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：	贺情怡
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：	3180105438
指导教师：	张泉方

2021 年　　1 月　　13 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：梁晨　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

Int DES （目的地的 socket ID）
REQ_TYPE OPT （请求的类型）
Char[1024] MES （消息内容，固定长度）

```
const int MAXMES = 1024;

enum REQ_TYPE {DISCON, TIME, NAME, LINK, SEND};

string REQ_STR[] = { "DISCON", "TIME", "NAME", "LINK", "SEND" };
```

```
struct MESSAGE {
    int DES;
    REQ_TYPE OPT;
    char MES[MAXMES];
};
```

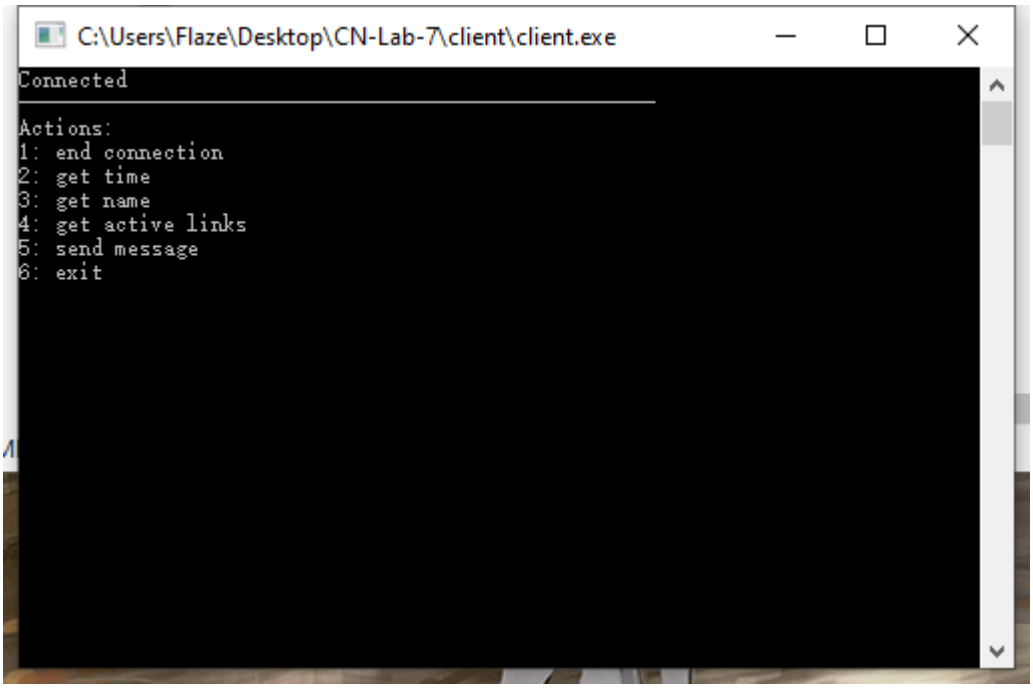
- 描述响应数据包的格式（画图说明），响应类型的定义

Int DES （目的地的 socket ID）
REQ_TYPE OPT （请求的类型）
Char[1024] MES （消息内容，固定长度）

- 描述指示数据包的格式（画图说明），指示类型的定义

Int DES （目的地的 socket ID）
REQ_TYPE OPT （请求的类型）
Char[1024] MES （消息内容，固定长度）

- 客户端初始运行后显示的菜单选项



- 客户端的主线程循环关键代码截图（描述总体，省略细节部分）

  根据输入的指令，选择像服务器发放什么类型的包

```
while(1) {
    // initialize the MESSAGE being sent
    MESSAGE sen ;
    sen = MESSAGE{
        -1, NAME, ""
    } ;

    int opt ;
    string Text ;
    int des_num ;
    cin >> opt ;
    switch (opt) {
        case 1:
            sen.OPT = DISCON ;
            break ;
        case 2:
            sen.OPT = TIME ;
            break ;
        case 3:
            sen.OPT = NAME ;
            break ;
        case 4:
            sen.OPT = LINK ;
            break ;
        case 5:
            // only the type of sending message to others need to be treated differently
            sen.OPT = SEND ;
            // 输入要发送的目的地和信息内容，略
        case 6:
            cout << "Quiting..." << endl ;
```

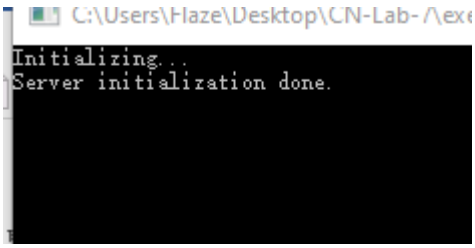
```
                return 0 ;
            default:
                cout << "Wrong option!" << endl ;
                break ;
        }

        if ( opt < 1 || opt > 6 )
            continue ;

        // sending the packet
        char buf[sizeof(MESSAGE)] ;
        memcpy(buf, &sen, sizeof(MESSAGE)) ;
        int seRet = send(client, buf, sizeof(buf), 0);
        if (seRet == -1) {
            cout << "Sending failed:" << errno << endl;
        }
    }
```

- 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 receiveT(SOCKET client) {
    char buf[sizeof(MESSAGE)];
    MESSAGE sen ;

    while ( 1 ) {
        // Message Receiving

    int reRet = recv(client, buf, sizeof(buf), 0);
    if (reRet == -1) {
```

```
            cout << "Receiving failed:"<< errno << endl;
            exit(0);
        }
        memcpy(&sen, buf, sizeof(buf)) ;

        if(sen.OPT == DISCON) {
            cout << "Disconnecting..." << endl ;
            exit(0);
        }
        }
    }
```

- 服务器初始运行后显示的界面



- 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）

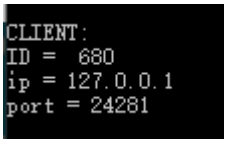
```
while (true) {
    int client = accept(slisten, (sockaddr*)&sin, &addr_size);
    if (client == -1)
            // 省略

    // lambda expression used when using thread
    thread t([&](sockaddr_in addr, int client) {
            // 放在下一题里

    }, sin, client);
    t.detach();
}
```

- 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）

```
char TMP_CLI[255];
// Get ip from hex
inet_ntop(AF_INET, (void*)&sin.sin_addr, TMP_CLI, 16);
Clients[client] = sin;
Clients_info[client].ip = TMP_CLI;
Clients_info[client].port = sin.sin_port;

// Initialize Flas as TRUE
Flags[client] = 1;

while (Flags[client]) {

	// Receive the packet
	char buf[sizeof(MESSAGE)];
	int reRet = recv(client, buf, sizeof(buf), 0);
	if (reRet == -1) {
		cout << "receive failed:" << errno << endl;
		break;
	}

	// interpret the packet
	MESSAGE rec, sen;
	memcpy(&rec, buf, sizeof(MESSAGE));
	memset(sen.MES, 0, sizeof(sen.MES));
	sen.DES = client;
	sen.OPT = rec.OPT;

	// Process it according to its type
	switch (rec.OPT) {
	case DISCON:
		Flags[client] = 0;
		Clients_info.erase(client);
		cout << "Disconnected link with " << client << endl;
		break;
	case TIME:
		cout << "Send time to " << client << endl;
		cc = time(0);
		tmp_time = ctime(&cc);
		strcpy(rec.MES, tmp_time.c_str());
		break;
	case NAME:
		cout << "Send name to " << client << endl;
		gethostname(name, size);
		strcpy(rec.MES, name);
```

```
            break;
        case LINK:
            cout << "Send current active links to " << client << endl;
            // 通过 map 获取现有的活跃链接，并发送

            break;
        case SEND:
            cout << "Send message from " << client << " to " << rec.DES << endl;
            // 向目标地址发送消息

            break;
        default:
            cout << "Wrong requirement type!" << endl;
        }

        // Always reply the sender after processing the packet
        if (rec.OPT >= DISCON && rec.OPT <= SEND) {
            // 回复发出指令的客户端
        }
    }
```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

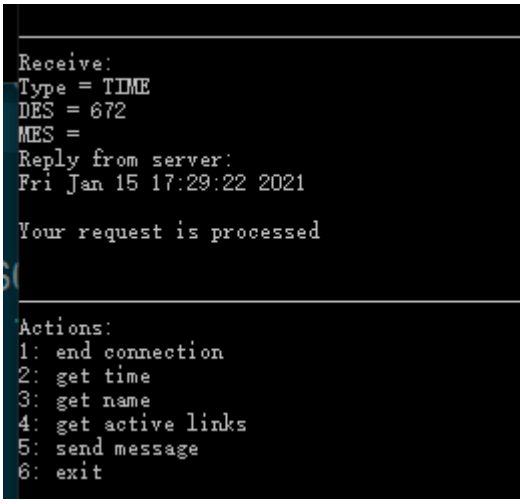


Wireshark 抓取的数据包截图：

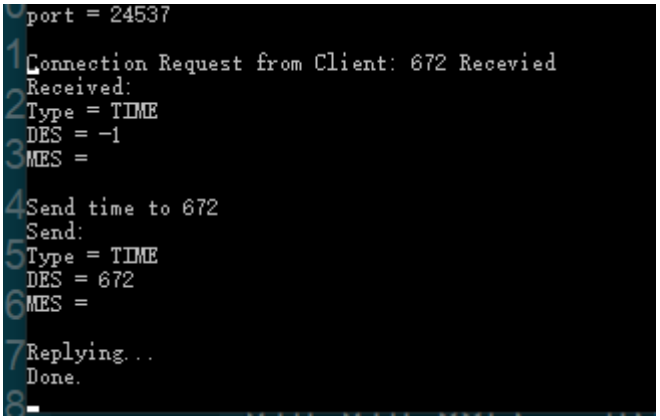
113	66.040368	127.0.0.1	127.0.0.1	TCP	64 62683 → 5438 [SYN] Se
114	66.040431	127.0.0.1	127.0.0.1	TCP	64 5438 → 62683 [SYN, AC
115	66.040464	127.0.0.1	127.0.0.1	TCP	56 62683 → 5438 [ACK] Se
116	66.954796	127.0.0.1	127.0.0.1	TCP	64 62684 → 5438 [SYN] Se
117	66.954854	127.0.0.1	127.0.0.1	TCP	64 5438 → 62684 [SYN, AC
118	66.954896	127.0.0.1	127.0.0.1	TCP	56 62684 → 5438 [ACK] Se
119	68.484666	127.0.0.1	127.0.0.1	TCP	64 62685 → 5438 [SYN] Se
120	68.484718	127.0.0.1	127.0.0.1	TCP	64 5438 → 62685 [SYN, AC
121	68.484769	127.0.0.1	127.0.0.1	TCP	56 62685 → 5438 [ACK] Se

- 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（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）：

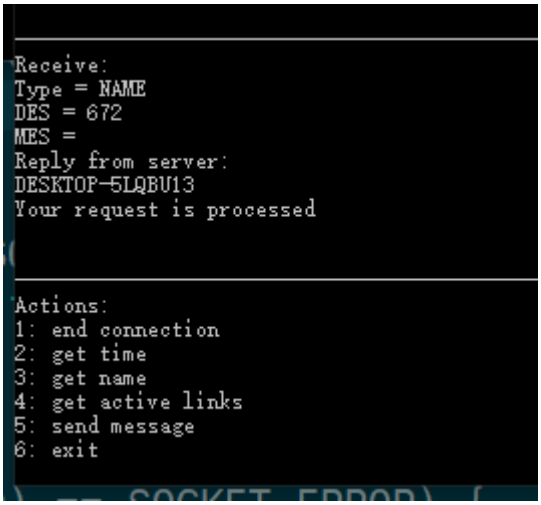
Wireshark · Packet 240 · Adapter for loopback traffic capture

```
    Acknowledgment Number: 2065    (relative ack number)
    Acknowledgment number (raw): 274863947
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x018 (PSH, ACK)
    Window: 10227
    [Calculated window size: 2618112]
    [Window size scaling factor: 256]
    Checksum: 0xa529 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  > [SEQ/ACK analysis]
  > [Timestamps]
    TCP payload (1032 bytes)
v Data (1032 bytes)
    Data: 9c020000010000005265706c792066726f6d207365727665723a0a467269204a616e2031…
    [Length: 1032]
```

```
0000  02 00 00 00 45 00 04 3c  5d 7d 40 00 40 06 00 00   ····E··< ]}@·@···
0010  7f 00 00 01 7f 00 00 01  15 3e f4 dc f9 99 b9 5c   ········ ·>·····\
0020  10 62 17 4b 80 18 27 f3  a5 29 00 00 01 01 08 0a   ·b·K··'· ·)······
0030  00 18 87 2d 00 18 87 2b  9c 02 00 00 01 00 00 00   ···-···+ ········
0040  52 65 70 6c 79 20 66 72  6f 6d 20 73 65 72 76 65   Reply fr om serve
0050  72 3a 0a 46 72 69 20 4a  61 6e 20 31 35 20 32 32   r:·Fri J an 15 22
0060  3a 32 36 3a 33 38 20 32  30 32 31 0a 0a 59 6f 75   :26:38 2 021··You
0070  72 20 72 65 71 75 65 73  74 20 69 73 20 70 72 6f   r reques t is pro
0080  63 65 73 73 65 64 0a 00  00 00 00 00 00 00 00 00   cessed·· ········
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
```

- 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。

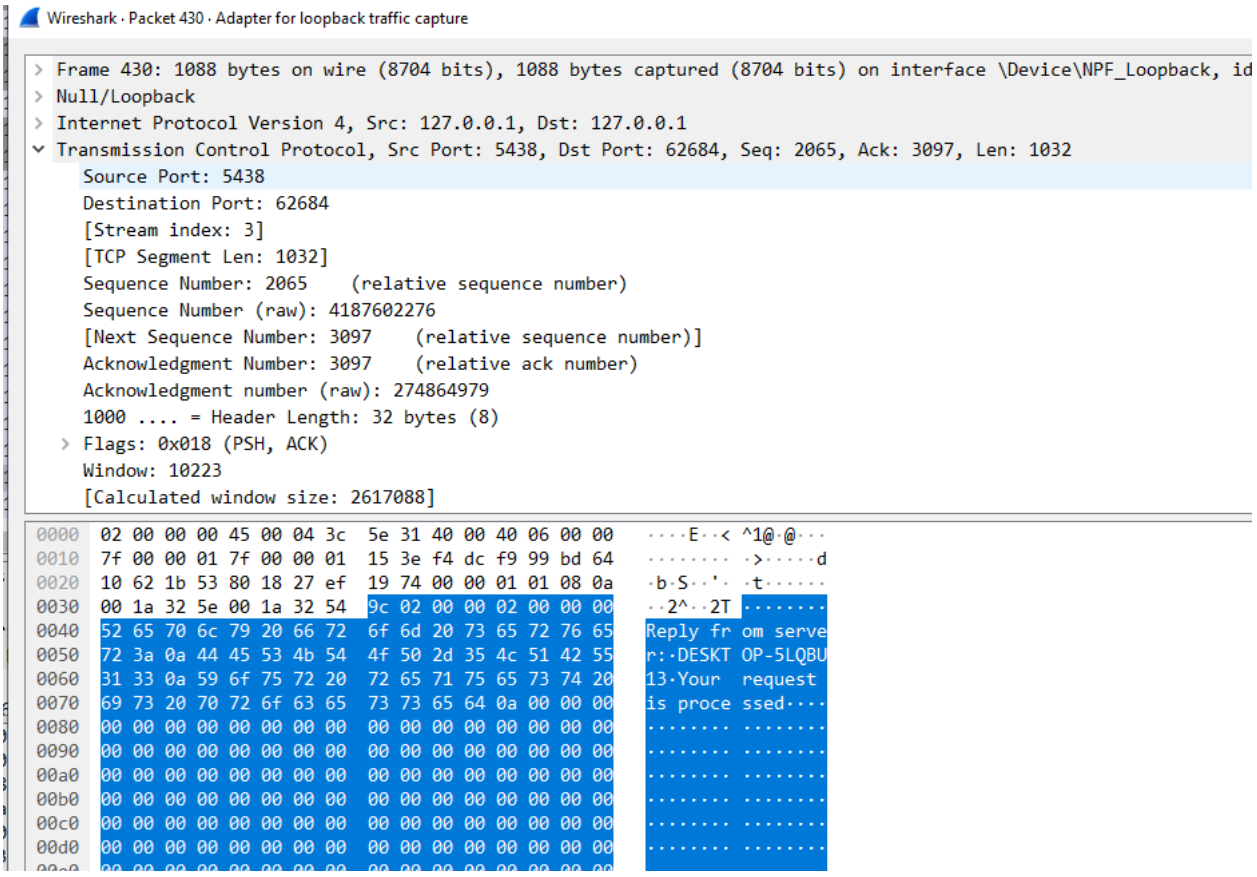
客户端:



服务端：

```
Replying...
Done.
Connection Request from Client: 672 Recevied
Received:
Type = NAME
DES = -1
MES =

Send name to 672
Send:
Type = NAME
DES = 672
MES =

Replying...
Done
```

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

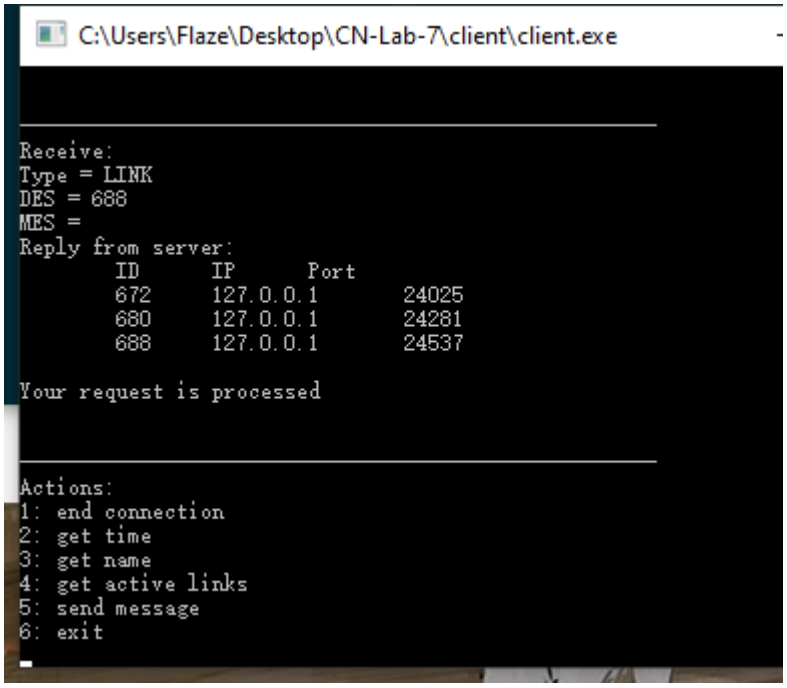


相关的服务器的处理代码片段：

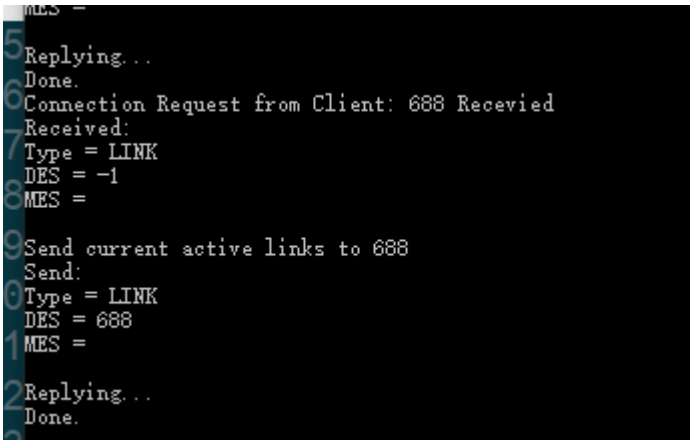
```
gethostname(name, size);

strcpy(rec.MES, name);
```

- 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。

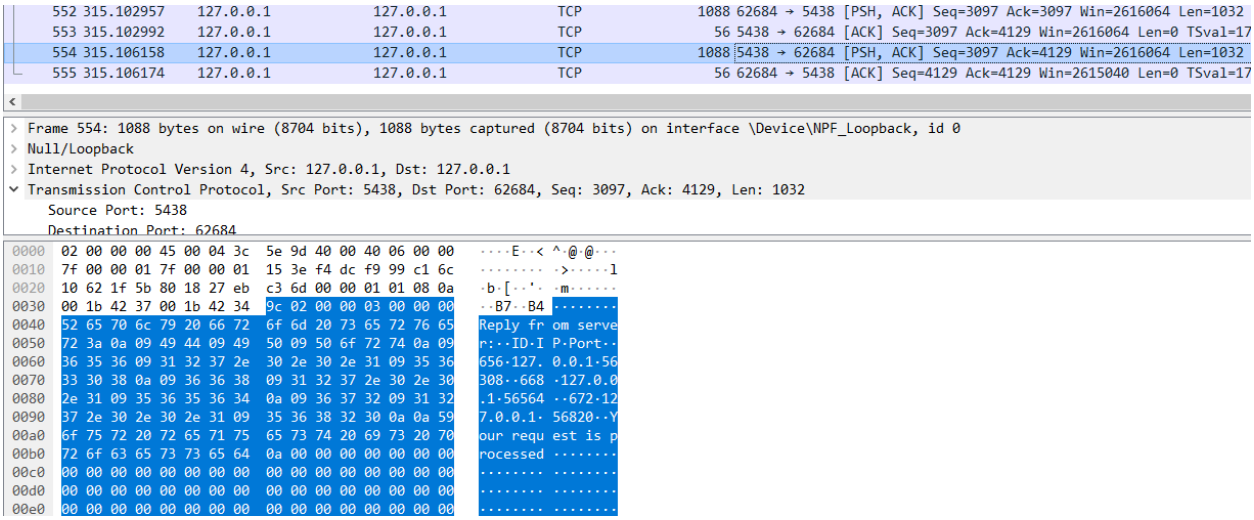
客户端：



服务端:



Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：

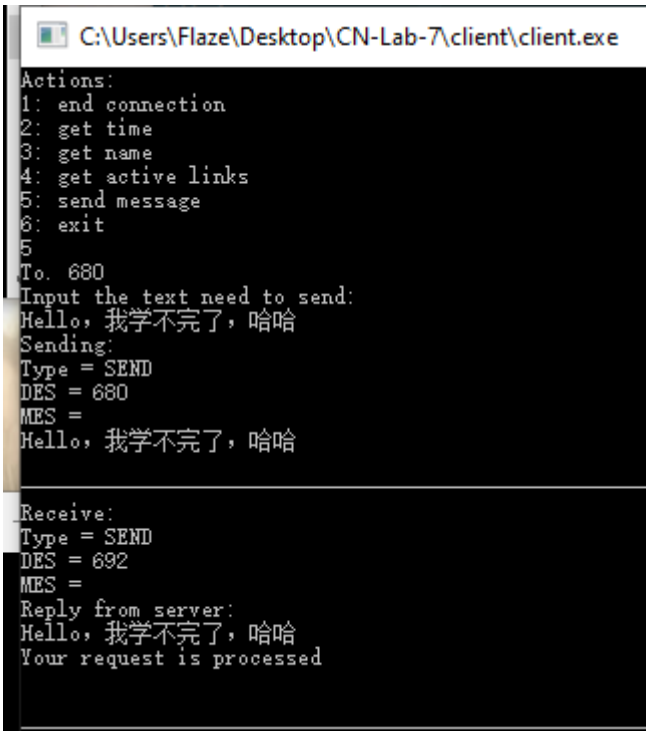


相关的服务器的处理代码片段：

```
tmp_link = "\tID\tIP\tPort\n";
for (auto i : Clients_info)
    tmp_link = tmp_link + "\t" + to_string(i.first) + "\t" + i.second.ip + "\t"
+ to_string(i.second.port) + "\n";
strcpy(rec.MES, tmp_link.c_str());
```

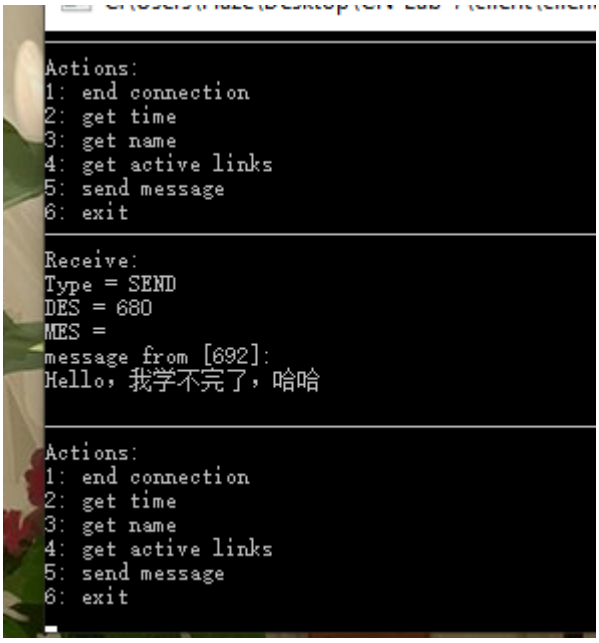
- 客户端选择发送消息功能时，客户端和服务端显示内容截图。

发送消息的客户端：



服务器：

接收消息的客户端：



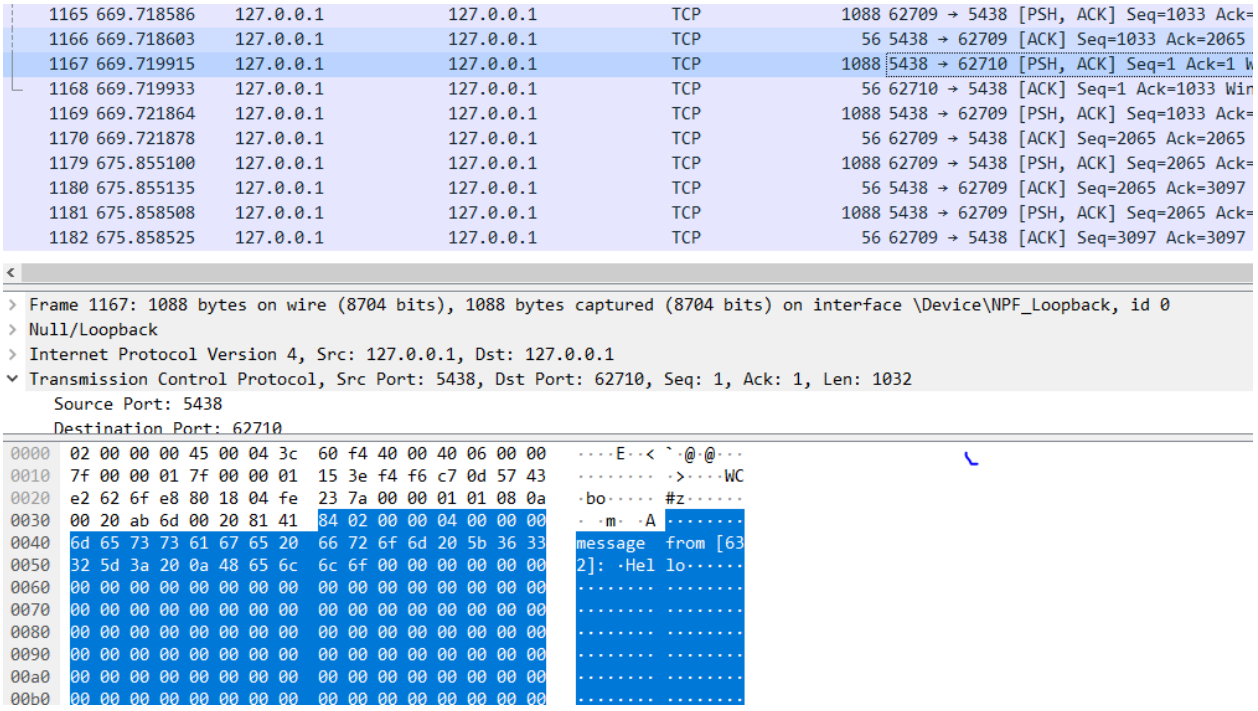
Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送:

1165	669.718586	127.0.0.1	127.0.0.1	TCP	1088	62709 → 5438 [PSH, ACK] Seq=1033
1166	669.718603	127.0.0.1	127.0.0.1	TCP	56	5438 → 62709 [ACK] Seq=1033 Ack=
1167	669.719915	127.0.0.1	127.0.0.1	TCP	1088	5438 → 62710 [PSH, ACK] Seq=1 Ac
1168	669.719933	127.0.0.1	127.0.0.1	TCP	56	62710 → 5438 [ACK] Seq=1 Ack=103
1169	669.721864	127.0.0.1	127.0.0.1	TCP	1088	5438 → 62709 [PSH, ACK] Seq=1033
1170	669.721878	127.0.0.1	127.0.0.1	TCP	56	62709 → 5438 [ACK] Seq=2065 Ack=
1179	675.855100	127.0.0.1	127.0.0.1	TCP	1088	62709 → 5438 [PSH, ACK] Seq=2065
1180	675.855135	127.0.0.1	127.0.0.1	TCP	56	5438 → 62709 [ACK] Seq=2065 Ack=
1181	675.858508	127.0.0.1	127.0.0.1	TCP	1088	5438 → 62709 [PSH, ACK] Seq=2065
1182	675.858525	127.0.0.1	127.0.0.1	TCP	56	62709 → 5438 [ACK] Seq=3097 Ack=

```
Frame 1165: 1088 bytes on wire (8704 bits), 1088 bytes captured (8704 bits) on interface \Device\NPF_Loopback, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 62709, Dst Port: 5438, Seq: 1033, Ack: 1033, Len: 1032
    Source Port: 62709
    Destination Port: 5438
0000  02 00 00 00 45 00 04 3c  60 f2 40 00 40 06 00 00   ····E··< `·@·@···
0010  7f 00 00 01 7f 00 00 01  f4 f5 15 3e b5 95 93 20   ········ ···>···
0020  66 86 73 32 80 18 27 f3  36 c8 00 00 01 01 08 0a   f·s2··'· 6·······
0030  00 20 ab 6b 00 20 91 cc  84 02 00 00 04 00 00 00   · ·k· ·· ········
0040  48 65 6c 6c 6f 00 00 00  00 00 00 00 00 00 00 00   Hello··· ········
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
```

接受：



相关的服务器的处理代码片段：

```
cout << "Send message from " << client << " to " << rec.DES << endl;
tmp_text = rec.MES;
tmp_text = "message from [" + to_string(client) + "]: \n" + tmp_text;
memset(sen.MES, 0, sizeof(sen.MES));
strcpy(sen.MES, tmp_text.c_str());
sen.DES = rec.DES;

cout << "Start sending..." << endl;
ret = send(sen.DES, (char*)&sen, sizeof(sen), 0);
if (ret < 0) {
    cout << "send failed" << endl;
    strcpy(rec.MES, "Sending Failed");
}
```

相关的客户端（发送和接收消息）处理代码片段：

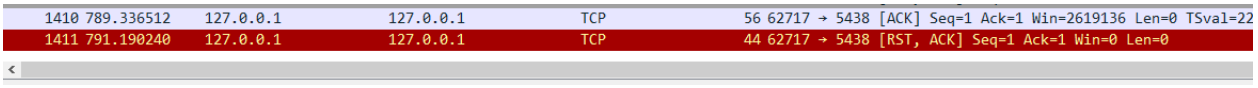
发送:

```
sen.OPT = SEND ;
cout << "To. " ;
cin >> des_num ;
sen.DES = des_num ;
cout << "Input the text need to send:\n" ;
cin >> Text ;
strcpy(sen.MES, Text.c_str()) ;
```

接受就是前面接受服务器消息的方法。

- 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。

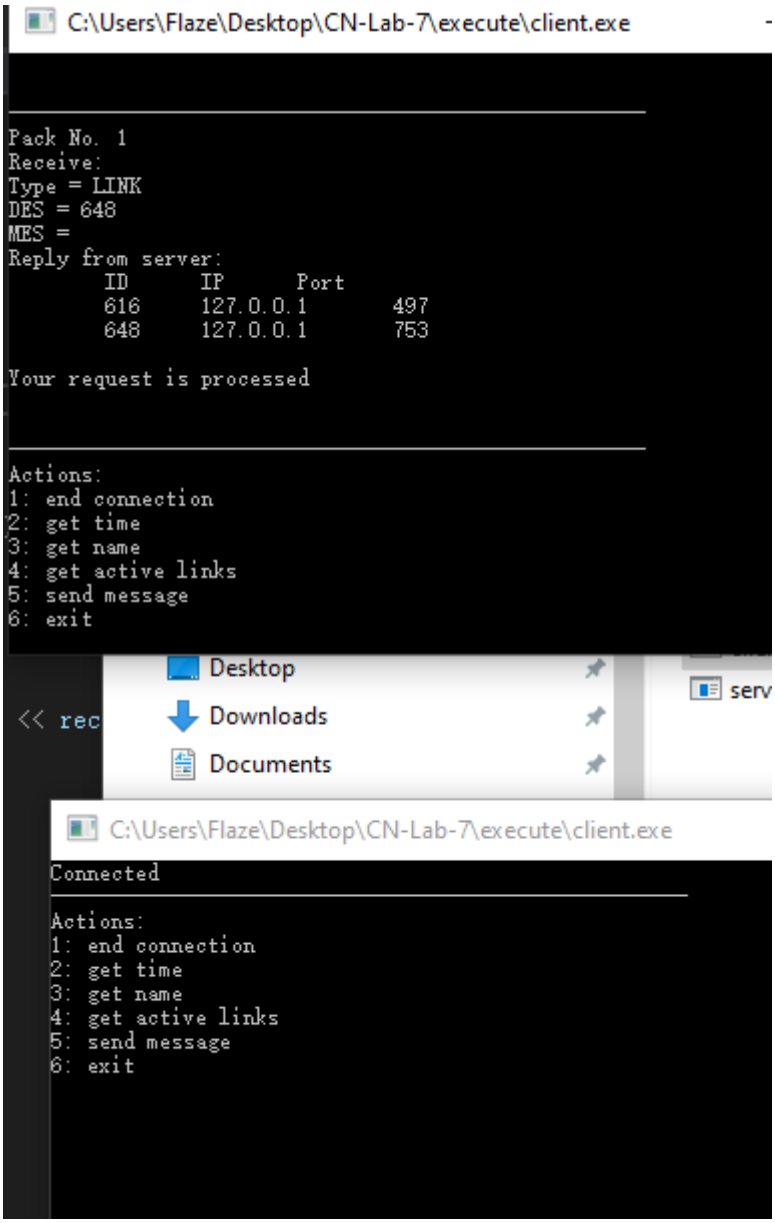
直接用关闭窗口的方式关掉了，有释放连接。



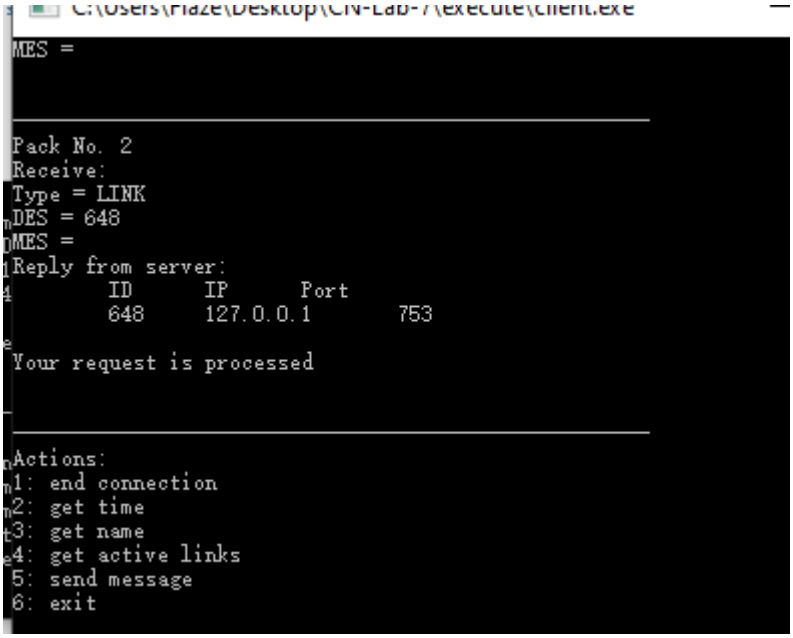
- 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

没有了该线程对应的 while 循环中会在链接断开的时候立即得到链接失败的信息。

断开前：


C:\Users\Flaze\Desktop\CN-Lab-7\execute\client.exe
Pack No. 1
Receive:
Type = LINK
DES = 648
MES =
Reply from server:
ID IP Port
616 127.0.0.1 497
648 127.0.0.1 753
Your request is processed
Actions:
1: end connection
2: get time
3: get name
4: get active links
5: send message
6: exit
Desktop
Downloads
Documents
C:\Users\Flaze\Desktop\CN-Lab-7\execute\client.exe
Connected
Actions:
1: end connection
2: get time
3: get name
4: get active links
5: send message
6: exit

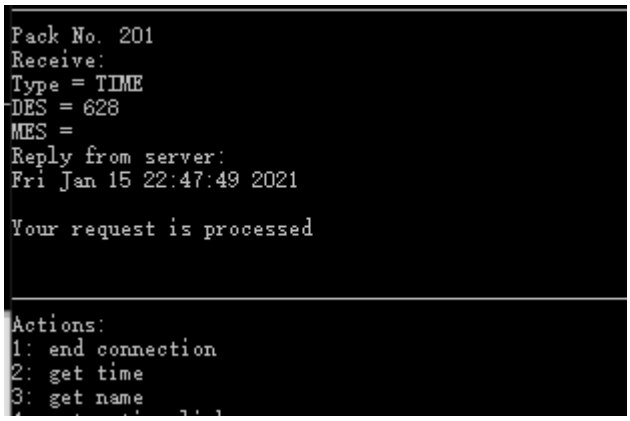
断开后：



- 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。

开始：

结束：



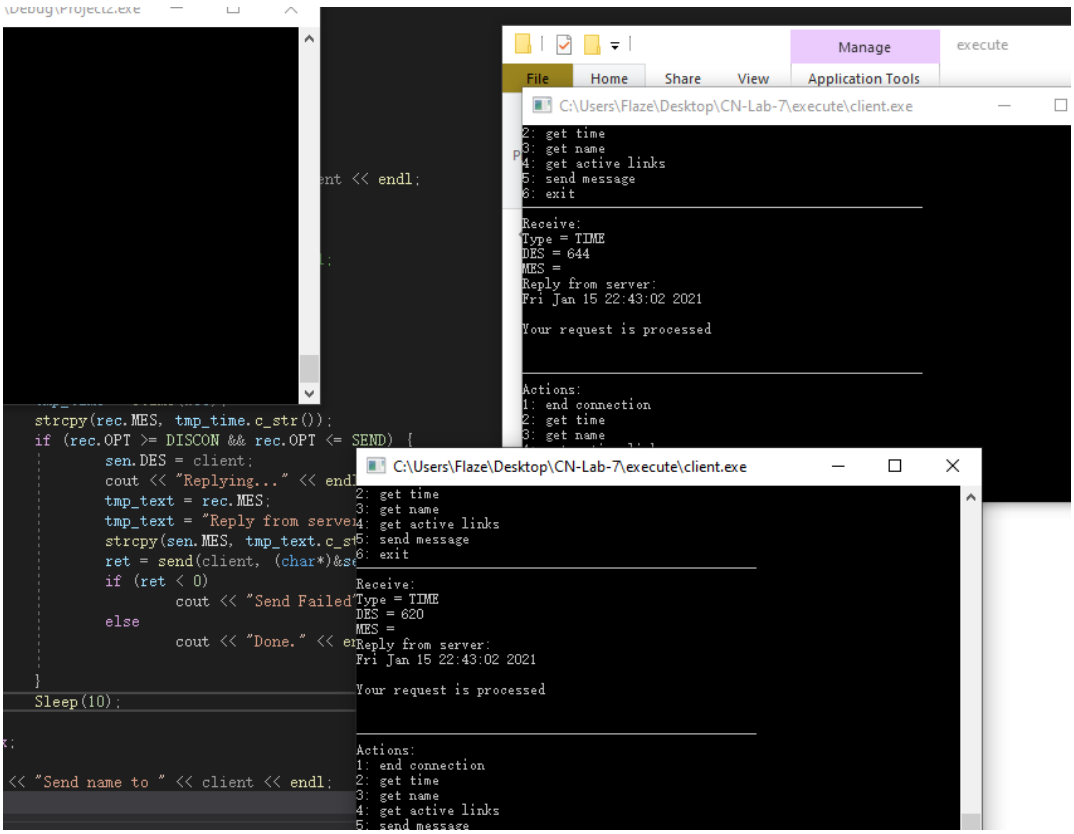
开始：

3436	1120.334063	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [AC
3557	1188.647731	127.0.0.1	127.0.0.1	TCP	1088	49223 → 5438 [PS
3558	1188.647767	127.0.0.1	127.0.0.1	TCP	56	5438 → 49223 [AC

结束：

3758	1190.222243	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=41
3759	1190.238742	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] S
3760	1190.238760	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=41

- 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



tcp.port==5438

No.	Time	Source	Destination	Protocol	Length	Info
3254	1119.291558	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=139321 Ack=2065 Win=
3255	1119.291578	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [ACK] Seq=2065 Ack=140353 Win=2610
3256	1119.291591	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=3097 Ack=287929 Win=2595
3257	1119.306237	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] Seq=287929 Ack=3097 Win=
3258	1119.306244	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=140353 Ack=2065 Win=
3259	1119.306273	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [ACK] Seq=2065 Ack=141385 Win=2609
3260	1119.306284	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=3097 Ack=288961 Win=2594
3261	1119.321996	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=141385 Ack=2065 Win=
3262	1119.322014	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] Seq=288961 Ack=3097 Win=
3263	1119.322020	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [ACK] Seq=2065 Ack=142417 Win=2608
3264	1119.322035	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=3097 Ack=289993 Win=2593
3265	1119.337364	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] Seq=289993 Ack=3097 Win=
3266	1119.337388	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=142417 Ack=2065 Win=
3267	1119.337406	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [ACK] Seq=2065 Ack=143449 Win=2607
3268	1119.337417	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=3097 Ack=291025 Win=2592
3269	1119.353462	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=143449 Ack=2065 Win=
3270	1119.353480	127.0.0.1	127.0.0.1	TCP	56	49224 → 5438 [ACK] Seq=2065 Ack=144481 Win=2606
3271	1119.353489	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] Seq=291025 Ack=3097 Win=
3272	1119.353502	127.0.0.1	127.0.0.1	TCP	56	49223 → 5438 [ACK] Seq=3097 Ack=292057 Win=2591
3273	1119.369354	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49224 [PSH, ACK] Seq=144481 Ack=2065 Win=
3274	1119.369360	127.0.0.1	127.0.0.1	TCP	1088	5438 → 49223 [PSH, ACK] Seq=292057 Ack=3097 Win=

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？

不需要，不是，client 的端口是自动分配的。

- 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。
  Accept 是取出链接的，不影响 connect

- 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

- 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？

  客户端的端口是不同的，可以以此区分

- 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）

  立即断开，几乎没有保持多久。

- 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？

  TCP 会断开，服务器在当前线程一直在询问链接的状态，如果断开也会及时反馈。

七、 讨论、心得

这，感觉上课和之前的实验……和这个完全脱节了啊…做得一脸茫然 QAQ……然后对着参考瞎写

一通，我人没了.jpg

做完准备截抓包图的时候发现 wireshark 没东西……只好重新整了一遍（重装 wireshark 啥的……）

Client 的编译命令要加一个 “-lwsock32 -std=c++11”

本来发现 client 可以直接命令行编译还想抛弃 vs2019 来着，结果 server 读 ip 的部分照着 c++20 写了，没时间搞一个支持 c++20 的 g++，只好一个用命令行编译一个用 vs 了，也还行，反正直接用 exe 就好。

在做讨论题的时候意识到好像可以通过在每一个线程中记录当前 socket 的 id，在断开链接时也许可以删除数据库里 active linkage 的信息，于是及时进行了修改（但是前面部分来不及全部改完）

添加了这一部分：

```
while (Flags[client]) {

        // Receive the packet
        char buf[sizeof(MESSAGE)];
        int reRet = recv(client, buf, sizeof(buf), 0);
        if (reRet == -1) {
                Flags[client] = 0;
                Clients_info.erase(client);
                cout << "receive failed:" << errno << endl;
                break;
        }
        cout << "Connection Request from Client: " << to_string(client) << " Recev

        // interpret the packet
        MESSAGE rec, sen;
```