

# 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: 李云帆

学 院: 计算机学院

系:

专 业: 计算机科学与技术

学 号: 3200102555

指导教师: 邱劲松

2022 年 12 月 31 日

# 浙江大学实验报告

实验名称: 基于 Socket 接口实现自定义协议通信 实验类型: 编程实验

同组学生: \_\_\_\_\_ 实验地点: 计算机网络实验室

## 一、 实验目的

- 学习如何设计网络应用协议
- 掌握 Socket 编程接口编写基本的网络应用软件

## 二、 实验内容

根据自定义的协议规范, 使用 Socket 编程接口编写基本的网络应用软件。

- 掌握 C 语言形式的 Socket 编程接口用法, 能够正确发送和接收网络数据包
- 开发一个客户端, 实现人机交互界面和与服务器的通信
- 开发一个服务端, 实现并发处理多个客户端的请求
- 程序界面不做要求, 使用命令行或最简单的窗体即可
- 功能要求如下:
  1. 运输层协议采用 TCP
  2. 客户端采用交互菜单形式, 用户可以选择以下功能:
    - a) 连接: 请求连接到指定地址和端口的服务端
    - b) 断开连接: 断开与服务端的连接
    - c) 获取时间: 请求服务端给出当前时间
    - d) 获取名字: 请求服务端给出其机器的名称
    - e) 活动连接列表: 请求服务端给出当前连接的所有客户端信息 (编号、IP 地址、端口等)
    - f) 发消息: 请求服务端把消息转发给对应编号的客户端, 该客户端收到后显示在屏幕上
    - g) 退出: 断开连接并退出客户端程序
  3. 服务端接收到客户端请求后, 根据客户端传过来的指令完成特定任务:
    - a) 向客户端传送服务端所在机器的当前时间
    - b) 向客户端传送服务端所在机器的名称
    - c) 向客户端传送当前连接的所有客户端信息
    - d) 将某客户端发送过来的内容转发给指定编号的其他客户端
    - e) 采用异步多线程编程模式, 正确处理多个客户端同时连接, 同时发送消息的情况
- 根据上述功能要求, 设计一个客户端和服务端之间的应用通信协议
- **本实验涉及到网络数据包发送部分不能使用任何的 Socket 封装类, 只能使用最底层的 C 语言形式的 Socket API**
- 本实验可组成小组, 服务端和客户端可由不同人来完成

## 三、 主要仪器设备

- 联网的 PC 机、Wireshark 软件
- Visual C++、gcc 等 C++集成开发环境。

#### 四、操作方法与实验步骤

- 设计请求、指示（服务器主动发给客户端的）、响应数据包的格式，至少要考虑如下问题：
  - a) 定义两个数据包的边界如何识别
  - b) 定义数据包的请求、指示、响应类型字段
  - c) 定义数据包的长度字段或者结尾标记
  - d) 定义数据包内数据字段的格式（特别是考虑客户端列表数据如何表达）
- 小组分工：1 人负责编写服务端，1 人负责编写客户端
- 客户端编写步骤（需要采用多线程模式）
  - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
  - b) 编写一个菜单功能，列出 7 个选项
  - c) 等待用户选择
  - d) 根据用户选择，做出相应的动作（未连接时，只能选连接功能和退出功能）
    1. 选择连接功能：请用户输入服务器 IP 和端口，然后调用 `connect()`，等待返回结果并打印。连接成功后设置连接状态为已连接。然后创建一个接收数据的子线程，循环调用 `receive()`，如果收到了一个完整的响应数据包，就通过线程间通信（如消息队列）发送给主线程，然后继续调用 `receive()`，直至收到主线程通知退出。
    2. 选择断开功能：调用 `close()`，并设置连接状态为未连接。通知并等待子线程关闭。
    3. 选择获取时间功能：组装请求数据包，类型设置为时间请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印时间信息。
    4. 选择获取名字功能：组装请求数据包，类型设置为名字请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印名字信息。
    5. 选择获取客户端列表功能：组装请求数据包，类型设置为列表请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印客户端列表信息（编号、IP 地址、端口等）。
    6. 选择发送消息功能（选择前需要先获得客户端列表）：请用户输入客户端的列表编号和要发送的内容，然后组装请求数据包，类型设置为消息请求，然后调用 `send()` 将数据发送给服务器，接着等待接收数据的子线程返回结果，并根据响应数据包的内容，打印消息发送结果（是否成功送达另一个客户端）。
    7. 选择退出功能：判断连接状态是否为已连接，是则先调用断开功能，然后再退出程序。否则，直接退出程序。
    8. 主线程除了在等待用户的输入外，还在处理子线程的消息队列，如果有消息到达，则进行处理，如果是响应消息，则打印响应消息的数据内容（比如时间、名字、客户端列表等）；如果是指示消息，则打印指示消息的内容（比如服务器转发的别的客户端的消息内容、发送者编号、IP 地址、端口等）。
- 服务端编写步骤（需要采用多线程模式）
  - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
  - b) 调用 `bind()`，绑定监听端口（请使用学号的后 4 位作为服务器的监听端口），接着调用 `listen()`，设置连接等待队列长度
  - c) 主线程循环调用 `accept()`，直到返回一个有效的 `socket` 句柄，在客户端列表中增加一个新客户端的项目，并记录下该客户端句柄和连接状态、端口。然后创建一个子线程后继续调用 `accept()`。该子线程的主要步骤是（刚获得的句柄要传递给子线程，子线程内部要使用该句柄发送和接收数据）：

- ✧ 调用 `send()`，发送一个 `hello` 消息给客户端（可选）
- ✧ 循环调用 `receive()`，如果收到了一个完整的请求数据包，根据请求类型做相应的动作：
  1. 请求类型为获取时间：调用 `time()` 获取本地时间，然后将时间数据组装进响应数据包，调用 `send()` 发给客户端
  2. 请求类型为获取名字：将服务器的名字组装进响应数据包，调用 `send()` 发给客户端
  3. 请求类型为获取客户端列表：读取客户端列表数据，将编号、IP 地址、端口等数据组装进响应数据包，调用 `send()` 发给客户端
  4. 请求类型为发送消息：根据编号读取客户端列表数据，如果编号不存在，将错误代码和出错描述信息组装进响应数据包，调用 `send()` 发回源客户端；如果编号存在并且状态是已连接，则将要转发的消息组装进指示数据包。调用 `send()` 发给接收客户端（使用接收客户端的 `socket` 句柄），发送成功后组装转发成功的响应数据包，调用 `send()` 发回源客户端。
- d) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 `Socket`，主程序退出。
- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

## 五、实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

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

以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- 描述请求数据包的格式（画图说明），请求类型的定义

type	IP_addr	^	port	\$
content				

- 描述响应数据包的格式（画图说明），响应类型的定义

type			
content			
IP_addr	^	port	\$
IP_addr	^	port	\$
IP_addr	^	port	\$

type 表示类型值，定义关系如下：

type	
11	time
12	server name
13	server list
14	response

- 描述指示数据包的格式（画图说明），指示类型的定义

type	content
------	---------

- 客户端初始运行后显示的菜单选项

```
hejiajun@luanr_mbp src % ./client
Please input a command:
- connect [IP] [port]
- close the connection
- get server time
- get server name
- get clients
- send [IP] [port] [content]
- exit
- help
```

- 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）

```

27 while (1)
28 {
29     std::cout<<"> ";
30     std::string line;
31     getline(std::cin, line);
32     // split
33     std::regex whitespace("\\s+");
34     std::vector<std::string> words(std::sregex_token_iterator(line.begin(), line.end(), whitespace, -1),
35                                   std::sregex_token_iterator());
36     if (words[0] == "")
37     {
38         continue;
39     }
40     else if (words[0] == "connect")
41     {
42         if (-1 != sockfd)
43         {
44             std::cout<<RED<<"Connected!\n"<<NORMAL;
45         }
46         else
47         {
48             connect(words[1], std::stoi(words[2]));
49         }
50     }
51     else if (words[0] == "close the connection")
52     {
53         if (-1 == sockfd)
54         {
55             std::cout<<RED<<"No connection.\n"<<NORMAL;
56         }
57         else
58         {
59             char buffer = 50;
60             send(sockfd, &buffer, sizeof(buffer), 0);
61             mt.lock();
62             pthread_cancel(connection_thread);
63             mt.unlock();
64             close(sockfd);
65             sockfd = -1;
66             std::cout<<GREEN<<"Connection closed.\n"<<NORMAL;
67         }
68     }
69     else if (words[0] == "get server time")
70     {
71         char buffer = 1;
72         messageStruct timemsg;
73         send(sockfd, &buffer, sizeof(buffer), 0);
74         msgrcv(msqid, &timemsg, BUFFER_MAX, 11, 0);
75         time_t time = atol(timemsg.text);
76         std::cout<<YELLOW<<" Server time: "<<ctime(&time)<<NORMAL;
77     }
78     else if (words[0] == "get server name")
79     {
80         char buffer = 2;
81         send(sockfd, &buffer, sizeof(buffer), 0);
82         messageStruct namemsg;
83         msgrcv(msqid, &namemsg, BUFFER_MAX, 12, 0);
84         std::cout<<YELLOW<<"Server host name: "<<namemsg.text<<'\n'<<NORMAL;
85     }
86     else if (words[0] == "get clients")
87     {
88         char buffer = 3;
89         send(sockfd, &buffer, sizeof(buffer), 0);
90         messageStruct peermsg;
91         msgrcv(msqid, &peermsg, BUFFER_MAX, 13, 0);
92     }
93 }

```

主体部分的工作原理就是不断的读入这几种指令, 然后根据这些指令调用对应函数或执行对应操作。

- 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)

```

messageStruct msg;
key_t key = ftok("/", 'a');
int msqid = msgget(key, IPC_CREAT | 0666);
while (1)
{
    memset(buffer, 0, BUFFER_MAX);
    recv(cfd, buffer, BUFFER_MAX, 0);
    if (20 == buffer[0])
    {
        std::cout<<buffer + 1<<'\n';
        continue;
    }
    msg.type = buffer[0];
    strcpy(msg.text, buffer + 1);
    msgsnd(msqid, &msg, BUFFER_MAX, 0);
}

```

用 buffer 存储子线程。

- 服务器初始运行后显示的界面

```

[hejiajun@lunar_mbp src % ./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）

```

while (true)
{
    sockaddr_in client;
    unsigned int clientAddrLength = sizeof(client);
    int connection_fd = accept(sockfd, (sockaddr*)&client, (socklen_t*)&clientAddrLength);
    clientlist.push_back(std::pair<int, ip_port>(connection_fd, ip_port(inet_ntoa(client.sin_addr), ntohs(client.sin_port))));
    std::cout<<inet_ntoa(client.sin_addr)<<": "<<ntohs(client.sin_port)<<" connected.\n";
    pthread_t connection_thread;
    pthread_create(&connection_thread, nullptr, thread_handle, &connection_fd);
}

```

- 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）

根据 buffer\_recv 的值判断用户需求，根据不同的需求调用对应的函数或进行相应的操作。



```

while (1)
{
    recv(cfd, buffer_recv, BUFFER_MAX, 0);
    memset(buffer_send, 0, BUFFER_MAX);
    mt.lock();
    switch (buffer_recv[0])
    {
    case 0:
    {
        for (auto it = clientList.begin(); it != clientList.end(); )
        {
            if ((*it).first == cfd)
            {
                it = clientList.erase(it);
                break;
            }
            else ++it;
        }
        break;
    }
    case 1:
    {
        time_t t;
        time(&t);
        std::cout<<cfd<<" get server time: "<<t<<"\n";
        buffer_send[0] = 11;
        sprintf(buffer_send + strlen(buffer_send), "%ld", t);
        send(cfd, buffer_send, strlen(buffer_send), 0);
        break;
    }
    case 2:
    {
        std::cout<<cfd<<" get server name"<<"\n";
        buffer_send[0] = 12;
        gethostname(buffer_send + strlen(buffer_send), sizeof(buffer_send) - sizeof(char));
        send(cfd, buffer_send, strlen(buffer_send), 0);
        break;
    }
    case 3:
    {
        std::cout<<cfd<<" get clients"<<"\n";
        buffer_send[0] = 13;
        for (auto& it: clientList)
        {
            sprintf(buffer_send + strlen(buffer_send), "%s", it.second.first.c_str());
            sprintf(buffer_send + strlen(buffer_send), "^");
            sprintf(buffer_send + strlen(buffer_send), "%d", it.second.second);
            sprintf(buffer_send + strlen(buffer_send), "$");
        }
        send(cfd, buffer_send, strlen(buffer_send), 0);
        break;
    }
    case 4:
    {
        std::string msg(buffer_recv + 1);
        size_t pos0 = msg.find("^"), pos1 = msg.find("$");
        std::string ip_addr = msg.substr(0, pos0);
        int port = atoi(msg.substr(pos0 + 1, pos1 - pos0 - 1).c_str());
        std::string content = msg.substr(pos1 + 1);
    }
    }
}

```

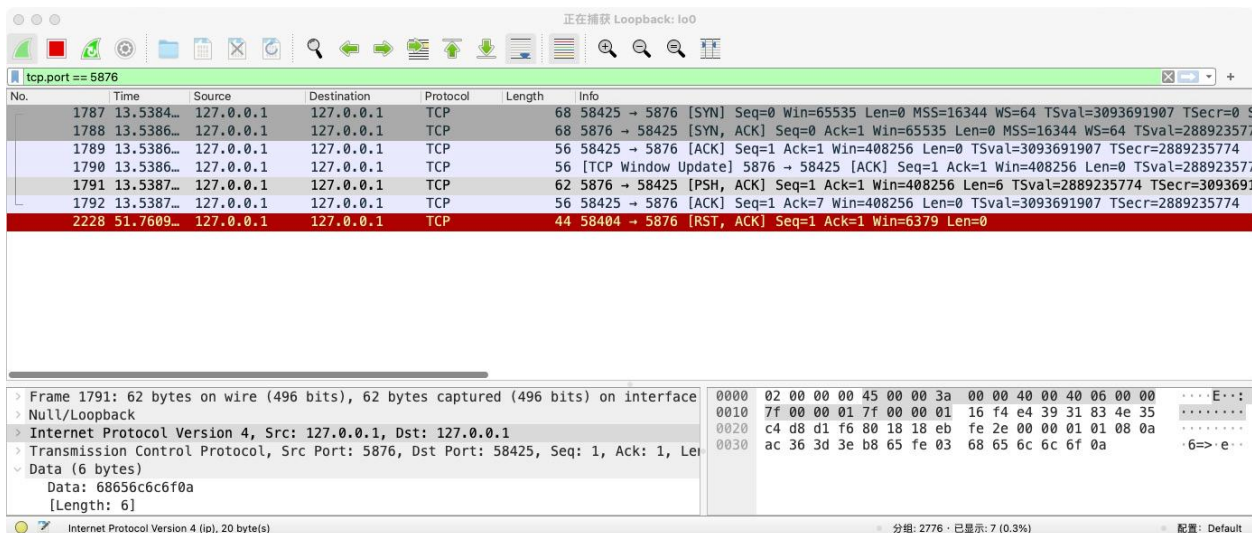


- 客户端选择连接功能时，客户端和服务端显示内容截图。

```
> connect 127.0.0.1 5876
> hello
>

127.0.0.1:58425 connected.
]
```

Wireshark 抓取的数据包截图：

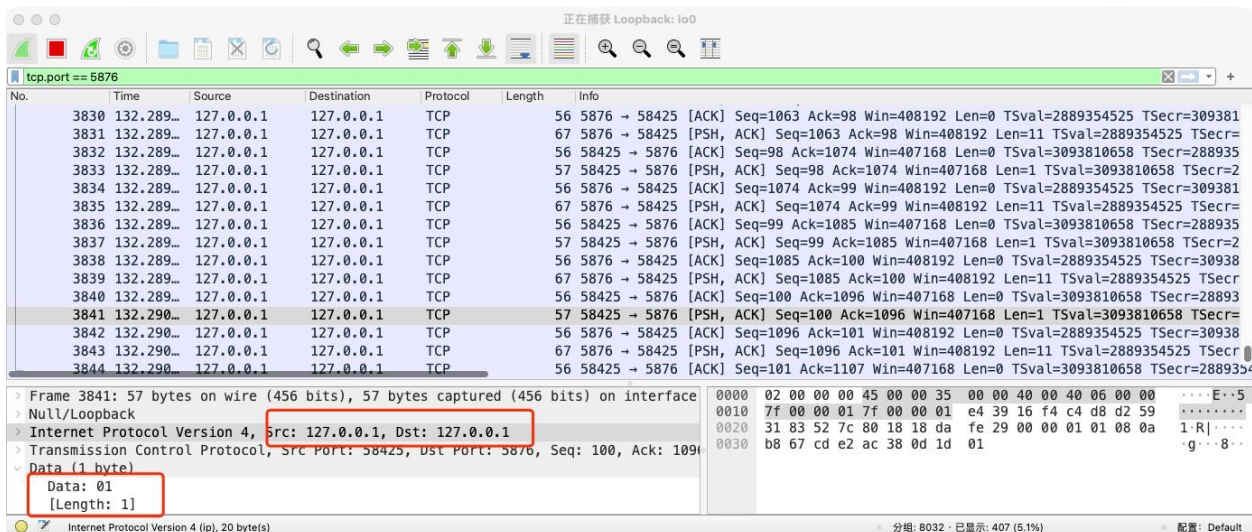


- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

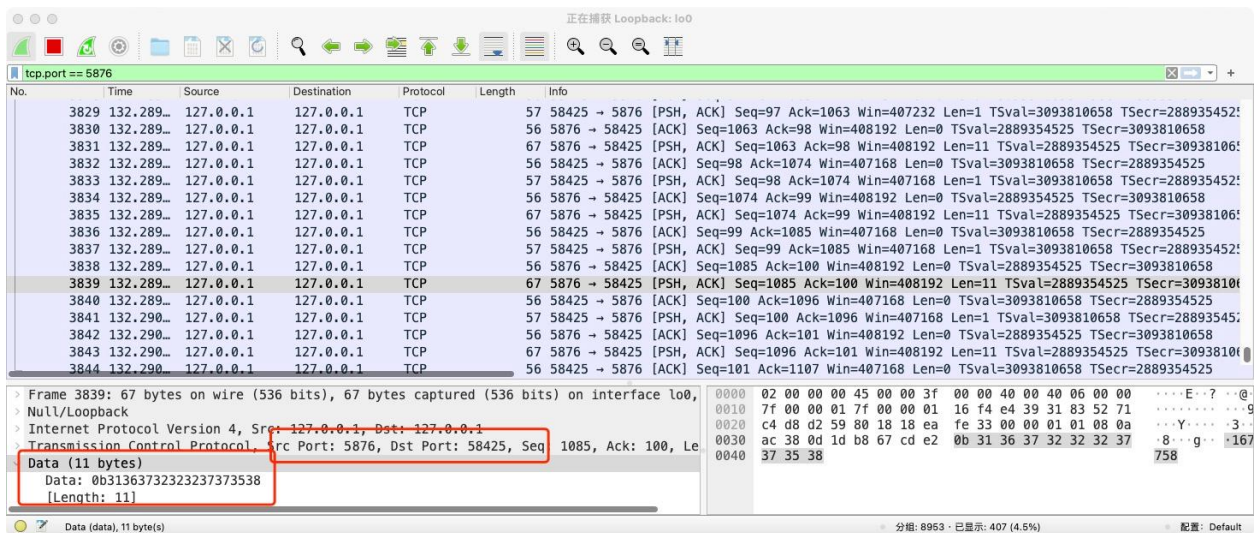
```
100 times: Server time: Wed Dec 28 19:42:38 2022
5 get server time: 1672227758
```

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

请求包：



响应包：

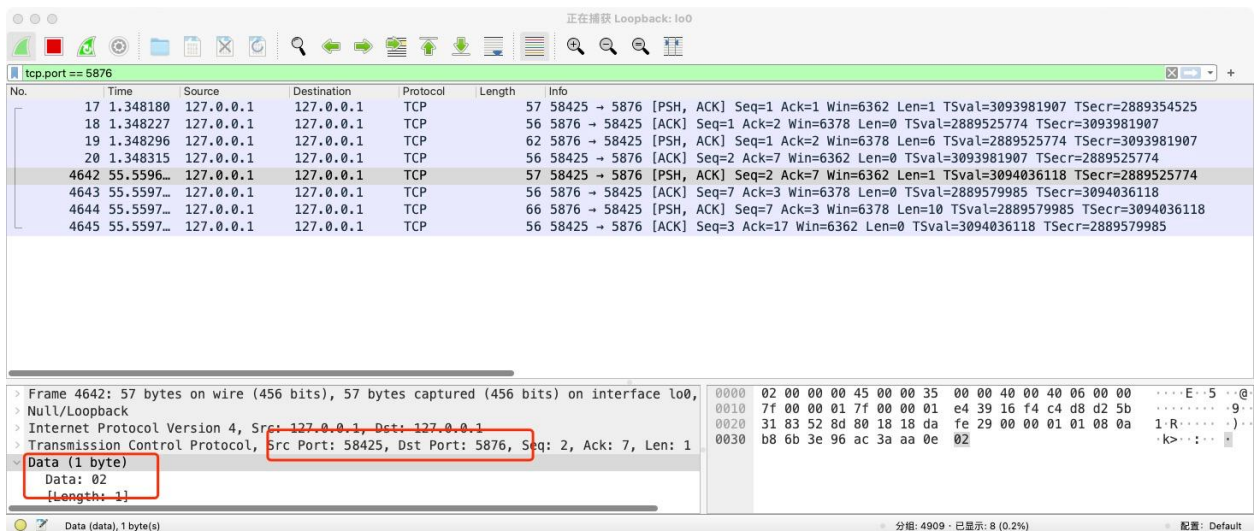


- 客户端选择获取名字功能时，客户端和服务端显示内容截图。

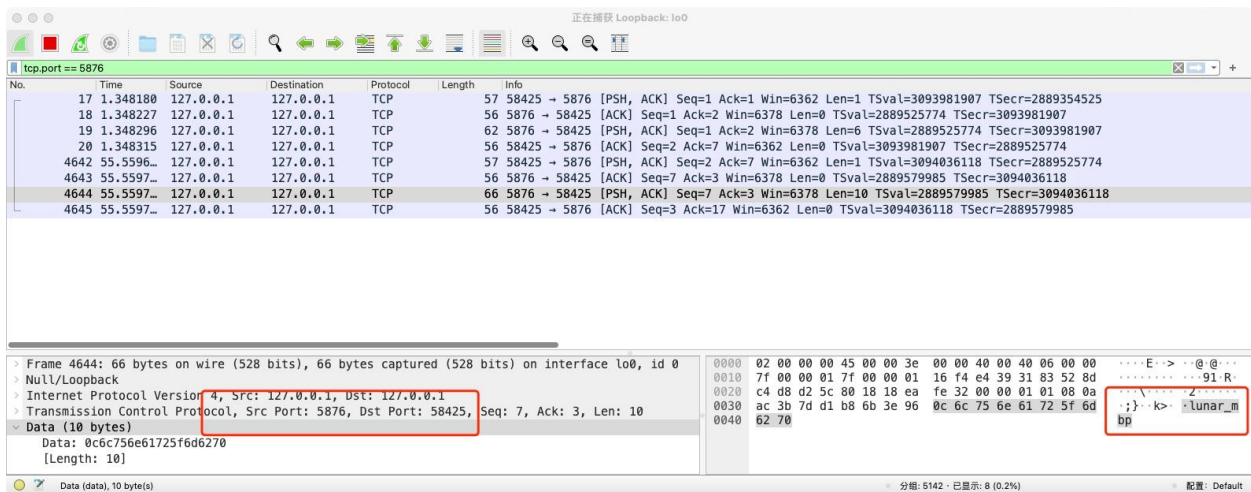
```
> get_server_name
Server host name: lunar_mbp
5 get server name
```

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

请求包：

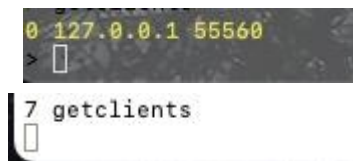


响应包：

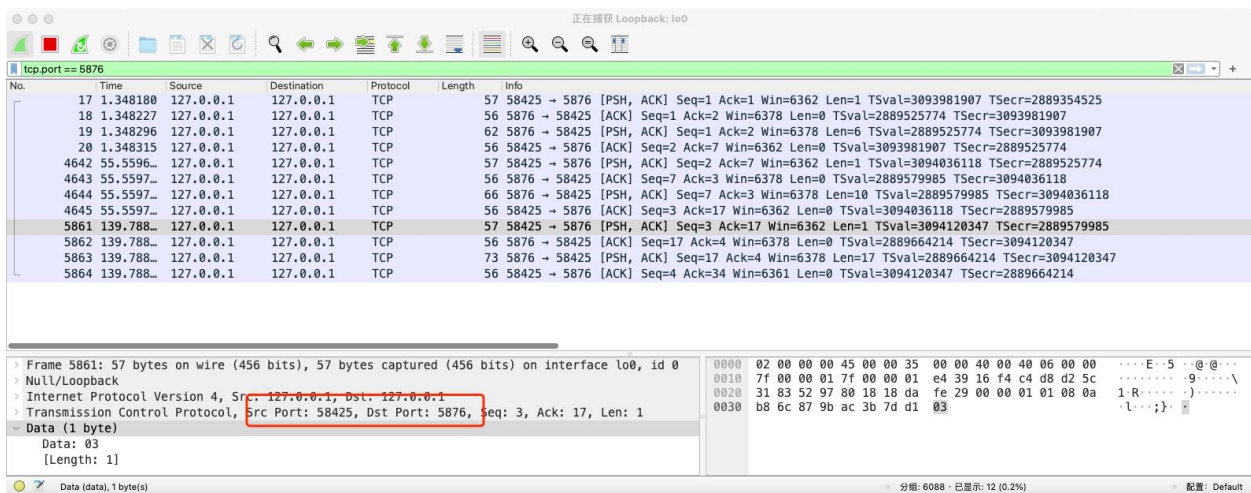


```
std::cout<<cfid<<" get server name"<<'\n';
buffer_send[0] = 12;
gethostname(buffer_send + strlen(buffer_send), sizeof(buffer_send) -
sizeof(char));
send(cfd, buffer_send, strlen(buffer_send), 0);
break;
```

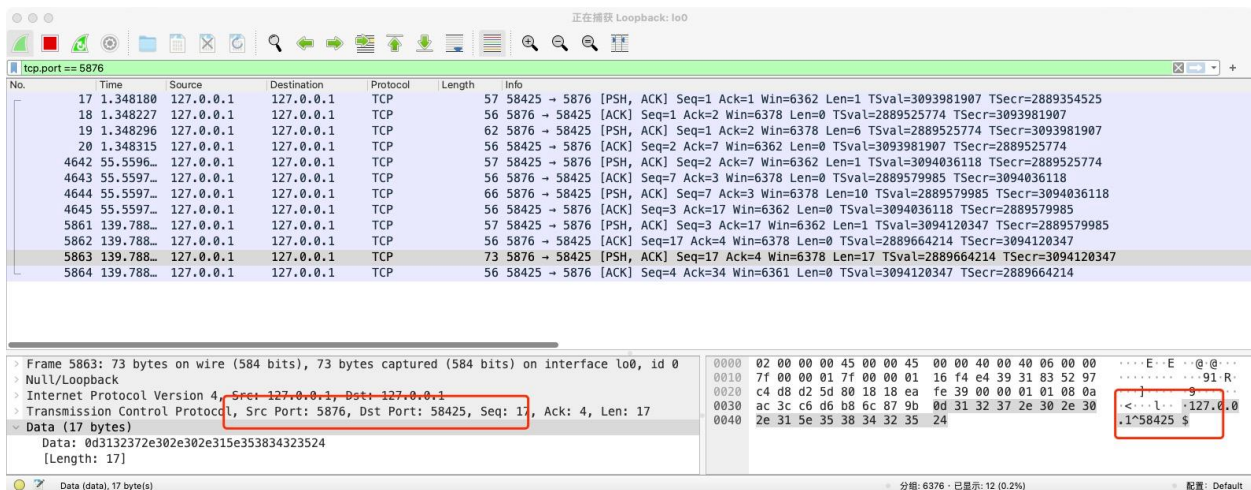
- 客户端选择获取客户端列表功能时，客户端和服务端显示内容截图。



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:

```
std::cout<<cfid<<" get server clients"<<'\n';
    buffer_send[0] = 13;
    for (auto& it: clientList)
    {
        sprintf(buffer_send + strlen(buffer_send), "%s",
it.second.first.c_str());
        sprintf(buffer_send + strlen(buffer_send), "^");
        sprintf(buffer_send + strlen(buffer_send), "%d",
it.second.second);
        sprintf(buffer_send + strlen(buffer_send), "$");
    }
    send(cfd, buffer_send, strlen(buffer_send), 0);
    break;
```

- 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&接收消息的客户端:

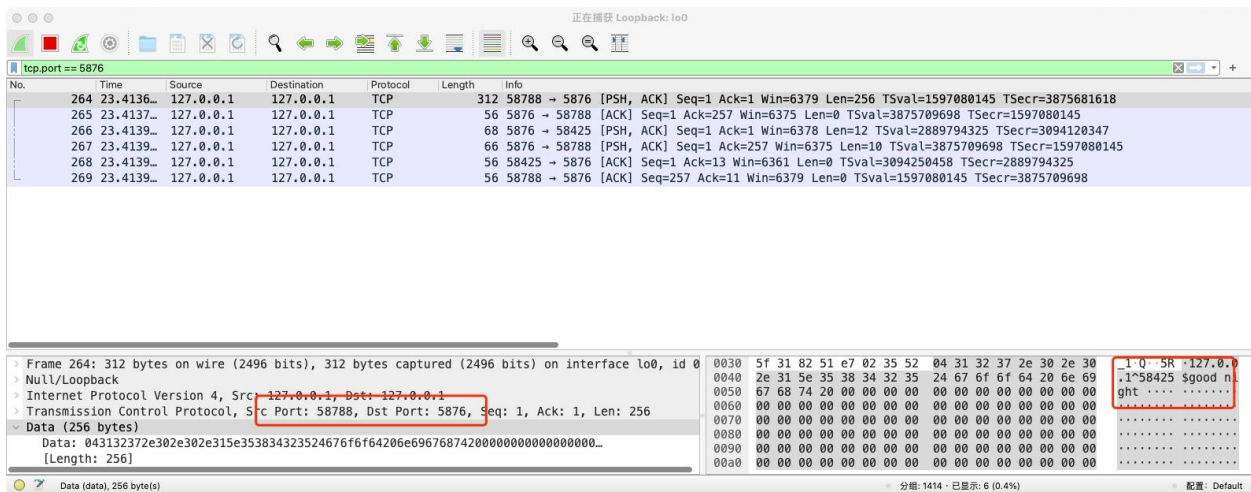
```
> send 127.0.0.1 58425 good night
Success.
0 127.0.0.1 58425
> good night
```

服务器:

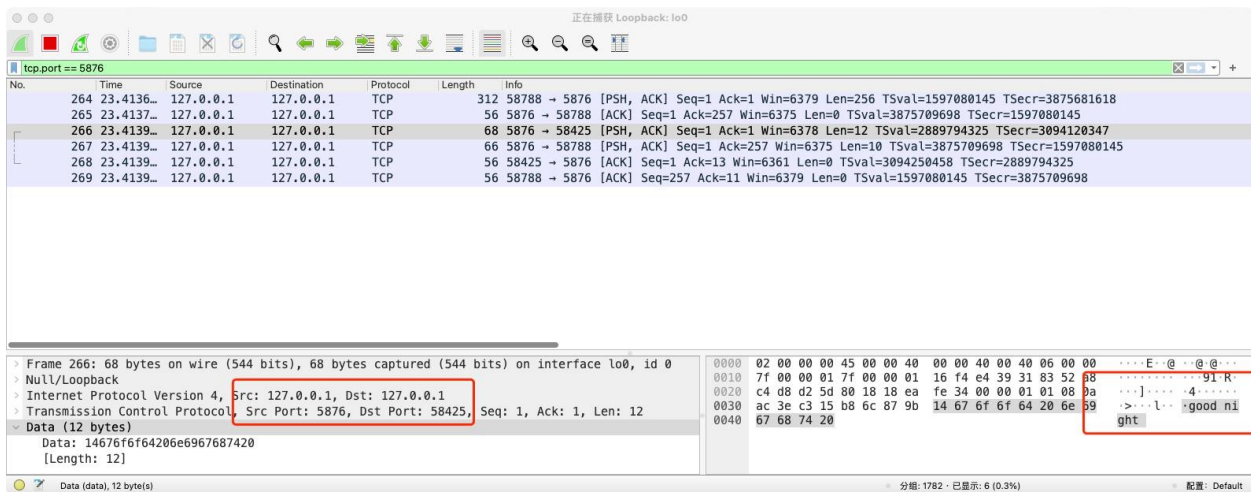
```
127.0.0.1:58788 connected.
6 send message to 127.0.0.1:58425
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送:



接收:



相关的服务器的处理代码片段:

```
for (auto it = clientList.begin(); it != clientList.end(); ++it)
{
    if (it->second.first == ip_addr && it->second.second
== port)
    {
        sock = it->first;
        break;
    }
    std::cout<<cfid<<"          send          message          to
"<<ip_addr<<": "<<port<<'\n';
    buffer_send[0] = 14;
    if (-1 == sock) sprintf(buffer_send + 1, "Fail.\n");
    else
    {
        send_msg(sock, content);
        sprintf(buffer_send + 1, "Success.\n");
    }
}
```

```

    }
    send(cfd, buffer_send, strlen(buffer_send), 0);

```

相关的客户端（发送和接收消息）处理代码片段：

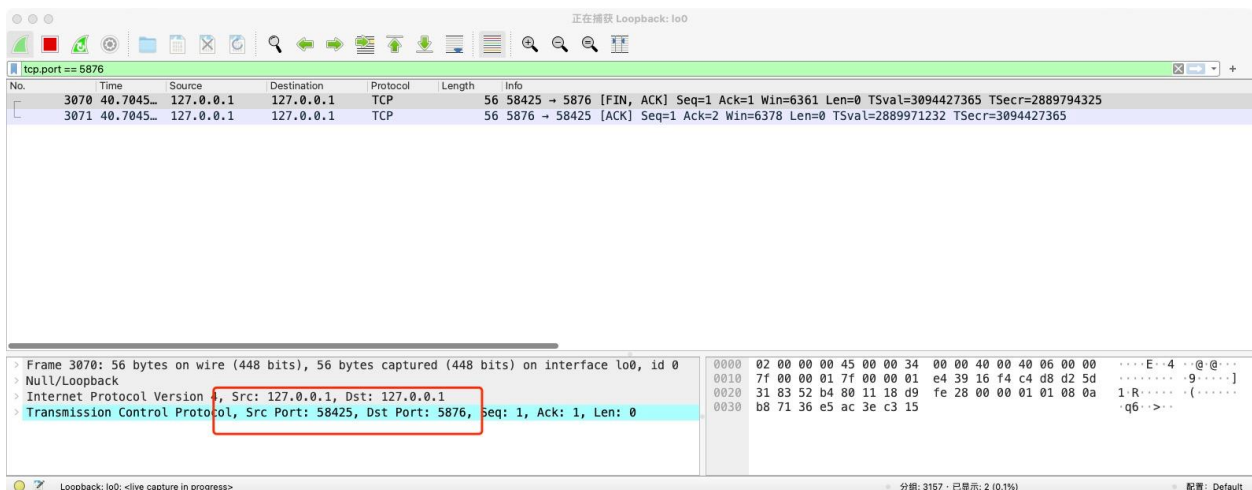
```

/* send */
for (int i = 3; i < words.size(); ++i)
{
    sprintf(buffer + strlen(buffer), "%s",
words[i].c_str());
    if (i != words.size()) sprintf(buffer + strlen(buffer),
" ");
    else sprintf(buffer + strlen(buffer), "\n");
}
send(sockfd, buffer, BUFFER_MAX, 0);

/* receive */
buffer[0] = 5;
sprintf(buffer + strlen(buffer), "receive");
send(sockfd, buffer, BUFFER_MAX, 0);

```

- 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。



可以发现在退出客户端后状态变为了 FIN\_WAIT\_2，也确实抓到了断开连接的包。但是由于我们使用退出终端的方式来模拟断网，这和真实情况应该有所差异，理论分析的结果应当是无法抓到这个包的。



```

[hejiajun@lunar_mbp ~ % netstat -an | grep 5876
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58788      ESTABLISHED
tcp4      0      0 127.0.0.1.58788      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58425      ESTABLISHED
tcp4      0      0 127.0.0.1.58425      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 *.5876              *.*                  LISTEN
[hejiajun@lunar_mbp ~ % netstat -an | grep 5876
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58788      ESTABLISHED
tcp4      0      0 127.0.0.1.58788      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58425      CLOSE_WAIT
tcp4      0      0 127.0.0.1.58425      127.0.0.1.5876      FIN_WAIT_2
tcp4      0      0 *.5876              *.*                  LISTEN

```

十分钟之后，连接消失。

```

[hejiajun@lunar_mbp ~ % netstat -an | grep 5876
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58788      ESTABLISHED
tcp4      0      0 127.0.0.1.58788      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 *.5876              *.*                  LISTEN

```

- 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？

```

[hejiajun@lunar_mbp ~ % netstat -an | grep 5876
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58788      ESTABLISHED
tcp4      0      0 127.0.0.1.58788      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58425      ESTABLISHED
tcp4      0      0 127.0.0.1.58425      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 *.5876              *.*                  LISTEN
[hejiajun@lunar_mbp ~ % netstat -an | grep 5876
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58788      ESTABLISHED
tcp4      0      0 127.0.0.1.58788      127.0.0.1.5876      ESTABLISHED
tcp4      0      0 127.0.0.1.5876      127.0.0.1.58425      CLOSE_WAIT
tcp4      0      0 127.0.0.1.58425      127.0.0.1.5876      FIN_WAIT_2
tcp4      0      0 *.5876              *.*                  LISTEN

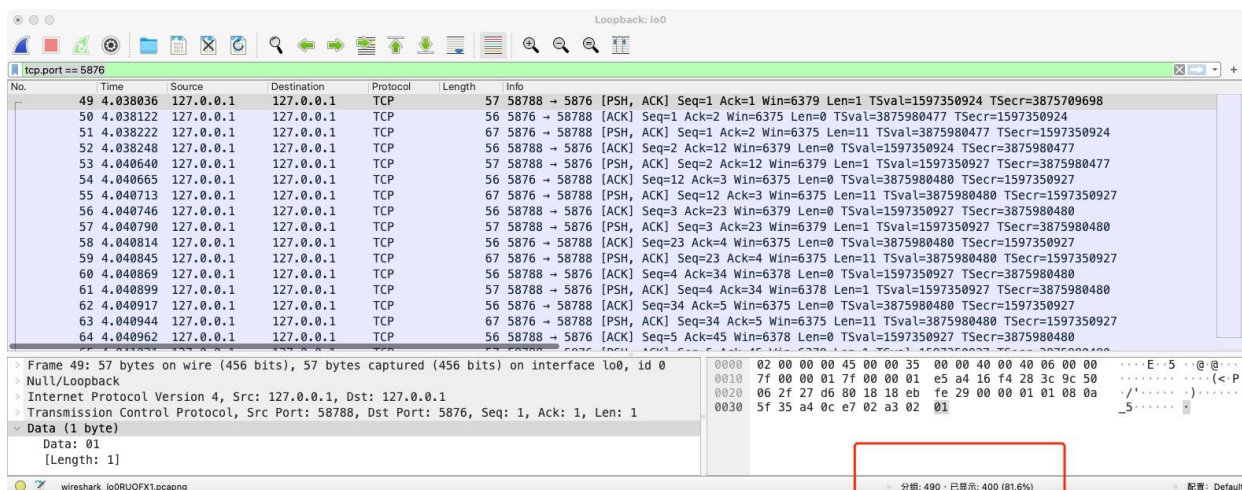
```

连接还在，但是无法发送消息。

- 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。

```
src — client — 80x24

78 times: Server time: Wed Dec 28 19:54:28 2022
79 times: Server time: Wed Dec 28 19:54:28 2022
80 times: Server time: Wed Dec 28 19:54:28 2022
81 times: Server time: Wed Dec 28 19:54:28 2022
82 times: Server time: Wed Dec 28 19:54:28 2022
83 times: Server time: Wed Dec 28 19:54:28 2022
84 times: Server time: Wed Dec 28 19:54:28 2022
85 times: Server time: Wed Dec 28 19:54:28 2022
86 times: Server time: Wed Dec 28 19:54:28 2022
87 times: Server time: Wed Dec 28 19:54:28 2022
88 times: Server time: Wed Dec 28 19:54:28 2022
89 times: Server time: Wed Dec 28 19:54:28 2022
90 times: Server time: Wed Dec 28 19:54:28 2022
91 times: Server time: Wed Dec 28 19:54:28 2022
92 times: Server time: Wed Dec 28 19:54:28 2022
93 times: Server time: Wed Dec 28 19:54:28 2022
94 times: Server time: Wed Dec 28 19:54:28 2022
95 times: Server time: Wed Dec 28 19:54:28 2022
96 times: Server time: Wed Dec 28 19:54:28 2022
97 times: Server time: Wed Dec 28 19:54:28 2022
98 times: Server time: Wed Dec 28 19:54:28 2022
99 times: Server time: Wed Dec 28 19:54:28 2022
100 times: Server time: Wed Dec 28 19:54:28 2022
> █
```



观察右下角可以发现正好 400 个。

- 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



## 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？
  1. 不需要
  2. 源端口的产生是在调用 connect 的时候，socket 为客户自动分配了一个端口
  3. 不是保持不变的
  
- 假设在服务端调用 listen 和调用 accept 之间设了一个调试断点，暂停在此断点时，此时客户端调用 connect 后是否马上能连接成功？

不能。
  
- 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？

不完全一致，大数据测试时产生了偏差。
  
- 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？

通过句柄进行判断。
  
- 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）
  1. 断开连接后 TCP 连接状态为 FIN\_WAIT2
  2. 保持了大约 1 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?

1. 不会发生变化
2. 用一个定时器隔一段时间对所有 Client 检测一次, 看对应的 Ping 时间是否超时, 如果是则直接关闭和释放资源

## 七、 讨论、心得

临近期末考才开始做实验, 时间压力和精神压力都挺大的, 又赶上了新冠大规模爆发的窗口期。只能说, 人总是很难长记性, 上学期一周速通数据库 lab 的事情又一次发生在计网上了。