!

[2]handle request...
   send messsage: target id not exist!
```

- 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。

添加如下代码来实现自动发送 100 此请求

```
#if test
    for(int i = 1; i <= 100; i++){
        if(send(tcp_socket, msg, str.size(), 0) == -1)
            throw("请求发送失败!");
        sleep(0);
    }
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "\033[32m[System]\033[0m 请求成功发送，请稍等..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
```

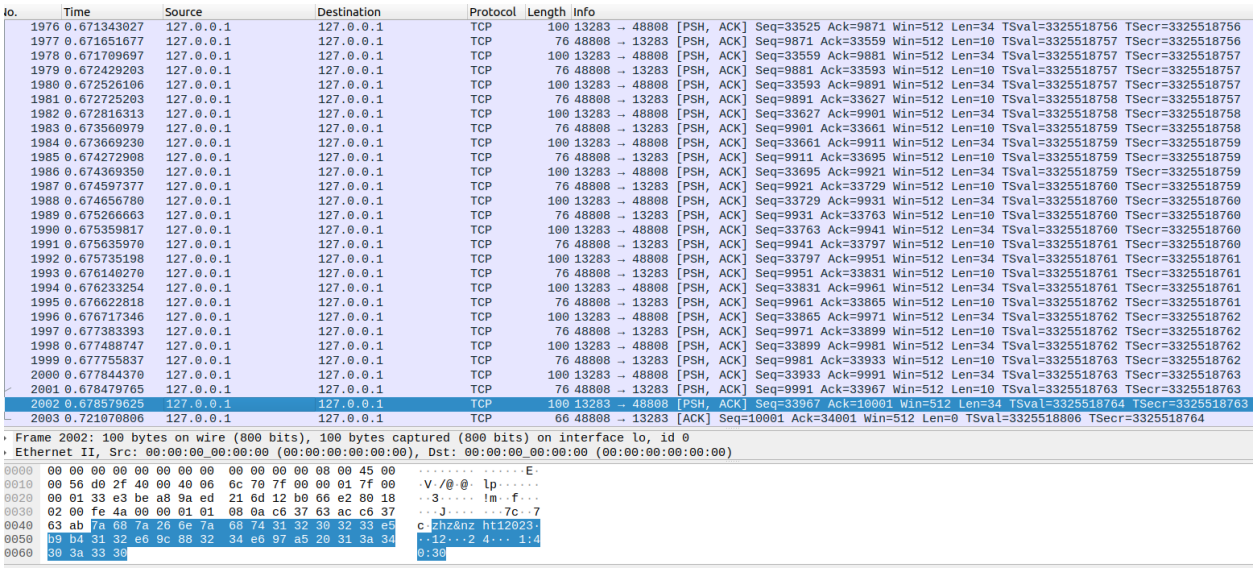
由于一些未知原因，发送一百次消息能够有 100 个响应回来，如下图所示：

```
[Server] 2023年12月24日 1:35:8 75
[Server] 2023年12月24日 1:35:8 76       [1]handle request...
[Server] 2023年12月24日 1:35:8 77          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 78
[Server] 2023年12月24日 1:35:8 79       [1]handle request...
[Server] 2023年12月24日 1:35:8 80          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 81
[Server] 2023年12月24日 1:35:8 82
[Server] 2023年12月24日 1:35:8 83       [1]handle request...
[Server] 2023年12月24日 1:35:8 84          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 85
[Server] 2023年12月24日 1:35:8 86
[Server] 2023年12月24日 1:35:8 87       [1]handle request...
[Server] 2023年12月24日 1:35:8 88          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 89
[Server] 2023年12月24日 1:35:8 90
[Server] 2023年12月24日 1:35:8 91       [1]handle request...
[Server] 2023年12月24日 1:35:8 92          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 93
[Server] 2023年12月24日 1:35:8 94       [1]handle request...
[Server] 2023年12月24日 1:35:8 95          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 96
[Server] 2023年12月24日 1:35:8 97
[Server] 2023年12月24日 1:35:8 98       [1]handle request...
[Server] 2023年12月24日 1:35:8 99          send messsage: 2023年12月24日 1:35:8
[Server] 2023年12月24日 1:35:8 100
```

因此连续发送 1000 次进行测试，发现能够响应 996 条消息

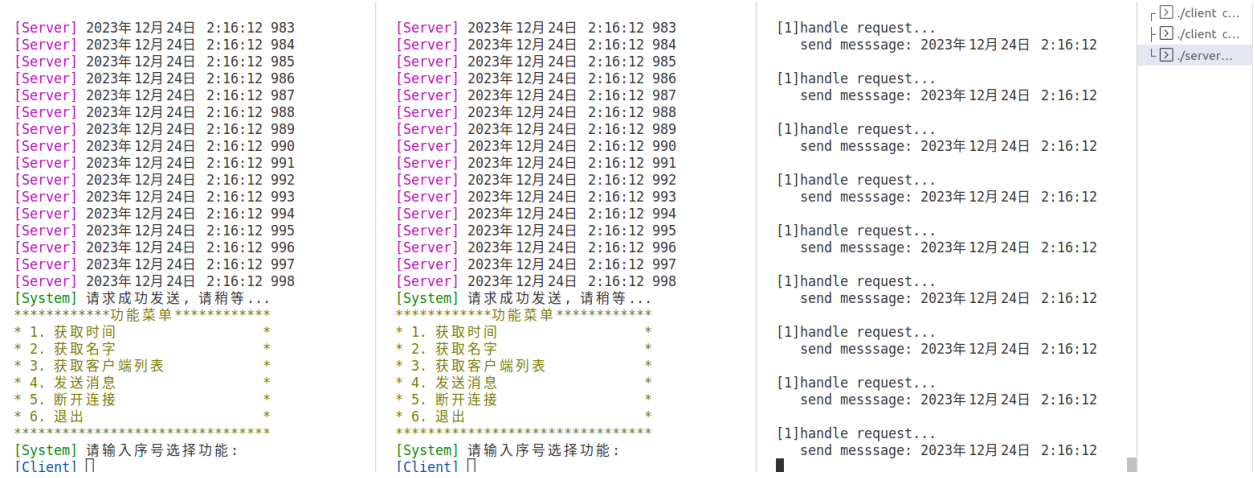
```
[Server] 2023年12月24日 1:40:30 984
[Server] 2023年12月24日 1:40:30 985
[Server] 2023年12月24日 1:40:30 986
[Server] 2023年12月24日 1:40:30 987
[Server] 2023年12月24日 1:40:30 988
[Server] 2023年12月24日 1:40:30 989
[Server] 2023年12月24日 1:40:30 990
[Server] 2023年12月24日 1:40:30 991
[Server] 2023年12月24日 1:40:30 992
[Server] 2023年12月24日 1:40:30 993
[Server] 2023年12月24日 1:40:30 994
[Server] 2023年12月24日 1:40:30 995
[Server] 2023年12月24日 1:40:30 996
```

在此期间，wireshark 一共抓取了 2003 条消息，包含 1000 次请求数据包和 996 次服务端向客户端发送的响应数据包



- 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

如下图所示，多个客户端同时连接服务器并同时发送时间请求，运行截图如下所示，可以看到每个客户端调用了 1000 次 send，最终每个客户端只收到了 998 个响应。



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：客户端不需要调用 bind 操作，因为客户端通过哪个端口与服务器建立连接并不重要，socket API 会为程序自动选择一个未被占用的端口，并通知程序数据什么时候打开端口，因此客户端的源端口是 socket API 自动选择未被占用的端口而产生的，每一次调用 connect 时候端口是会变化的。

- 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：服务端即使不调用 accept，客户端依然可以 connect 成功。当客户端调用 connect 函数时，将引发三次握手过程，服务端收到 ACK 确认报文后，将 SYN 里的连接请求移入 ACCEPT 队列。此时三次握手结束，即 TCP 连接成功建立。然后内核通知用户空间的阻塞的服务进程，服务进程调用 accept 仅仅是从 ACCEPT 队列里取出一个连接而已。也就是说客户端调用 connect 连接服务器，与服务器调用 accept“接受”连接是两个独立的过程。

- 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：两者并不完全一致。连续多次 send，有可能 TCP 为提高传输效率，要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常 TCP 会根据优化算法把这些数据合成一包后一次发送出去。

- 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：若 IP 地址不同，则通过 IP 地址进行区分；在实验中，IP 地址相同，则通过端口号或者通过客户端产生的 socket 描述符来加以区分。

- 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 的连接状态是 TIME_WAIT，该状态持续了一两分钟左右。

- 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：客户端断网异常退出后，服务器的 TCP 连接状态并没有变化。服务器可以每隔一段时间对客户端发送一个数据包进行探测，看客户端是否能在时间范围内进行响应。如果客户端未能在时间范围内进行响应，那么说明客户端可能已经掉线，服务端主动断开连接。

七、 讨论、心得

朱镐哲：这次实验我负责服务端代码的编写，代码的编写其实还是挺快的，首先第一个难点就是服务端代码和客户端代码的同步，由于事先只是口头说了一下数据包的结构，所以刚开始出现了很多的问题，比如因为 send 发送的是一个字符数组，但是 id 在为 0 的时候，直接就截断了，导致后面的消息根本就没有发出去。感觉这次实验挑战更大的是写实验报告，因为那几道断网的题目，开始一直在 127.0.0.1 上面做，没有明白断网的真正意思，后面才发现可以两台电脑来做这个测试，但是遇到非常棘手的问题是完全无法连接。试了一阵子发现连 ping 都 ping 不通，上网查了很多资料，发现因为我们两个都是在 ubuntu 虚拟机中进行的实验，而虚拟机的网络有很多种模式，默认的是 NAT 模式，而这种模式下，虚拟机无法被外网访问，所以需要改成桥接模式，而且桥接模式下还不能是在校园网的情况下，因为校园网需要认证，虚拟机无法连接，最后还是通过手机热点让虚拟机成功在桥接模式下联网成功，成功后还需要运行客户端的电脑也使用我的热点，因为需要在同一个局域网内才行，就这样终于能够连接并完成了断网的测试。抛开这些不谈，这次的实验还是让我更加地熟悉了套接字编程，受益匪浅。

南梓涵：在本次实验中我有许多收获。首先，设计网络应用协议是整个实验的核心。在开始实验之前，我们进行了详细的讨论和规划，明确了通信的目标和需求。我们通过仔细分析通信过程中的数据交换和操作流程，设计了一种简洁高效的自定义数据包格式。这个过程让我深刻体会到协议设计的重要性。一个好的协议可以确保通信的可靠性和高效性，而不合理的协议设计可能导致通信失败或性能下降。因此，在实际设计中，我们需要考虑数据格式、命令交互、错误处理等方面，以确保通信的稳定和可靠。其次，通过实现客户端功能，我深入理解了 Socket 编程接口的使用，掌握了套接字的创建、绑定、连接和通信等基本操作。我学会了如何使用 Socket API 发送和接收数据，以及如何处理异常情况。这让我对网络编程有了更深入的了解，并为今后开发网络应用打下了坚实的基础。此外，在这次实验中我接触到了多进程编程，使用子进程处理服务器端的消息，学习了如何管理和控制子进程，包括进程间的通信、进程的创建和终止等。

在本次实验过程中，挑战和问题主要来自于服务器端和客户端的配合，包括接口方法使用一致以及消息机制的统一，尤其是对消息的格式化处理，花费了较多的时间进行调整；同时，在完成思考题的过程中，我们也遇到了一些设计实现与题目要求不完全相符的问题，做出了进一步的调整。通过 wireshark 抓包分析，在这个过程中进一步加深了对 socket 编程的理解。

这次实验让我深入了解了网络通信的原理和过程以及 socket 编程，培养了我的团队合作能力和问题解决能力，令我收获很多。