

**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

# 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

# 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：
   1. 连接：请求连接到指定地址和端口的服务端
   2. 断开连接：断开与服务端的连接
   3. 获取时间: 请求服务端给出当前时间
   4. 获取名字：请求服务端给出其机器的名称
   5. 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）
   6. 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
   7. 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：
   1. 向客户端传送服务端所在机器的当前时间
   2. 向客户端传送服务端所在机器的名称
   3. 向客户端传送当前连接的所有客户端信息
   4. 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
   5. 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。**

# 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**。

# 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：

1. 定义两个数据包的边界如何识别
2. 定义数据包的请求、指示、响应类型字段
3. 定义数据包的长度字段或者结尾标记
4. 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**）

1. 运行初始化，调用socket()，向操作系统申请socket句柄
2. 编写一个菜单功能，列出7个选项
3. 等待用户选择
4. 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）
5. 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。**
6. 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。
7. 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。
8. 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。
9. 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）。
10. 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）。
11. 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。
12. 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**）

1. 运行初始化，调用socket()，向操作系统申请socket句柄
2. 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
3. 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。
5. 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。

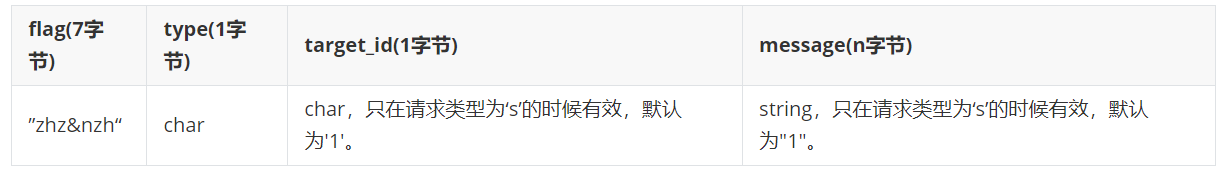
* 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
* 使用多个客户端同时连接服务端，检查并发性
* 使用Wireshark抓取每个功能的交互数据包

# 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）”：

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;

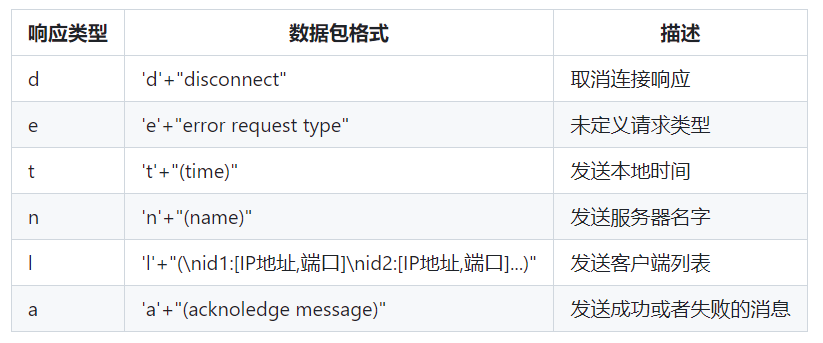
}

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



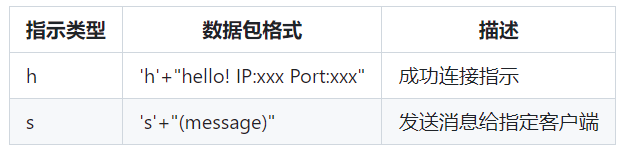
* 描述响应数据包的格式（画图说明），响应类型的定义

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：



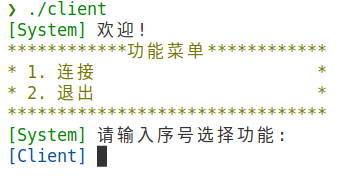
* 描述指示数据包的格式（画图说明），指示类型的定义

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：



* 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



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：



* 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 \*/

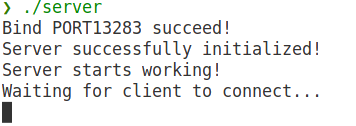
std::cout << "\033[35m[Server]\033[0m " << recv\_packet.message << std::endl;

}

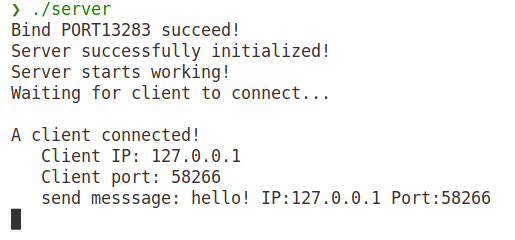
}

* 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



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



* 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;

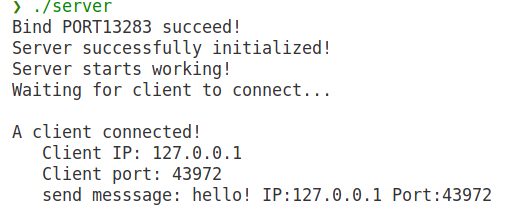
}

* 客户端选择连接功能时，客户端和服务端显示内容截图。

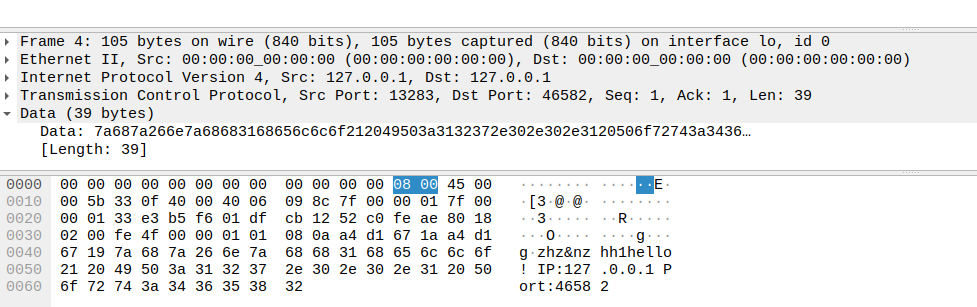
客户端：



服务端：

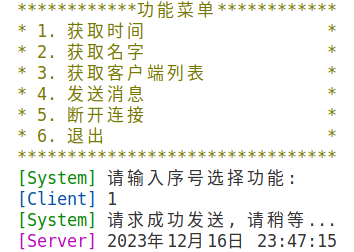


Wireshark抓取的数据包截图：

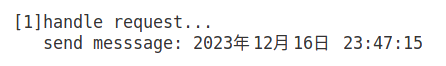


* 客户端选择获取时间功能时，客户端和服务端显示内容截图。

客户端：

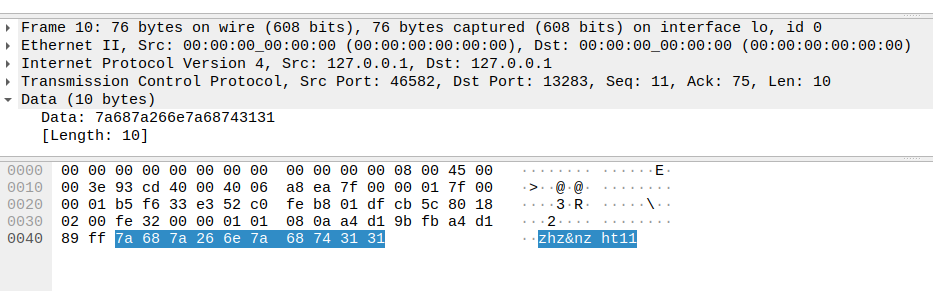


服务端：

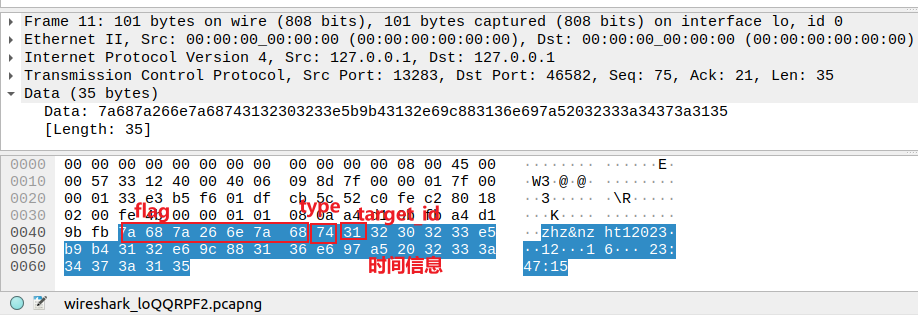


Wireshark抓取的数据包截图：

请求：

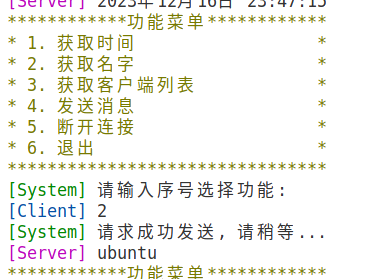


响应：



* 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：

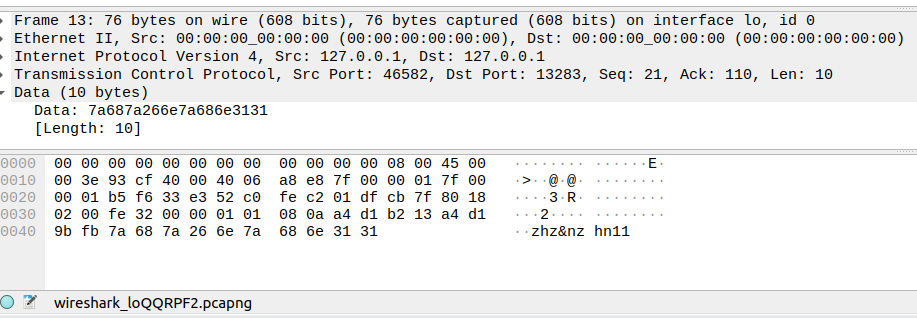


服务端：

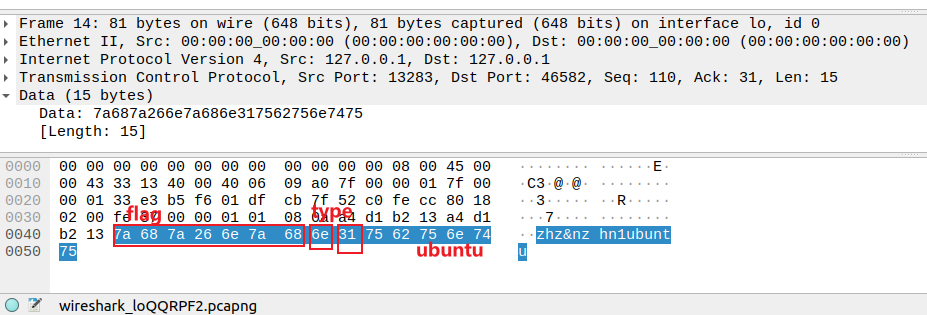


Wireshark抓取的数据包截图：

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);

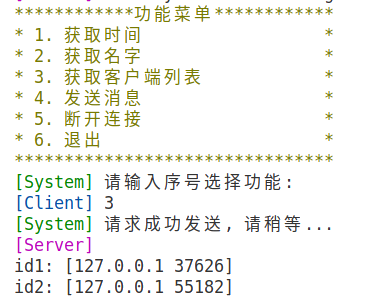
  cout << "   send messsage: " << hostname << endl;

  send(client\_sockfd, send\_packet.to\_string().c\_str(), send\_packet.to\_string().size(), 0);

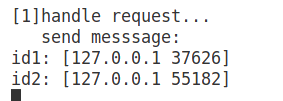
}

* 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：

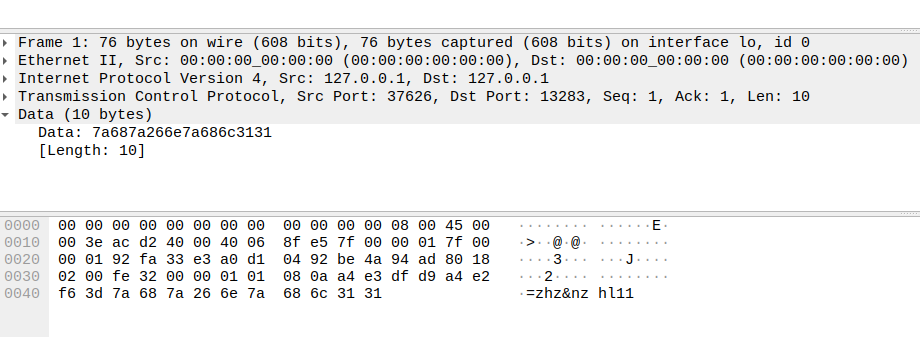


服务端：

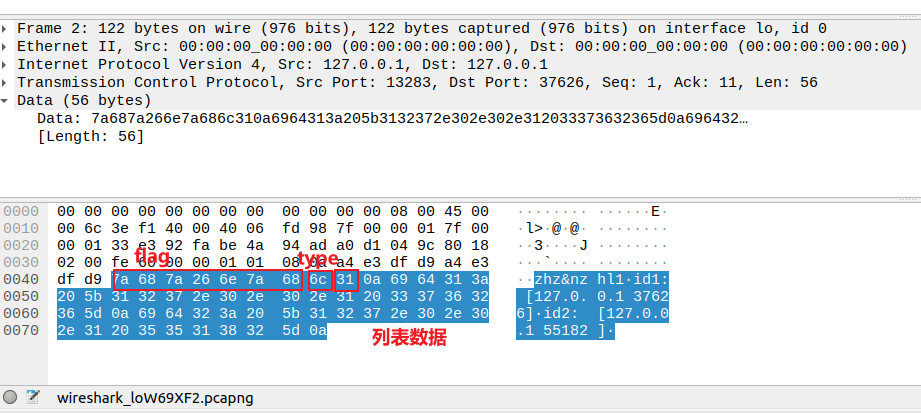


Wireshark抓取的数据包截图：

请求：



响应：



相关的服务器的处理代码片段：

//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;

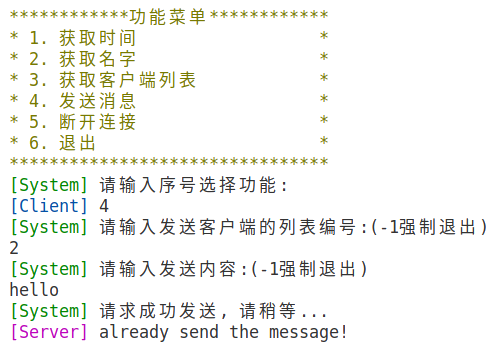
  }

  return ss.str();

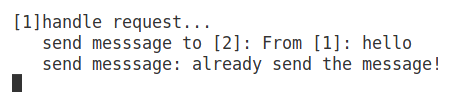
}

* 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：



服务器：

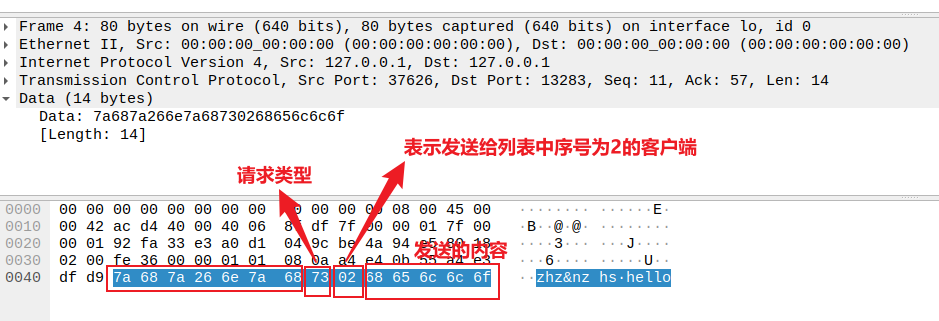


接收消息的客户端：

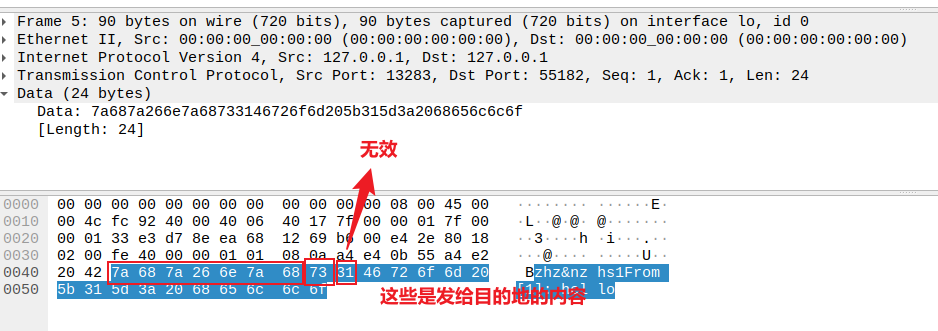


Wireshark抓取的数据包截图（发送和接收分别标记）：

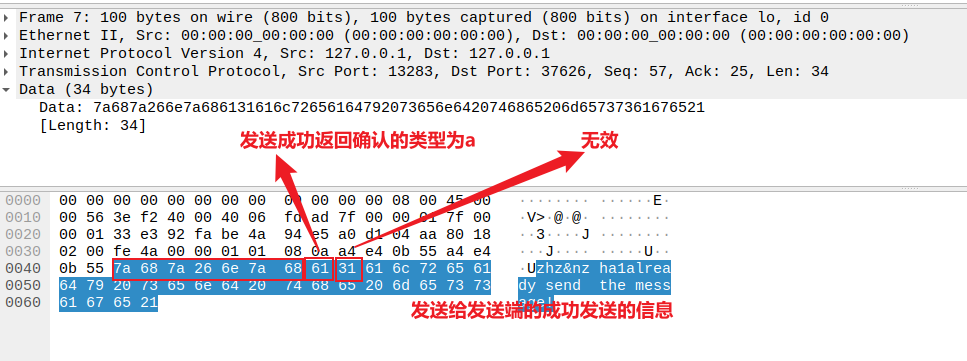
发送请求：



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) {

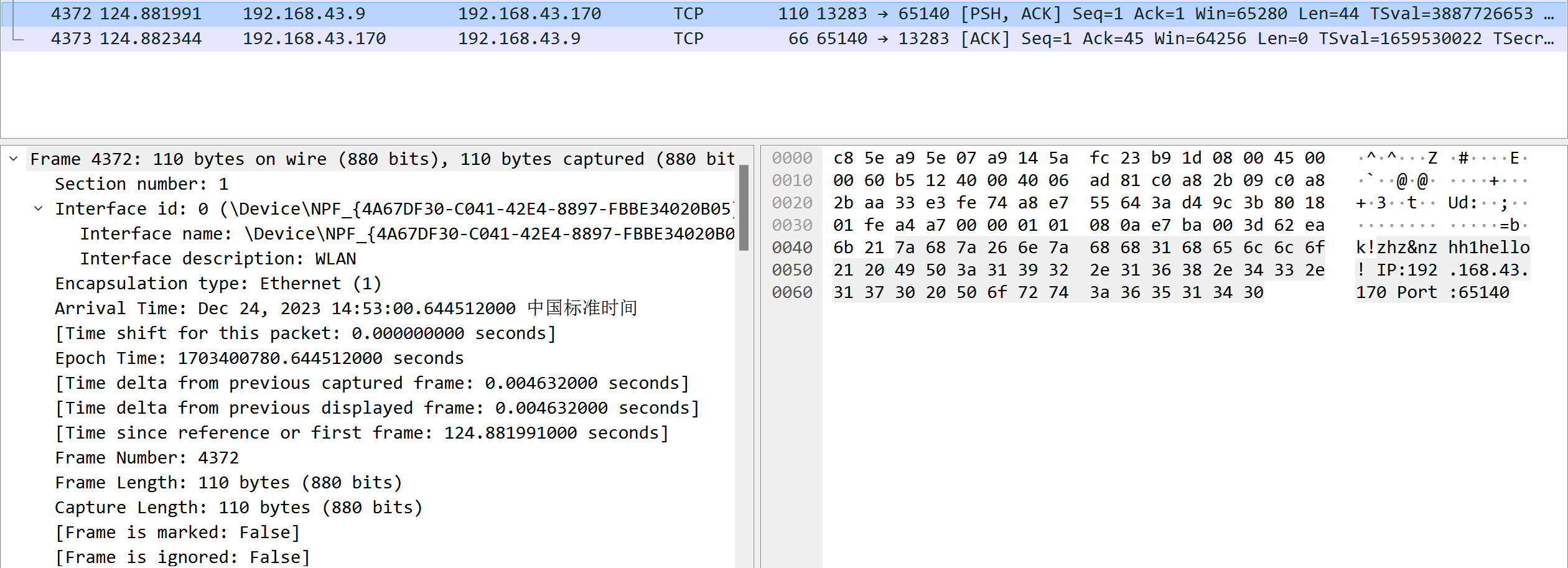
std::cout << "\033[31m[System]\033[0m 发生异常: " << e.what() << std::endl;

}

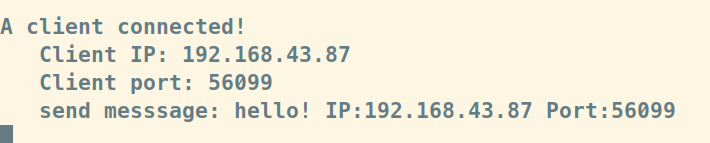
}

* 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。



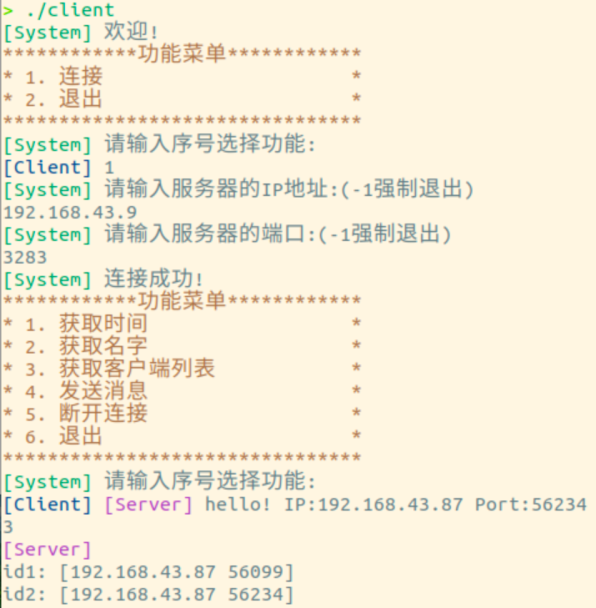
等待15分钟之后，通过netstat指令查看，发现该客户端的TCP连接仍然是ESTABLISHED，并没有发生变化。



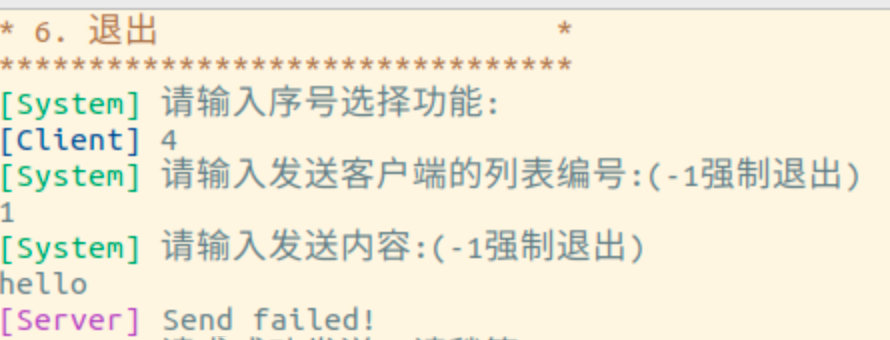


* 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

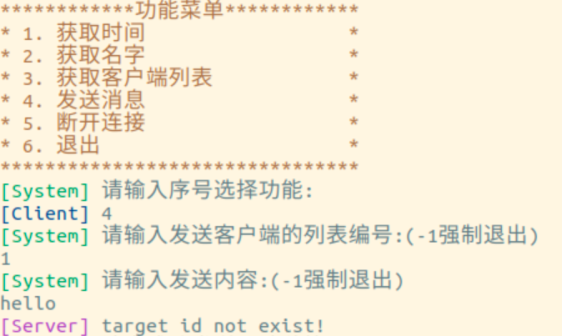
首先客户端发送了列表请求，发现上一次因为断网而没有断开的tcp连接仍然在server列表中。



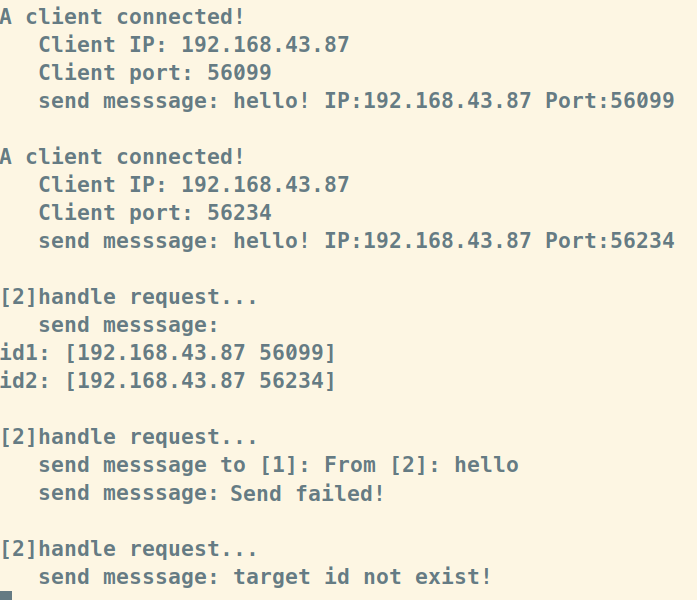
然后使用发送功能，返回的消息是发送失败



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。



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。



* 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);

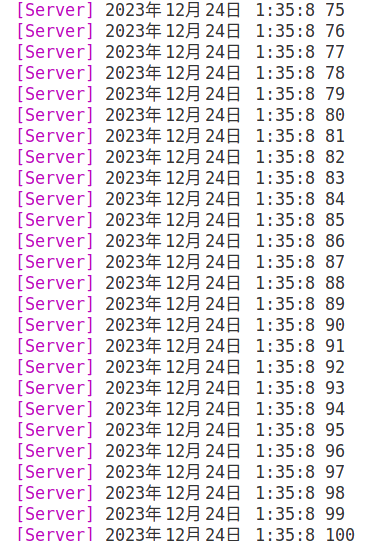
    }

    std::this\_thread::sleep\_for(std::chrono::seconds(1));

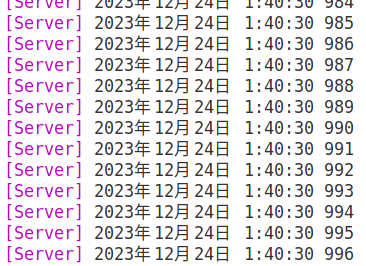
    std::cout << "\033[32m[System]\033[0m 请求成功发送, 请稍等..." << std::endl;

    std::this\_thread::sleep\_for(std::chrono::seconds(1));

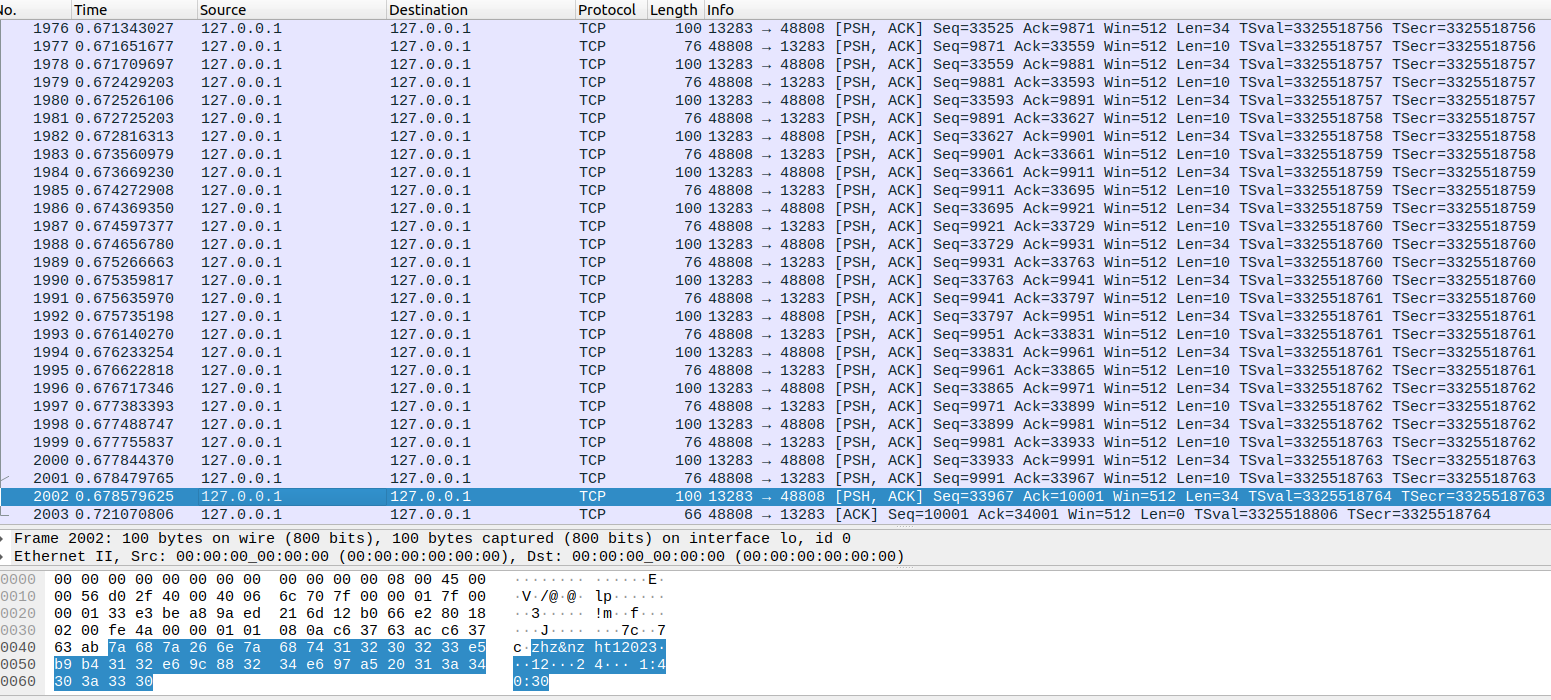
由于一些未知原因，发送一百次消息能够有100个响应回来，如下图所示：

因此连续发送1000次进行测试，发现能够响应996条消息

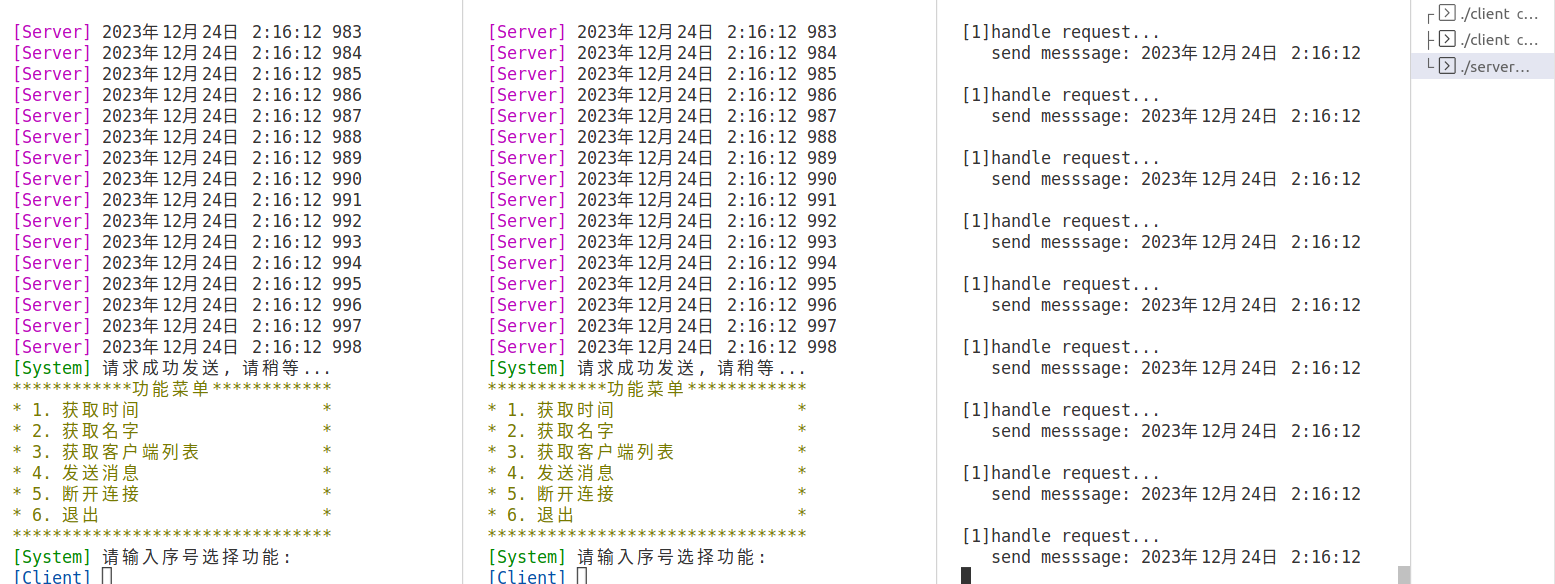


在此期间，wireshark一共抓取了2003条消息，包含1000次请求数据包和996次服务端向客户端发送的响应数据包



* 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。



# 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。

# 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。