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：	朱理真
学　　院：	计算机学院
系：	计算机系
专　　业：	计算机
学　　号：	3190101094
指导教师：	张泉方

2022 年　1　月　4　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：无　实验地点：计算机网络实验室

一、 实验目的

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

二、 实验内容

根据自定义的协议规范，使用 Socket 编程接口编写基本的网络应用软件。

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

三、 主要仪器设备

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

四、 操作方法与实验步骤

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

- ✧ 调用 send()，发送一个 hello 消息给客户端（可选）
- ✧ 循环调用 receive()，如果收到了一个完整的请求数据包，根据请求类型做相应的动作：
  1. 请求类型为获取时间：调用 time()获取本地时间，然后将时间数据组装进响应数据包，调用 send()发给客户端
  2. 请求类型为获取名字：将服务器的名字组装进响应数据包，调用 send()发给客户端
  3. 请求类型为获取客户端列表：读取客户端列表数据，将编号、IP 地址、端口等数据组装进响应数据包，调用 send()发给客户端
  4. 请求类型为发送消息：根据编号读取客户端列表数据，如果编号不存在，将错误代码和出错描述信息组装进响应数据包，调用 send()发回源客户端；如果编号存在并且状态是已连接，则将要转发的消息组装进指示数据包。调用 send()发给接收客户端（使用接收客户端的 socket 句柄），发送成功后组装转发成功的响应数据包，调用 send()发回源客户端。

d) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 Socket，主程序退出。

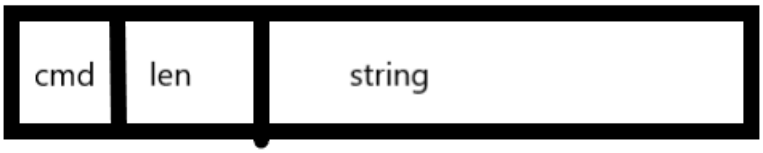
- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

五、 实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

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

- 描述请求数据包的格式（画图说明），请求类型的定义



（string part is optional）

cmd（int）：请求类型（int）

cmd = GET_TIME(7) 获取时间

= GET_NAME(10) 获取服务器名字

= GET_CLIENTS(9) 获取客户端列表

= DATA(1) 发送消息

= END(2) 发送结束连接请求

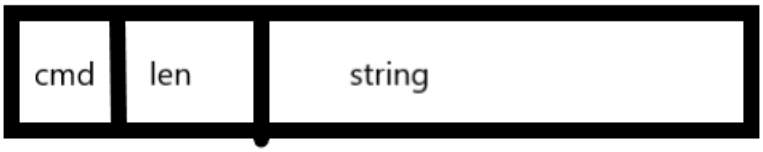
len（int）: string 的长度，若大于 0，消息体后会附加 string 内容

string（String，发送时为 char 数组）：字符串消息

```
enum class MessageType{
    START,DATA,END,HEART,
    ECHO,PRINT,BROADCAST,
    GET_TIME,GET_LIST,GET_CLIENTS,GET_NAME
};
class Message{
public:
    // enum typeCMD{START,DATA,END,HEART};
    int cmd;
    int len=0;
    static const int MAX_BUF_CNT=1024;
    std::string data;
    boolean readMessage(SOCKET s);
    static void recv_n_byte(SOCKET s,char* buf,int n_byte);
    void sendMessage(SOCKET s);
    Message();
    Message(int cmd);
    Message(const std::string& s,int cmd=(int)MessageType::DATA);
    void setData(std::string&s);
};
```

- 描述响应数据包的格式（画图说明），响应类型的定义

格式同上



cmd（int）：响应类型（int），与请求类型相对应

cmd = GET_TIME(7) 获取时间

= GET_NAME(10) 获取服务器名字

= GET_CLIENTS(9) 获取客户端列表

= DATA(1) 发送消息

= END(2) 发送结束连接请求

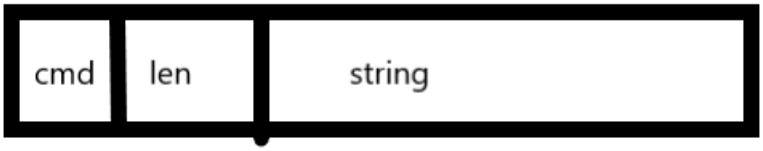
实际上，cmd 字段对响应消息无意义，client 只是简单返回的打印消息

len（int）: string 的长度

string（String，发送时为 char 数组）：字符串消息

- 描述指示数据包的格式（画图说明），指示类型的定义

  格式同上



  cmd（int）：指示类型（int）

  DATA(1) 发送消息

  实际上，cmd 字段对响应消息无意义，client 只是简单返回的打印消息

  len（int）: string 的长度

  string（String，发送时为 char 数组）：字符串消息

- 客户端初始运行后显示的菜单选项

```
connected...
menu:
1: show menu
2: close client
3: get time
4: get server name
5: get clients list
6: send message
>
```

- 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(running){
    std::cout<<"> ";
    cmd=7;
    std::cin.getline(buf,sizeof(buf));
    cmd=atoi(buf);
    switch (cmd)
    {
    case 1:
        std::cout<<help;
        break;
```

```
        case 2:
            closesocket(cs.getSocket());
            running=false;
            break;
        case 3:
            m.cmd=(int)MessageType::GET_TIME;
            m.sendMessage(cs.getSocket());
            break;
        case 4:
            m.cmd=(int)MessageType::GET_NAME;
            m.sendMessage(cs.getSocket());
            break;
        case 5:
            m.cmd=(int)MessageType::GET_CLIENTS;
            m.sendMessage(cs.getSocket());
            break;
        case 6:
            m.cmd=(int)MessageType::DATA;
            std::cout<<"id> ";
            std::cin.getline(buf,sizeof(buf));
            if(atoi(buf)<=0) {
                std::cout<<"illegal"<<std::endl;
            }
            m.data.assign(buf);
            std::cout<<"content> ";
            std::cin.getline(buf,sizeof(buf));
            m.data.append(buf);
            m.sendMessage(cs.getSocket());
            break;
        case 7:
            closesocket(cs.getSocket());
            running=false;
            break;
        case 8:
            m.cmd=(int)MessageType::END;
            m.sendMessage(cs.getSocket());
            break;
        default:
            break;
        }
    }
```

根据不同的命令发送不同的请求

- 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）

```
// std::cout<<"start task"<<std::endl;
while(running){
    ClientSocket wSocket = freeLst->pop();
    // std::cout<<"new socket allocated"<<std::endl;
    while(socket_task_running){
        Message msg;
        if(msg.readMessage(wSocket.getSocket())){
            for(auto& l:lstListener){
                l.get().processMessage(msg,wSocket);
            }
        }
        else{
            for(auto& l:lstListener){
                l.get().stop(wSocket);
                stop_socket_task();
                std::cout<<"socket closed"<<std::endl;
            }
        }
    }
}
```

- 服务器初始运行后显示的界面

```
initialized
new client
```

- 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(running){
    SOCKET ClntSock=accept(servSocket,(SOCKADDR*)&clntAddr,&clntAddrSize);
    if(ClntSock==-1){
        throw std::runtime_error("accept failure");
    }

    addSocket(ClientSocket(ClntSock,clntAddr));
    std::cout<<"new client"<<std::endl;
}
```

- 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）

  读线程：

  （同客户端，但传入的监听者不同）

```
while(running){
    ClientSocket wSocket = freeLst->pop();
    // std::cout<<"new socket allocated"<<std::endl;
    while(socket_task_running){
        Message msg;
        if(msg.readMessage(wSocket.getSocket())){
            for(auto& l:lstListener){
                l.get().processMessage(msg,wSocket);
            }
        }
        else{
            for(auto& l:lstListener){
                l.get().stop(wSocket);
                stop_socket_task();
                std::cout<<"socket closed"<<std::endl;
            }
        }
    }
}
```

  写线程：

```
void SendTask::run(){
    while(running){
        auto p = mq->pop();
        if(!running) break;
        p.first.sendMessage(p.second.getSocket());
    }
}
```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

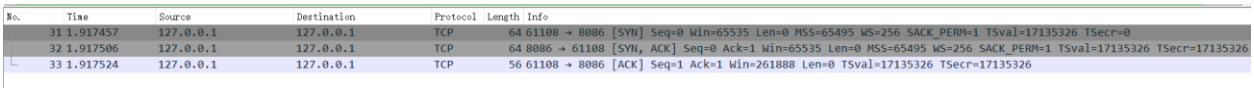
  客户端

```
connected...
menu:
1: show menu
2: close client
3: get time
4: get server name
5: get clients list
6: send message
```

服务端

```
PS G:\jsj\network_lab7\bin> ./s*
initialized
new client
```

Wireshark 抓取的数据包截图：



No.	Time	Source	Destination	Protocol	Length	Info
31	1.917457	127.0.0.1	127.0.0.1	TCP	64	61108 → 8086 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1 TSval=17135326 TSecr=0
32	1.917506	127.0.0.1	127.0.0.1	TCP	64	8086 → 61108 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1 TSval=17135326 TSecr=17135326
33	1.917524	127.0.0.1	127.0.0.1	TCP	56	61108 → 8086 [ACK] Seq=1 Ack=1 Win=261888 Len=0 TSval=17135326 TSecr=17135326

三次握手数据

包 1

```
> Frame 31: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61108, Dst Port: 8086, Seq: 0, Len: 0
    Source Port: 61108
    Destination Port: 8086
    [Stream index: 7]
    [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 2867965555
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  > Flags: 0x002 (SYN)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0x7045 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (20 bytes), Maximum segment size, No-Operation (NOP), Window scale, SACK permitted, Timestamps
  > [Timestamps]
```

包 2

```
> Frame 32: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 8086, Dst Port: 61108, Seq: 0, Ack: 1, Len: 0
    Source Port: 8086
    Destination Port: 61108
    [Stream index: 7]
    [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 4223114889
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2867965556
    1010 .... = Header Length: 40 bytes (10)
  > Flags: 0x012 (SYN, ACK)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0x5e0f [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (20 bytes), Maximum segment size, No-Operation (NOP), Window scale, SACK permitted, Timestamps
  > [Timestamps]
  v [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 31]
      [The RTT to ACK the segment was: 0.000049000 seconds]
      [iRTT: 0.000067000 seconds]
```

包 3

```
> Frame 33: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61108, Dst Port: 8086, Seq: 1, Ack: 1, Len: 0
    Source Port: 61108
    Destination Port: 8086
    [Stream index: 7]
    [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 2867965556
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 4223114890
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x010 (ACK)
    Window: 1023
    [Calculated window size: 261888]
    [Window size scaling factor: 256]
    Checksum: 0x8301 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  > [Timestamps]
  v [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 32]
      [The RTT to ACK the segment was: 0.000018000 seconds]
      [iRTT: 0.000067000 seconds]
```

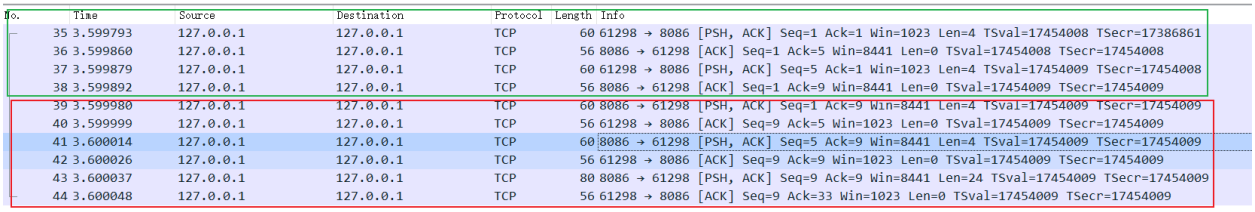
- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

  服务端无更新

  客户端：

```
> 3
> request time: 1641302084
```

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）：



No.	Time	Source	Destination	Protocol	Length	Info
35	3.599793	127.0.0.1	127.0.0.1	TCP	60	61298 → 8086 [PSH, ACK] Seq=1 Ack=1 Win=1023 Len=4 TSval=17454008 TSecr=17386861
36	3.599860	127.0.0.1	127.0.0.1	TCP	56	8086 → 61298 [ACK] Seq=1 Ack=5 Win=8441 Len=0 TSval=17454008 TSecr=17454008
37	3.599879	127.0.0.1	127.0.0.1	TCP	60	61298 → 8086 [PSH, ACK] Seq=5 Ack=1 Win=1023 Len=4 TSval=17454009 TSecr=17454008
38	3.599892	127.0.0.1	127.0.0.1	TCP	56	8086 → 61298 [ACK] Seq=1 Ack=9 Win=8441 Len=0 TSval=17454009 TSecr=17454009
39	3.599980	127.0.0.1	127.0.0.1	TCP	60	8086 → 61298 [PSH, ACK] Seq=1 Ack=9 Win=8441 Len=4 TSval=17454009 TSecr=17454009
40	3.599999	127.0.0.1	127.0.0.1	TCP	56	61298 → 8086 [ACK] Seq=9 Ack=5 Win=1023 Len=0 TSval=17454009 TSecr=17454009
41	3.600014	127.0.0.1	127.0.0.1	TCP	60	8086 → 61298 [PSH, ACK] Seq=5 Ack=9 Win=8441 Len=4 TSval=17454009 TSecr=17454009
42	3.600026	127.0.0.1	127.0.0.1	TCP	56	61298 → 8086 [ACK] Seq=9 Ack=9 Win=1023 Len=0 TSval=17454009 TSecr=17454009
43	3.600037	127.0.0.1	127.0.0.1	TCP	80	8086 → 61298 [PSH, ACK] Seq=9 Ack=9 Win=8441 Len=24 TSval=17454009 TSecr=17454009
44	3.600048	127.0.0.1	127.0.0.1	TCP	56	61298 → 8086 [ACK] Seq=9 Ack=33 Win=1023 Len=0 TSval=17454009 TSecr=17454009

请求发送了两个包（cmd,len）

响应发送了三个包（cmd,len,string）

请求：

```
    [TCP Segment Len: 4]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 4029854888
    [Next Sequence Number: 5    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2485883382
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x018 (PSH, ACK)
    Window: 1023
    [Calculated window size: 1023]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xf070 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  > [Timestamps]
  v [SEQ/ACK analysis]
      [Bytes in flight: 4]
      [Bytes sent since last PSH flag: 4]
    TCP payload (4 bytes)
Data (4 bytes)
    Data: 07000000
    [Length: 4]
```

cmd=7

```
    [TCP Segment Len: 4]
    Sequence Number: 5    (relative sequence number)
    Sequence Number (raw): 4029854892
    [Next Sequence Number: 9    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2485883382
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x018 (PSH, ACK)
    Window: 1023
    [Calculated window size: 1023]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xf11f [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  > [Timestamps]
  v [SEQ/ACK analysis]
      [Bytes in flight: 4]
      [Bytes sent since last PSH flag: 4]
    TCP payload (4 bytes)
v Data (4 bytes)
    Data: 00000000
    [Length: 4]
```

len=0

响应：

```
   [TCP Segment Len: 4]
   Sequence Number: 5    (relative sequence number)
   Sequence Number (raw): 2485883386
   [Next Sequence Number: 9    (relative sequence number)]
   Acknowledgment Number: 9    (relative ack number)
   Acknowledgment number (raw): 4029854896
   1000 .... = Header Length: 32 bytes (8)
 > Flags: 0x018 (PSH, ACK)
   Window: 8441
   [Calculated window size: 8441]
   [Window size scaling factor: -1 (unknown)]
   Checksum: 0xbc1c [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
 > [Timestamps]
 v [SEQ/ACK analysis]
     [Bytes in flight: 4]
     [Bytes sent since last PSH flag: 4]
   TCP payload (4 bytes)
Data (4 bytes)
   Data: 18000000
   [Length: 4]
```

len=0x18=24

```
[TCP Segment Len: 24]
Sequence Number: 9    (relative sequence number)
Sequence Number (raw): 2485883390
[Next Sequence Number: 33    (relative sequence number)]
Acknowledgment Number: 9    (relative ack number)
Acknowledgment number (raw): 4029854896
1000 .... = Header Length: 32 bytes (8)
> Flags: 0x018 (PSH, ACK)
Window: 8441
[Calculated window size: 8441]
[Window size scaling factor: -1 (unknown)]
Checksum: 0xf7aa [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
> Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
> [Timestamps]
v [SEQ/ACK analysis]
    [Bytes in flight: 24]
    [Bytes sent since last PSH flag: 24]
TCP payload (24 bytes)
Data (24 bytes)
Data: 726571756573742074696d653a203136343133303230383 4
[Length: 24]
```

```
20  f0 32 b4 b0 80 18 20 f9  f7 aa 00 00 01 01 08 0a   ·2···· · ········
30  01 0a 53 b9 01 0a 53 b9  72 65 71 75 65 73 74 20   ··S···S· request
40  74 69 6d 65 3a 20 31 36  34 31 33 30 32 30 38 34   time: 16 41302084
```

string = “request time: 1641302084”

- 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。

  服务端无新内容

  客户端：

```
4
> request name: LAPTOP-S6B6C46G
```

  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：

```
v Data (4 bytes)
    Data: 0a000000
    [Length: 4]
```

(cmd=10)

```
Data (4 bytes)
    Data: 00000000
    [Length: 4]
```
(len=0)

响应：

```
Data (4 bytes)
    Data: 0a000000
    [Length: 4]
```

cmd=10

```
Data (4 bytes)
    Data: 1d000000
    [Length: 4]
```

len=0x1d=29

```
Data (29 bytes)
    Data: 726571756573742 06e616d653a204c4150544f502d5336423643343647
    [Length: 29]
```

```
0030  01 16 a2 50 01 16 a2 50  72 65 71 75 65 73 74 20   ···P···P request
0040  6e 61 6d 65 3a 20 4c 41  50 54 4f 50 2d 53 36 42   name: LA PTOP-S6B
0050  36 43 34 36 47                                     6C46G
```

string 如图

相关的服务器的处理代码片段：

```
void ServerFuncs::name(Server&s,Message&m,ClientSocket&c){
    std::ostringstream os;
    char buf[80];
    gethostname(buf,sizeof(buf));
    os<<"request name: "<<buf;
    Message msg(os.str(),(int)MessageType::GET_NAME);
    s.msgQ.push(msg,c);
}
```

- 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：

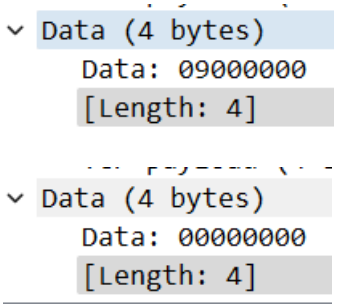
```
5
> Client Infomation| id: 264 IP: 127.0.0.1 port: 61298
```

服务端：

```
Client Infomation| id: 264 IP: 127.0.0.1 port: 61298
```

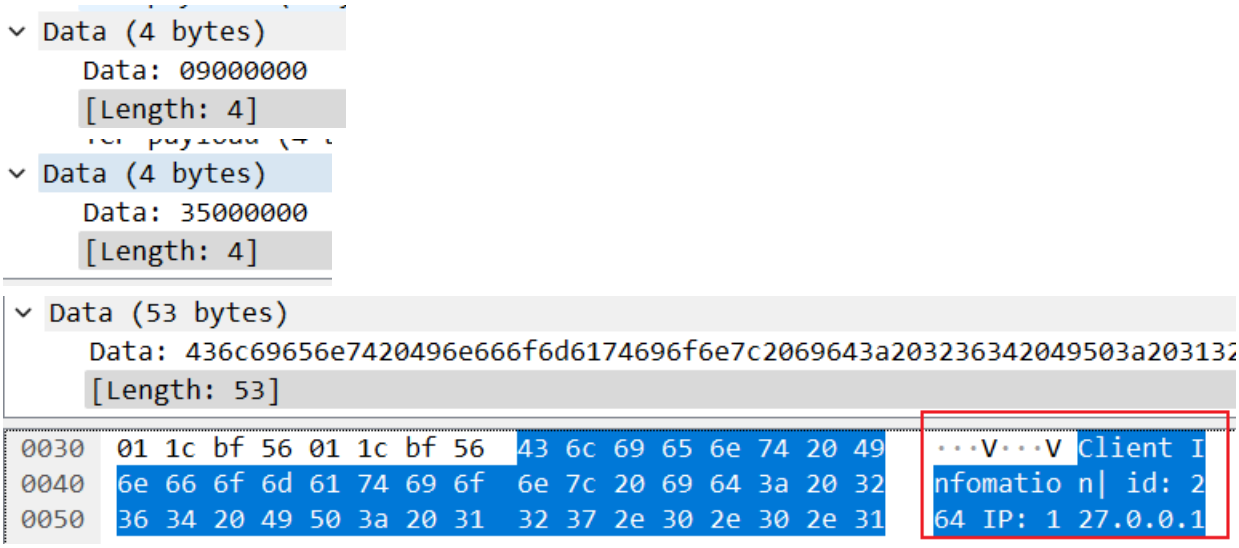
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：

--请求—



分别代表 cmd，len

--响应—



分别代表 cmd，len，string

相关的服务器的处理代码片段：

```cpp
void ServerFuncs::list(Server&s,Message&m,ClientSocket&c){
    // std::cout<<"request list:"<<std::endl;
    std::ostringstream os;
    auto func = [&os](ClientSocket& cs){
        os<<"Client Infomation| id: "<<cs.s<<" IP: "<<cs.getIP()<<" port: "<<cs.getPort()<<"\n";
    };
    s.lstClient.consume(func);
    std::cout<<os.str()<<std::endl;
    Message msg(os.str(),(int)MessageType::GET_CLIENTS);
    s.msgQ.push(msg,c);
}
```

- 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：

```
6
id> 264
content> hello world!
```

服务器：

无新内容

接收消息的客户端：

```
> send message:
Client Infomation| id: 264 IP: 127.0.0.1 port: 61298
send info:
hello world!
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

--发送：

```
Data (4 bytes)
    Data: 01000000
    [Length: 4]
```

```
Data (4 bytes)
    Data: 0f000000
    [Length: 4]
```

```
Data (15 bytes)
    Data: 32363468656c6c6f20776f726c6421
    [Length: 15]
```

```
0020  94 2b 92 78 80 18 04 fe  b5 af 00 00 01 01 08 0a   ·+·x···· ········
0030  01 20 c5 a2 01 20 c5 a2  32 36 34 68 65 6c 6c 6f   · ··· ·· 264hello
0040  20 77 6f 72 6c 64 21                               world!
```

分别代表 cmd，len，string

--接收：

```
Data (4 bytes)
    Data: 01000000
    [Length: 4]
```

```
Data (4 bytes)
    Data: 5c000000
    [Length: 4]
```

```
Data (92 bytes)
    Data: 73656e64206d6573736167653a200a436c69656e7420496e666f6d6174696f6e7c2
    [Length: 92]
```

```
0020  f0 32 b4 d7 80 18 20 f8  ce a4 00 00 01 01 08 0a   ·2···· · ········
0030  01 20 c5 a3 01 20 c5 a3  73 65 6e 64 20 6d 65 73   · ··· ·· send mes
0040  73 61 67 65 3a 20 0a 43  6c 69 65 6e 74 20 49 6e   sage: ·C lient In
0050  66 6f 6d 61 74 69 6f 6e  7c 20 69 64 3a 20 32 36   fomation | id: 26
0060  34 20 49 50 3a 20 31 32  37 2e 30 2e 30 2e 31 20   4 IP: 12 7.0.0.1
0070  70 6f 72 74 3a 20 36 31  32 39 38 0a 73 65 6e 64   port: 61 298·send
0080  20 69 6e 66 6f 3a 20 0a  68 65 6c 6c 6f 20 77 6f    info: · hello wo
0090  72 6c 64 21                                       rld!
```

分别代表 cmd，len，string

相关的服务器的处理代码片段：

```cpp
void ServerFuncs::send(Server&s,Message&m,ClientSocket&c){
    // std::cout<<"request send:"<<std::endl;
    std::istringstream is(m.data);
    std::ostringstream os;
    SOCKET toSend;
    char str[300];
    is>>toSend;
    is.getline(str,sizeof(str));

    os<<"send message: "<<std::endl;
    os<<"Client Infomation| id: "<<c.s<<" IP: "<<c.getIP()<<" port: "<<c.getPort()<<std::endl;
    os<<"send info: "<<std::endl;
    os<<str;

    // std::cout<<toSend<<" "<<os.str()<<std::endl;

    Message msg(os.str(),(int)MessageType::DATA);
    auto func = [&s,&msg,toSend](ClientSocket& cs){
        if(toSend==cs.s){
            s.msgQ.push(msg,cs);
            // std::cout<<"sent"<<std::endl;
        }
    };
    s.lstClient.consume(func);
}
```

相关的客户端（发送和接收消息）处理代码片段：

发送：

```cpp
case 6:
    m.cmd=(int)MessageType::DATA;
    std::cout<<"id> ";
    std::cin.getline(buf,sizeof(buf));
    if(atoi(buf)<=0) {
        std::cout<<"illegal"<<std::endl;
    }
    m.data.assign(buf);
    std::cout<<"content> ";
    std::cin.getline(buf,sizeof(buf));
    m.data.append(buf);
    m.sendMessage(cs.getSocket());
    break;
```

接收：

```
void ReadTask::run(){
    // std::cout<<"start task"<<std::endl;
    while(running){
        ClientSocket wSocket = freeLst->pop();
        // std::cout<<"new socket allocated"<<std::endl;
        while(socket_task_running){
            Message msg;
            if(msg.readMessage(wSocket.getSocket())){
                for(auto& l:lstListener){
                    l.get().processMessage(msg,wSocket);
                }
            }
            else{
                for(auto& l:lstListener){
                    l.get().stop(wSocket);
                    stop_socket_task();
                    std::cout<<"socket closed"<<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。

客户端：

8776	602.302577	10.192.172.162	10.181.243.152	TCP	70	64009 → 12345 [PSH, ACK] Seq=25 Ack=147 Win=131584 Len=4 TSval=21256816 TSecr=30364170
8777	602.398624	10.181.243.152	10.192.172.162	TCP	66	12345 → 64009 [ACK] Seq=147 Ack=29 Win=1049600 Len=0 TSval=30404054 TSecr=21256816
8778	602.398666	10.192.172.162	10.181.243.152	TCP	70	64009 → 12345 [PSH, ACK] Seq=29 Ack=147 Win=131584 Len=4 TSval=21256912 TSecr=30404054
8779	602.461437	10.181.243.152	10.192.172.162	TCP	70	12345 → 64009 [PSH, ACK] Seq=147 Ack=33 Win=1049600 Len=4 TSval=30404117 TSecr=21256912
8780	602.509810	10.192.172.162	10.181.243.152	TCP	66	64009 → 12345 [ACK] Seq=33 Ack=151 Win=131584 Len=0 TSval=21257023 TSecr=30404117
8781	602.567928	10.181.243.152	10.192.172.162	TCP	128	12345 → 64009 [PSH, ACK] Seq=151 Ack=33 Win=1049600 Len=62 TSval=30404224 TSecr=21257023
8782	602.618438	10.192.172.162	10.181.243.152	TCP	66	64009 → 12345 [ACK] Seq=33 Ack=213 Win=131328 Len=0 TSval=21257132 TSecr=30404224

wifi 断开后，没有捕捉到释放的消息

服务端：

没有变化：

No.	Time	Source	Destination	Protocol	Length	Info
18286	237.055239	10.181.243.152	10.192.172.162	TCP	66	9856 → 8086 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
18287	237.055589	10.192.172.162	10.181.243.152	TCP	66	8086 → 9856 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
18288	237.128750	10.181.243.152	10.192.172.162	TCP	56	9856 → 8086 [ACK] Seq=1 Ack=1 Win=131328 Len=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？

  还在。

```
5
> Client Infomation| id: 278 IP: 10.181.234.152 port:64009
```

  发送消息失败，之前异常退出的客户端被服务端关掉了 socket。

```
socket closed
```

- 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。

  开头：

41	3.782305	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=1 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21797850
42	3.782331	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=5 Win=8441 Len=0 TSval=21812528 TSecr=21812528
43	3.782346	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=5 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
44	3.782355	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=9 Win=8441 Len=0 TSval=21812528 TSecr=21812528
45	3.782365	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=9 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
46	3.782375	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=13 Win=8441 Len=0 TSval=21812528 TSecr=21812528
47	3.782386	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=13 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
48	3.782395	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=17 Win=8441 Len=0 TSval=21812528 TSecr=21812528
49	3.782405	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=17 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
50	3.782426	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=21 Win=8441 Len=0 TSval=21812528 TSecr=21812528
51	3.782439	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=21 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
52	3.782449	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=25 Win=8441 Len=0 TSval=21812528 TSecr=21812528
53	3.782459	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=25 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
54	3.782468	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=29 Win=8441 Len=0 TSval=21812528 TSecr=21812528
55	3.782478	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=29 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
56	3.782487	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=33 Win=8441 Len=0 TSval=21812528 TSecr=21812528
57	3.782504	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=33 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
58	3.782518	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=1 Ack=37 Win=8441 Len=0 TSval=21812528 TSecr=21812528
59	3.782526	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=1 Ack=37 Win=8441 Len=4 TSval=21812528 TSecr=21812528
60	3.782535	127.0.0.1	127.0.0.1	TCP	60	60363 → 1094 [PSH, ACK] Seq=37 Ack=1 Win=8441 Len=4 TSval=21812528 TSecr=21812528
61	3.782549	127.0.0.1	127.0.0.1	TCP	56	1094 → 60363 [ACK] Seq=5 Ack=41 Win=8441 Len=0 TSval=21812528 TSecr=21812528
62	3.782563	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=41 Ack=5 Win=8441 Len=0 TSval=21812528 TSecr=21812528

  …

  结尾：

536	3.810245	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3077 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
537	3.810254	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3081 Win=8429 Len=0 TSval=21812556 TSecr=21812556
538	3.810263	127.0.0.1	127.0.0.1	TCP	80	1094 → 60363 [PSH, ACK] Seq=3081 Ack=801 Win=8438 Len=24 TSval=21812556 TSecr=21812556
539	3.810272	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3105 Win=8429 Len=0 TSval=21812556 TSecr=21812556
540	3.810282	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3105 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
541	3.810291	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3109 Win=8429 Len=0 TSval=21812556 TSecr=21812556
542	3.810300	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3109 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
543	3.810309	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3113 Win=8429 Len=0 TSval=21812556 TSecr=21812556
544	3.810318	127.0.0.1	127.0.0.1	TCP	80	1094 → 60363 [PSH, ACK] Seq=3113 Ack=801 Win=8438 Len=24 TSval=21812556 TSecr=21812556
545	3.810326	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3137 Win=8428 Len=0 TSval=21812556 TSecr=21812556
546	3.810336	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3137 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
547	3.810344	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3141 Win=8428 Len=0 TSval=21812556 TSecr=21812556
548	3.810353	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3141 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
549	3.810361	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3145 Win=8428 Len=0 TSval=21812556 TSecr=21812556
550	3.810370	127.0.0.1	127.0.0.1	TCP	80	1094 → 60363 [PSH, ACK] Seq=3145 Ack=801 Win=8438 Len=24 TSval=21812556 TSecr=21812556
551	3.810380	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3169 Win=8428 Len=0 TSval=21812556 TSecr=21812556
552	3.810390	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3169 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
553	3.810399	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3173 Win=8428 Len=0 TSval=21812556 TSecr=21812556
554	3.810408	127.0.0.1	127.0.0.1	TCP	60	1094 → 60363 [PSH, ACK] Seq=3173 Ack=801 Win=8438 Len=4 TSval=21812556 TSecr=21812556
555	3.810417	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3177 Win=8428 Len=0 TSval=21812556 TSecr=21812556
556	3.810426	127.0.0.1	127.0.0.1	TCP	80	1094 → 60363 [PSH, ACK] Seq=3177 Ack=801 Win=8438 Len=24 TSval=21812556 TSecr=21812556
557	3.810435	127.0.0.1	127.0.0.1	TCP	56	60363 → 1094 [ACK] Seq=801 Ack=3201 Win=8428 Len=0 TSval=21812556 TSecr=21812556

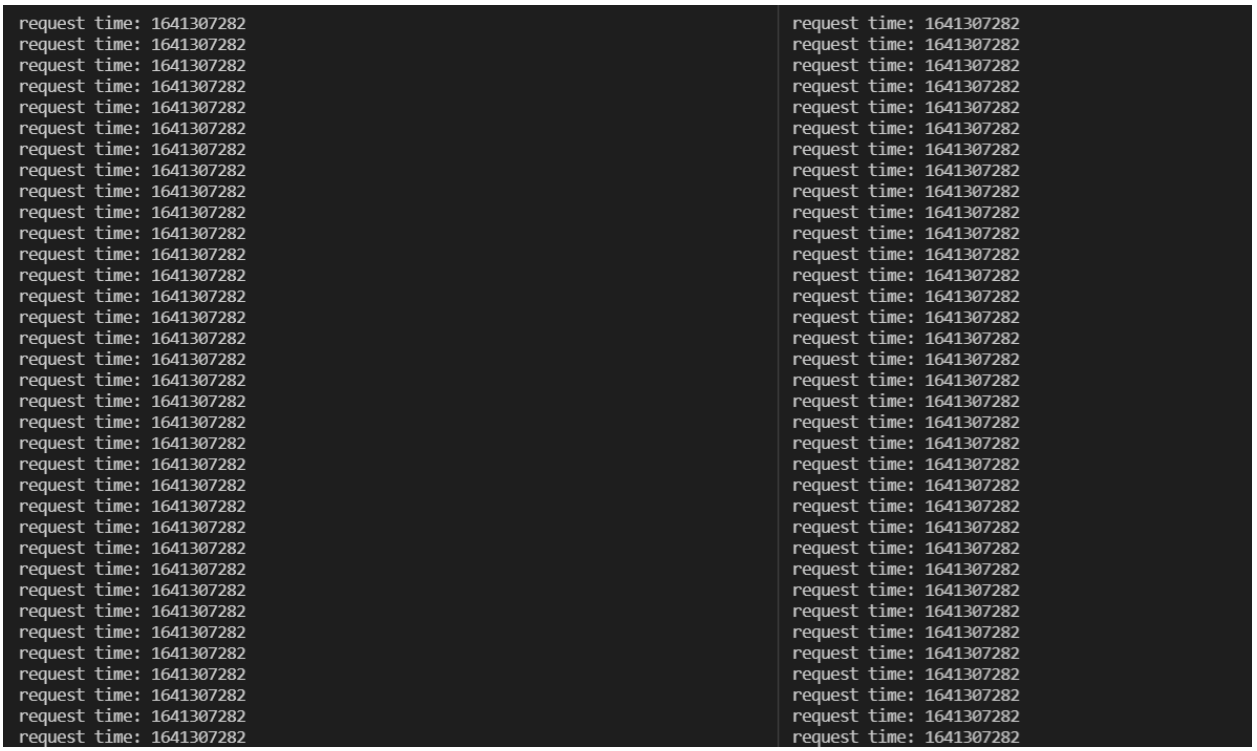
需验证，请看 seq 序号

总共响应包 247 个，而每个响应应该有 3 个包，因此理应有 300 个包

- 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

  服务器无内容

  两个客户端：



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？

  不需要，connect 时自动分配的

  可能会改变

- 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？

  不同数据包是从不同客户端的套接字（包含 ip，port 信息）中取得的，因此可以区分。

- 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）

  主动断开连接后

  FIN-WAIT-1 0

- 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？

  没有变化

  持续发送心跳包，若多次未收到则视为断开

七、 讨论、心得

客户端拔掉网线再连上比较困难，需要联系外部，或者去机房获得更好的网络环境。