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：

学　　院：　计算机学院

系：　计算机系

专　　业：　计算机科学与技术

学　　号：

指导教师：　张泉方

2022 年　11 月　01 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

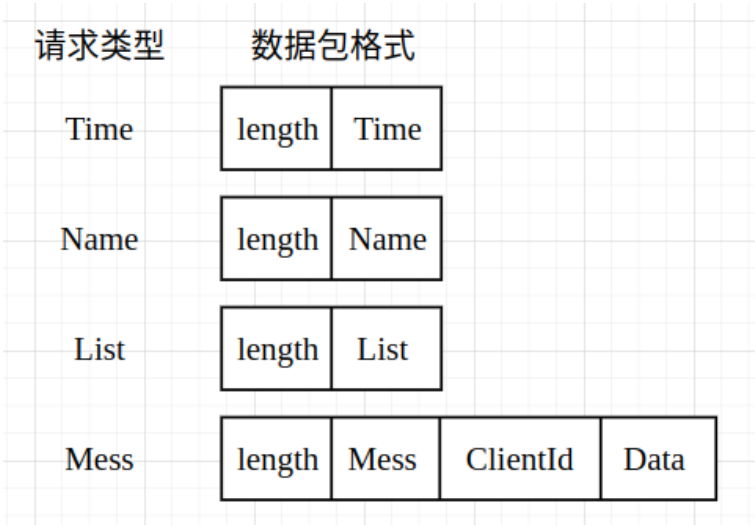
五、 实验数据记录和处理

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

请求数据包格式(第二列表示请求类型)：

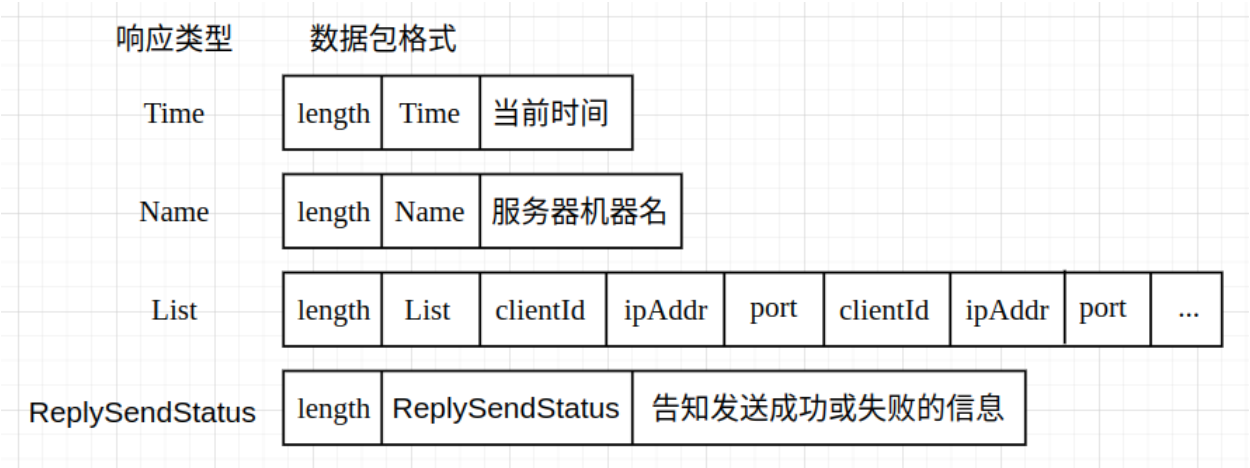


请求类型的定义：

请求类型	定义
Time	请求服务器返回当前时间
Name	请求服务器返回其机器名
List	请求服务器返回当前与其连接的客户端列表
Mess	请求向另一个与服务器相连的客户发送消息

- 描述响应数据包的格式（画图说明），响应类型的定义

响应数据包格式如下：

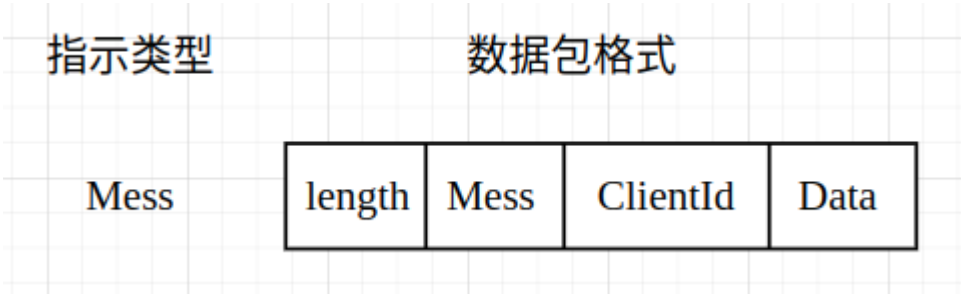


响应类型的定义如下：

响应类型	定义
Time	服务器返回当前时间
Name	服务器返回机器名
List	服务器返回当前与其连接的客户端列表
ReplySendStatus	服务器告知源客户端信息是否发送成功

- 描述指示数据包的格式（画图说明），指示类型的定义

  指示数据包格式如下：



  指示类型定义如下：

指示类型	定义
Mess	服务器将来自源客户端的消息转发给目标客户端

- 客户端初始运行后显示的菜单选项

```
Here is the guide menu
-----------------------------------------------------
| Cmd |                Description                    |
-----------------------------------------------------
|  c  |  Connect to the server                        |
|  d  |  Disconnect from the server                   |
|  t  |  Get the current time from the server         |
|  n  |  Get the machine name of the server           |
|  l  |  List all clients connected to the server     |
|  s  |  Send a message to another specified client   |
|  e  |  Disconnect and exit this client              |
|  m  |  Show this menu again                         |
-----------------------------------------------------
Usage: Type one of the commands at a time
```

  如图，会显示引导菜单，列出了可用的各个命令及对其作用的简要描述。在菜单下方提示使用

方法为每次在命令行输入一个命令字符。

- 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) {
  cin >> cmd;
  if(cmd.empty()) continue;
  if(cmd == "c") {
    if(connectState) {
      printPrompt("Already connected");
      continue;
    }
    sockfd = connectSEV();
    if(sockfd == -1) continue;
    connectState = true;
    receiver = thread(rev, sockfd);
    printPrompt("Connect successfully");
  } else if(cmd == "d") {
    if(!connectState) {
      printPrompt("Not connected yet");
      continue;
    }
    synDisconnect(sockfd);
    receiver.join();
    close(sockfd);
    connectState = false;
    printPrompt("Disconnect successfully");
  } else if(cmd == "t") {
    if(isConnected()) {
      getTime(sockfd);
    }
```

```
  } else if(cmd == "n") {
    if(isConnected()) {
      getName(sockfd);
    }
  } else if(cmd == "l") {
    if(isConnected()) {
      getClientList(sockfd);
    }
  } else if(cmd == "s") {
    if(isConnected()) {
      sendMessage(sockfd);
    }
  } else if(cmd == "e") {
    if(connectState) {
      synDisconnect(sockfd);
      receiver.join();
      close(sockfd);
    }
    printPrompt("Exiting the client");
    return 0;
  } else if(cmd == "m") {
    printMenu();
  } else {
    printPrompt("Illegal command, please try again");
  }
}
```

如图，客户端主线程不断接受用户的命令，然后调用相应的函数进行处理。调用函数前会检测连接状况，未连接服务端时，只能选择连接和退出功能。如果命令非法，会提示用户重新输入。

- 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 rev(int sockfd) {
  while(true) {
    Message msg = Message();
    // errif(msg.revMessage(sockfd) == -1, "server socket disconnected!");
    if(msg.revMessage(sockfd) != Error::Normal) {
      raise(SIGQUIT);
      close(sockfd);
      connectState = false;
      printPrompt("Server disconnected, the connection is down");
      return;
    }

    if(msg.type == Type::Disconnect) break;
    else if(msg.type == Type::Time) {
      cout << "Time: " << msg.data;
    } else if(msg.type == Type::Name) {
      cout << "Server Machine Name: " << msg.data << endl;
    } else if(msg.type == Type::List) {
      cout << "Clients' Information: " << endl;
      printClientsInfo(msg);
    } else if(msg.type == Type::Mess) {
      cout << "Message from client with id[" << msg.clientId << "]: ";
      cout << msg.data << endl;
    } else if(msg.type == Type::ReplySendStatus) {
      cout << msg.data << endl;
    } else if(msg.type == Type::ServerExit) {
      raise(SIGQUIT);
      ackServerExit(sockfd);
      close(sockfd);
      connectState = false;
      printPrompt("Server has exited, the connection is down");
      return;
    }
  }
}
```

该子线程负责接收来自服务端的数据包，如果接收出错，即服务端异常断开连接，则客户端也断开与服务端的连接，释放资源，并结束子线程。如果正常接收到数据包，则根据类型字段做相应的处理，主要是将数据以更友好的方式展现给用户，如添加一些提示信息、合理分行等。如果接收到的数据包类型为 ServerExit，即服务端提示将要退出，此时客户端 ack 此消息，并准备与服务端断开连接，结束子线程。

- 服务器初始运行后显示的界面

```
(base) yyb@yyb-laptop:~/zju/cn22fall$ ./server
[PROMPT] Waiting for new client
[PROMPT] Enter "quit" to exit
```

服务器初始化完成后等待用户连接，可以通过输入 quit 命令退出。

- 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 (!serverExit) {
    // accept new client
    printPrompt("Waiting for new client");
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_len = sizeof(clnt_addr);
    bzero(&clnt_addr, sizeof(clnt_addr));
    int clnt_sockfd;

    setjmp(jmpbuf);
    if(serverExit) {
        close(sevSockfd);
        printPrompt("Server shutdown!");
        return 0;
    }
    clnt_sockfd = accept(sevSockfd, (sockaddr*)&clnt_addr, &clnt_addr_len);
    // errif(clnt_sockfd == -1, "socket accept error");

    ClientInfo cInfo;
    cInfo.clientId = nextClientId++;
    bzero(cInfo.ipAddr, IPV4_LEN);
    strcpy(cInfo.ipAddr, inet_ntoa(clnt_addr.sin_addr));
    cInfo.port = ntohs(clnt_addr.sin_port);
    cInfo.sockfd = clnt_sockfd;
    cList.addClientInfo(cInfo); // need mutex

    cout << "New client: [ClientId]" << cInfo.clientId << " [IPAddress]" << cInfo.ipAddr << " [Port]" << cInfo.port << endl;
    childThreads.push_back(thread(handleClient, cInfo));
}
```

服务器等待新客户的连接，accept 接受新客户后为其分配新的 socket fd。然后将新客户的信息添加到客户列表中，并新建子线程与该客户通信，主线程进入下一轮循环，继续等待新客户的连接。

- 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）

```
void handleClient(ClientInfo cInfo) {
    int clnt_sockfd = cInfo.sockfd;
    Message revMsg = Message();

    while(!serverExit) {
        if(revMsg.revMessage(clnt_sockfd) != Error::Normal) {
            break;
        }

        if(revMsg.type == Type::Time) {
            sendTime(clnt_sockfd);
        } else if(revMsg.type == Type::Name) {
            sendName(clnt_sockfd);
        } else if(revMsg.type == Type::List) {
            sendList(clnt_sockfd);
        } else if(revMsg.type == Type::Mess) {
            if(transMessage(cInfo.clientId, revMsg) == 0) {
                replySendStatus(clnt_sockfd, "Message sent successfully");
            } else {
                replySendStatus(clnt_sockfd, "Message failed to send, client could not be found");
            }
        } else if(revMsg.type == Type::Disconnect) {
            ackDisconnect(clnt_sockfd);
            break;
        } else if(revMsg.type == Type::ServerExit) {
        } else {
            printPrompt("Unknown message type");
        }
    }

    close(clnt_sockfd);
    cList.rmClientInfo(cInfo.clientId);
    cout << PROMPT << "Client with id[" << cInfo.clientId << "] " << "disconnected" << endl;
    printPrompt("Child thread exited");
}
```

子线程接收来自客户端的数据包，如果接收出错，即客户端异常断开连接，则退出循环，移除客户信息，结束子线程。如果正常接收到数据包，则根据类型字段做相应的处理，主要是将客户需要的信息打包送回。若请求类型是发送消息，则转发给别的客户，并告知原客户消息是否成功发送。若类型为 Disconnect，即客户端通知要断开连接，此时服务端 ack 该消息，并准备断开连接，结束子线程。

- 客户端选择连接功能时，客户端和服务端显示内容截图。

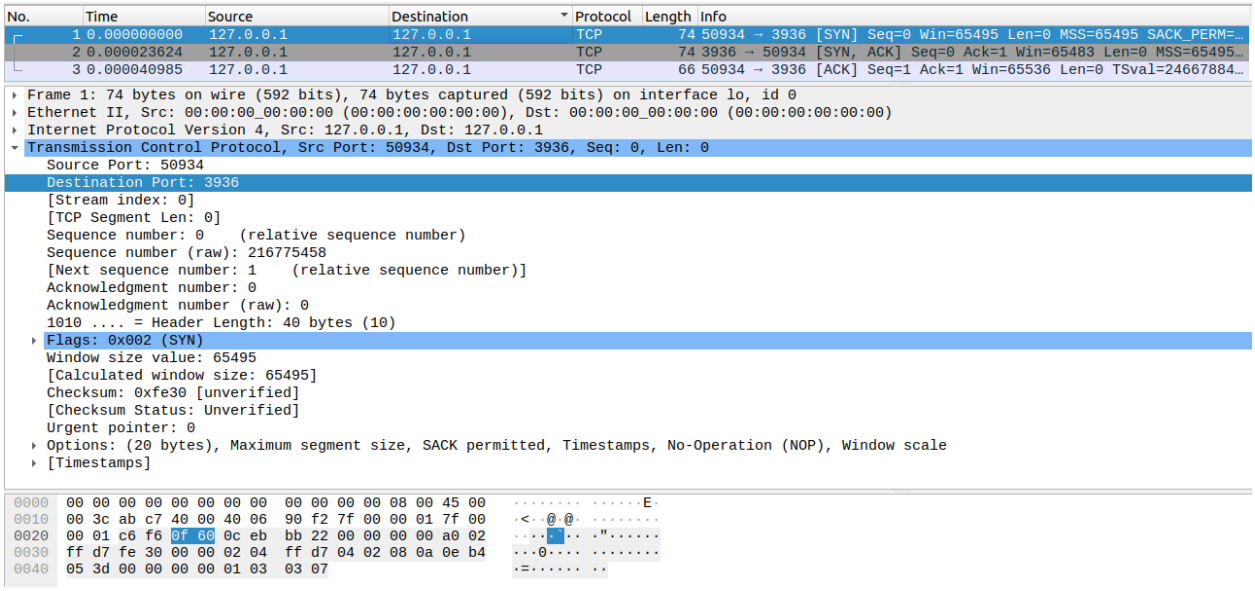
客户端：

```
c
[PROMPT] Please enter the IP of the server
127.0.0.1
[PROMPT] Please enter the Port of the server
3936
[PROMPT] Connect successfully
```

服务端：

```
New client: [ClientId]0 [IPAddress]127.0.0.1 [Port]37126
[PROMPT] Waiting for new client
```

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：

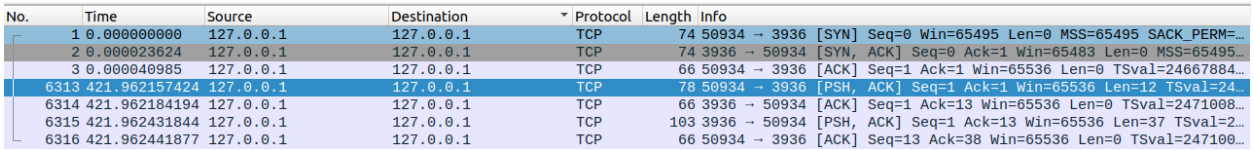
```
t
Time: Fri Nov 11 19:38:20 2022
```

服务端：

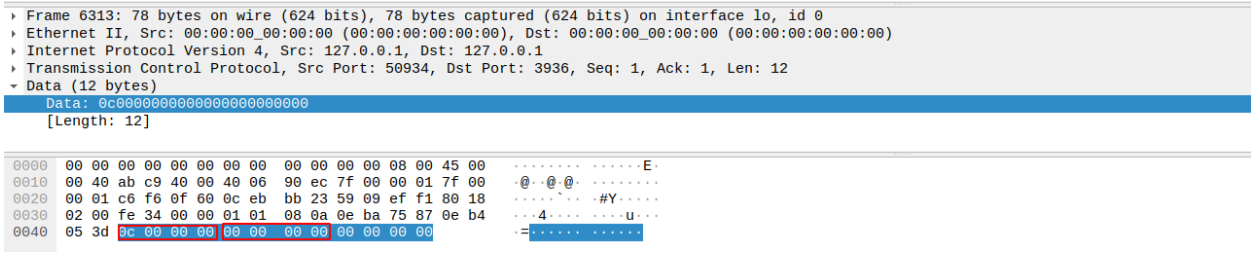
```
[PROMPT] Time has been sent to client with id[0]
```

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：



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	50934 → 3936 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=…
2	0.000023624	127.0.0.1	127.0.0.1	TCP	74	3936 → 50934 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495…
3	0.000040985	127.0.0.1	127.0.0.1	TCP	66	50934 → 3936 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=24667884…
6313	421.962157424	127.0.0.1	127.0.0.1	TCP	78	50934 → 3936 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=24…
6314	421.962184194	127.0.0.1	127.0.0.1	TCP	66	3936 → 50934 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=2471008…
6315	421.962431844	127.0.0.1	127.0.0.1	TCP	103	3936 → 50934 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=37 TSval=2…
6316	421.962441877	127.0.0.1	127.0.0.1	TCP	66	50934 → 3936 [ACK] Seq=13 Ack=38 Win=65536 Len=0 TSval=247100…

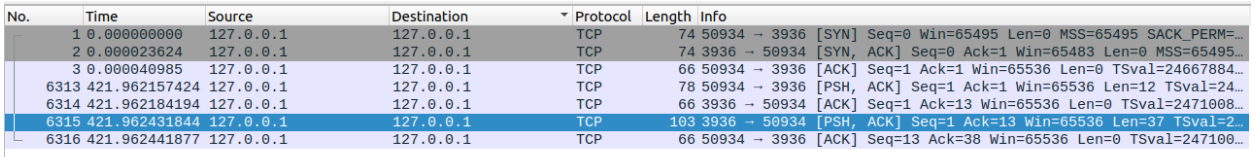


```
▸ Frame 6313: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 50934, Dst Port: 3936, Seq: 1, Ack: 1, Len: 12
▾ Data (12 bytes)
    Data: 0c000000000000000000000000
    [Length: 12]
```

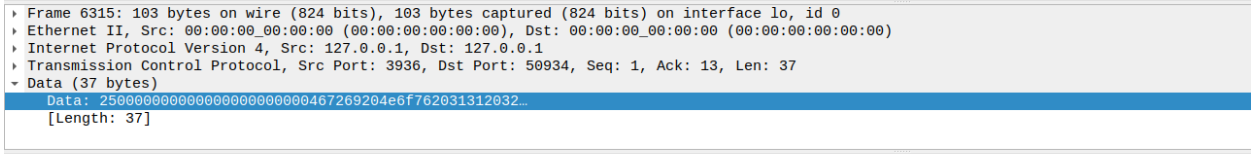
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ······ ·· ·····E·
0010  00 40 ab c9 40 00 40 06  90 ec 7f 00 00 01 7f 00   ·@··@·@· ·· ····
0020  00 01 c6 f6 0f 60 0c eb  bb 23 59 09 ef f1 80 18   ····`·· ·#Y·····
0030  02 00 fe 34 00 00 01 01  08 0a 0e ba 75 87 0e b4   ···4···· ····u···
0040  05 3d 0c 00 00 00 00 00  00 00 00 00               ·=···· ·· ····
```

如图 0c 00 00 00 对应 length 字段，表示数据包总长度为 12 字节，而后的 00 00 00 00 对应请求类型 Time。最后的 00 00 00 00 是预留给 ClientId 的，可以忽略。

响应包：



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	50934 → 3936 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=…
2	0.000023624	127.0.0.1	127.0.0.1	TCP	74	3936 → 50934 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495…
3	0.000040985	127.0.0.1	127.0.0.1	TCP	66	50934 → 3936 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=24667884…
6313	421.962157424	127.0.0.1	127.0.0.1	TCP	78	50934 → 3936 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=24…
6314	421.962184194	127.0.0.1	127.0.0.1	TCP	66	3936 → 50934 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=2471008…
6315	421.962431844	127.0.0.1	127.0.0.1	TCP	103	3936 → 50934 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=37 TSval=2…
6316	421.962441877	127.0.0.1	127.0.0.1	TCP	66	50934 → 3936 [ACK] Seq=13 Ack=38 Win=65536 Len=0 TSval=247100…



```
▸ Frame 6315: 103 bytes on wire (824 bits), 103 bytes captured (824 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 3936, Dst Port: 50934, Seq: 1, Ack: 13, Len: 37
▾ Data (37 bytes)
    Data: 250000000000000000000000467269204e6f762031312032…
    [Length: 37]
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ······ ·· ·····E·
0010  00 59 47 1e 40 00 40 06  f5 7e 7f 00 00 01 7f 00   ·YG·@·@· ·~ ····
0020  00 01 0f 60 c6 f6 59 09  ef f1 0c eb bb 2f 80 18   ···`··Y· ····/··
0030  02 00 fe 4d 00 00 01 01  08 0a 0e ba 75 88 0e ba   ···M···· ····u···
0040  75 87 25 00 00 00 00 00  00 00 00 00 46 72        u·%····· ····Fr
0050  69 20 4e 6f 76 20 31 31  20 32 31 3a 33 3a 34      i Nov 11  21:13:4
0060  30 20 32 30 32 32 0a                              0 2022·
```

如图 25 00 00 00 对应 length 字段，表示数据包总长度为 37 字节，而后的 00 00 00 00 对应响应类型 Time，从 46 72 开始到末尾的数据对应返回的时间数据。

- 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：

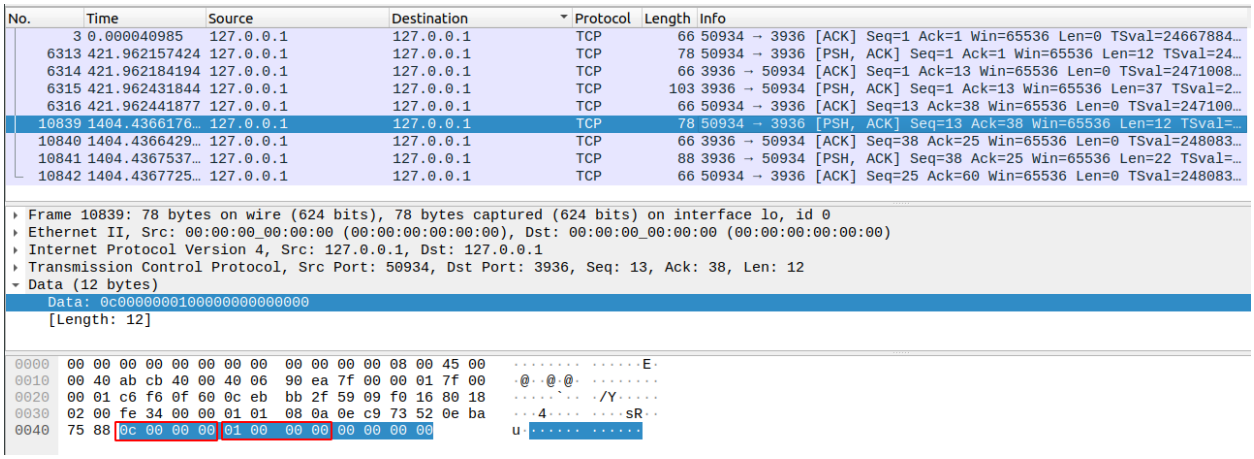
```
n
Server Machine Name: yyb-laptop
```

服务器：

```
[PROMPT] Machine name has been sent to client with id[0]
```

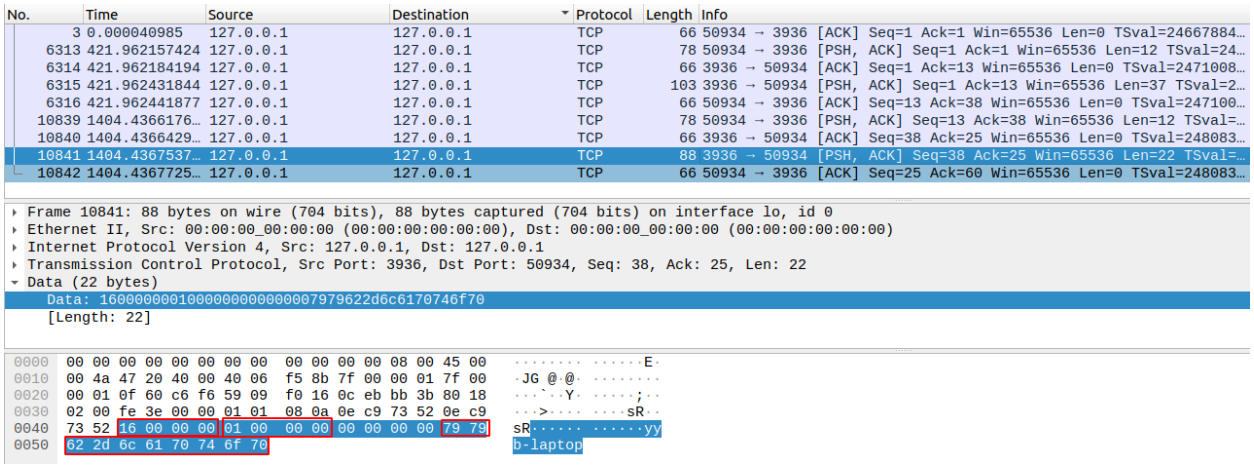
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的名字数据对应的位置）：

请求包



如图 0c 00 00 00 对应 length 字段，表示数据包总长度为 12 字节，而后的 01 00 00 00 对应请求类型 Name。

响应包



如图 16 00 00 00 对应 length 字段，表示数据包总长度为 22 字节，而后的 01 00 00 00 对应响应类型 Name，从 79 79 开始到末尾的数据对应返回的机器名数据。

相关的服务器的处理代码片段：

```
void sendName(int sockfd) {
    Message msg = Message();
    char hostname[20];
    gethostname(hostname, 20);
    string s = string(hostname, strlen(hostname));
    msg.setMessage(Type::Name, 0, s);
    msg.sendMessage(sockfd);
}
```

- 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：

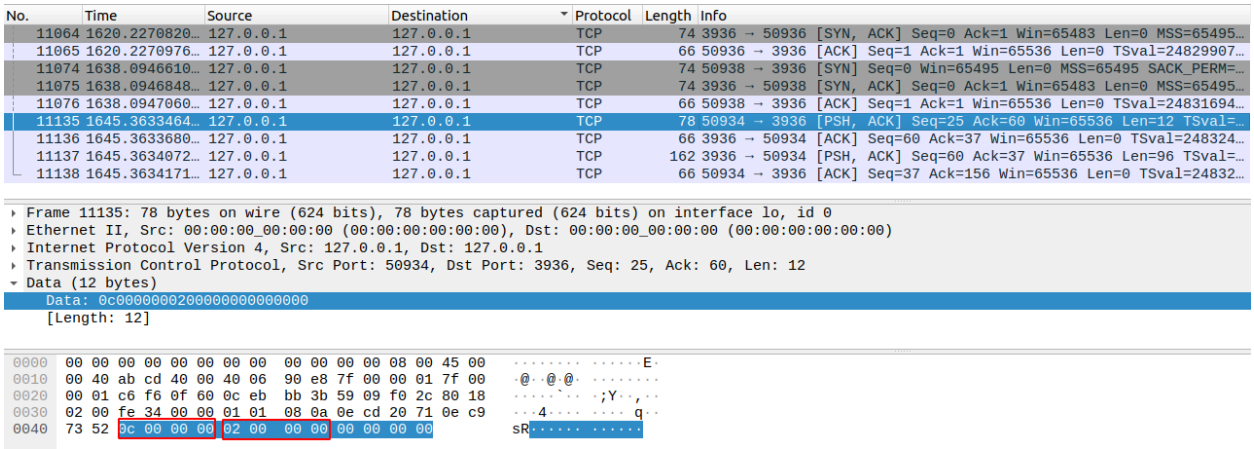
```
l
Clients' Information:
[ClientId]0 [IPAddress]127.0.0.1 [Port]37128
[ClientId]1 [IPAddress]127.0.0.1 [Port]37130
[ClientId]2 [IPAddress]127.0.0.1 [Port]37132
```

服务端：

```
[PROMPT] Client list has been sent to client with id[0]
```

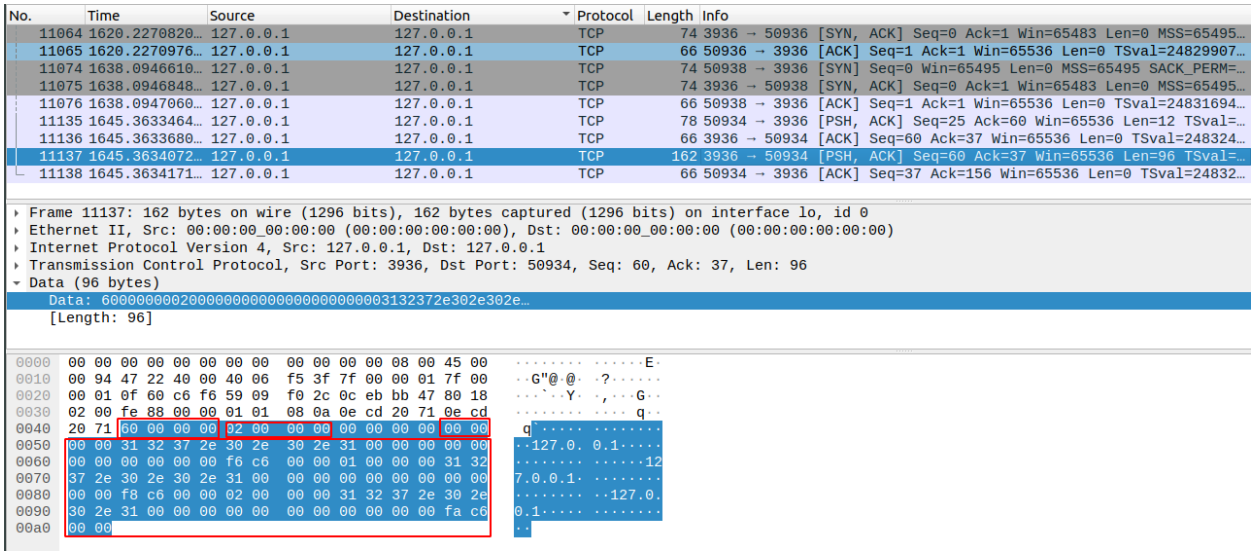
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：

请求包



如图 0c 00 00 00 对应 length 字段，表示数据包总长度为 12 字节，而后的 02 00 00 00 对应请求类型 List。

响应包



如图 60 00 00 00 对应 length 字段，表示数据包总长度为 96 字节，而后的 02 00 00 00 对应响应类型 List，从 00 00 开始到末尾的数据对应返回的客户端列表数据。

相关的服务器的处理代码片段：

```
void sendList(int sockfd) {
    Message msg = Message();
    uint32_t dataLen = cList.serialClientInfos(msg.data);
    msg.type = Type::List;
    msg.clientId = 0;
    msg.length = dataLen + sizeof(Type) + 2*sizeof(uint32_t);
    msg.sendMessage(sockfd);
}
```

- 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：

```
l
Clients' Information:
[ClientId]0 [IPAddress]127.0.0.1 [Port]37128
[ClientId]1 [IPAddress]127.0.0.1 [Port]37130
[ClientId]2 [IPAddress]127.0.0.1 [Port]37132
s
[PROMPT] Type the message you want to send in next lines, end with symbol #
Hello Client 2
Have a good day~#
[PROMPT] Type the ClientId you want to send to in the next line
2
Message sent successfully
```

服务器：

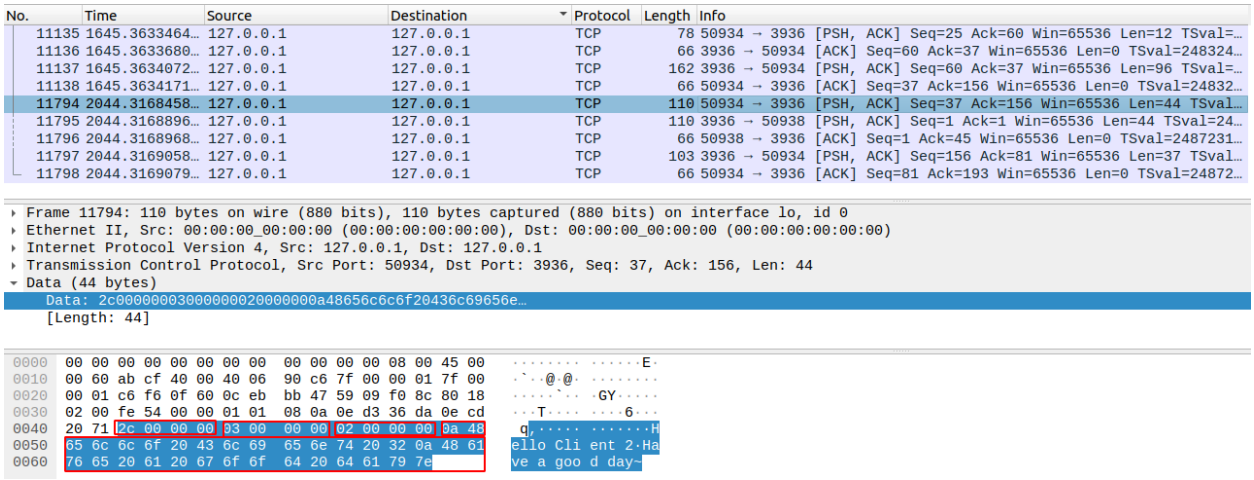
```
[PROMPT] Receive message from client with id[0]
[PROMPT] Sending message to client with id[2]
[PROMPT] Message sent successfully
```

接收消息的客户端：

```
Message from client with id[0]:
Hello Client 2
Have a good day~
```

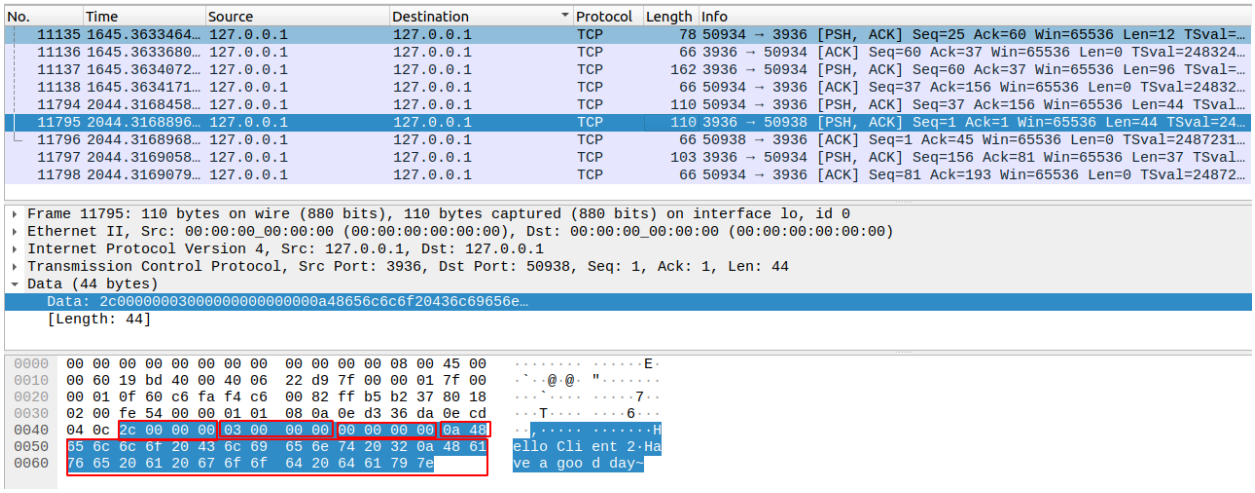
Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送方请求包



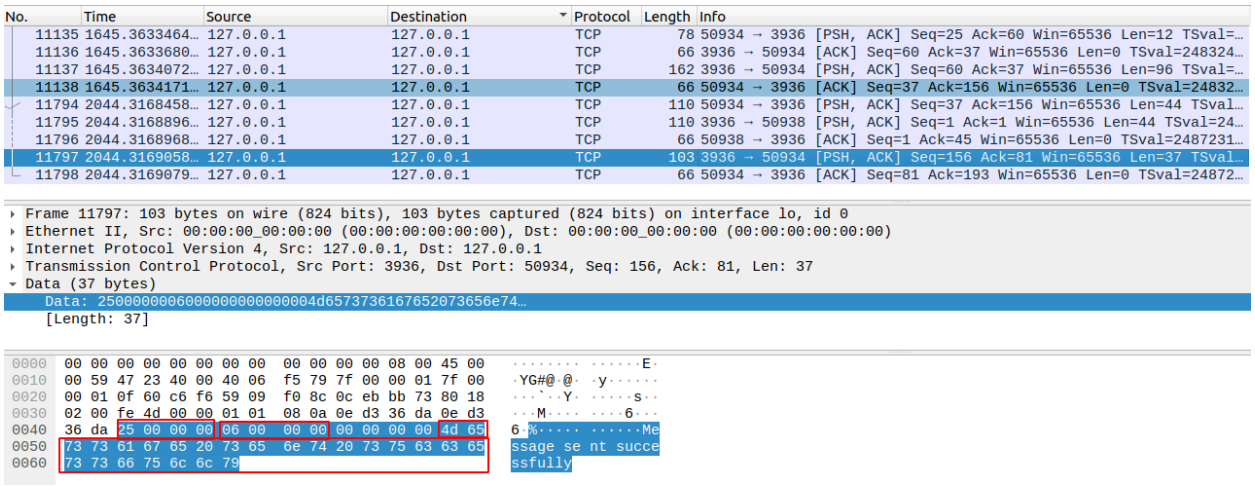
如图 2c 00 00 00 对应 length 字段，表示数据包总长度为 44 字节，而后的 03 00 00 00 对应请求类型 Mess，02 00 00 00 对应目标 ClientId 2，从 0a 48 开始到末尾的数据对应源客户发送的消息数据。

接收方收到的指示包



如图 2c 00 00 00 对应 length 字段，表示数据包总长度为 44 字节，而后的 03 00 00 00 对应指示类型 Mess，00 00 00 00 对应源 ClientId 0，从 0a 48 开始到末尾的数据对应目标客户接收的消息数据。

发送方收到的响应包



如图 25 00 00 00 对应 length 字段，表示数据包总长度为 37 字节，而后的 06 00 00 00 对应响应类型 ReplySendStatus，从 4d 65 开始到末尾的数据对应源客户端接收的告知发送成功与否的信息。

相关的服务器的处理代码片段：

```
int transMessage(uint32_t srcClientId, Message& revMsg) {
    uint32_t desClientId = revMsg.clientId;
    cout << PROMPT << "Receive message from client with id[" << srcClientId << "]" << endl;
    cout << PROMPT << "Sending message to client with id[" << desClientId << "]" << endl;
    int sockfd = cList.getSockfd(desClientId);
    if(sockfd == -1) return -1;
    revMsg.clientId = srcClientId; // change to source clientId
    revMsg.sendMessage(sockfd); // send to destination clientId
    return 0;
}
```

相关的客户端（发送和接收消息）处理代码片段：

发送消息：

```
void sendMessage(int sockfd) {
  uint32_t clientId;
  string data;
  printPrompt("Type the message you want to send in next lines, end with symbol #");
  getline(cin, data, '#');
  printPrompt("Type the ClientId you want to send to in the next line");
  cin >> clientId;
  msg.setMessage(Type::Mess, clientId, data);
  msg.sendMessage(sockfd);
}
```

接收消息：

```
} else if(msg.type == Type::Mess) {
  cout << "Message from client with id[" << msg.clientId << "]: ";
  cout << msg.data << 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。

  发送了 TCP 连接释放的消息，但是在另一个客户端中查看客户列表发现并没有发生变化：

  ```
  l
  Clients' Information:
  [ClientId]0 [IPAddress]127.0.0.1 [Port]50950
  [ClientId]1 [IPAddress]127.0.0.1 [Port]50952
  ```

  说明该连接依然存在

- 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？

  该连接还存在，但是无法向其发送消息(发送失败)。

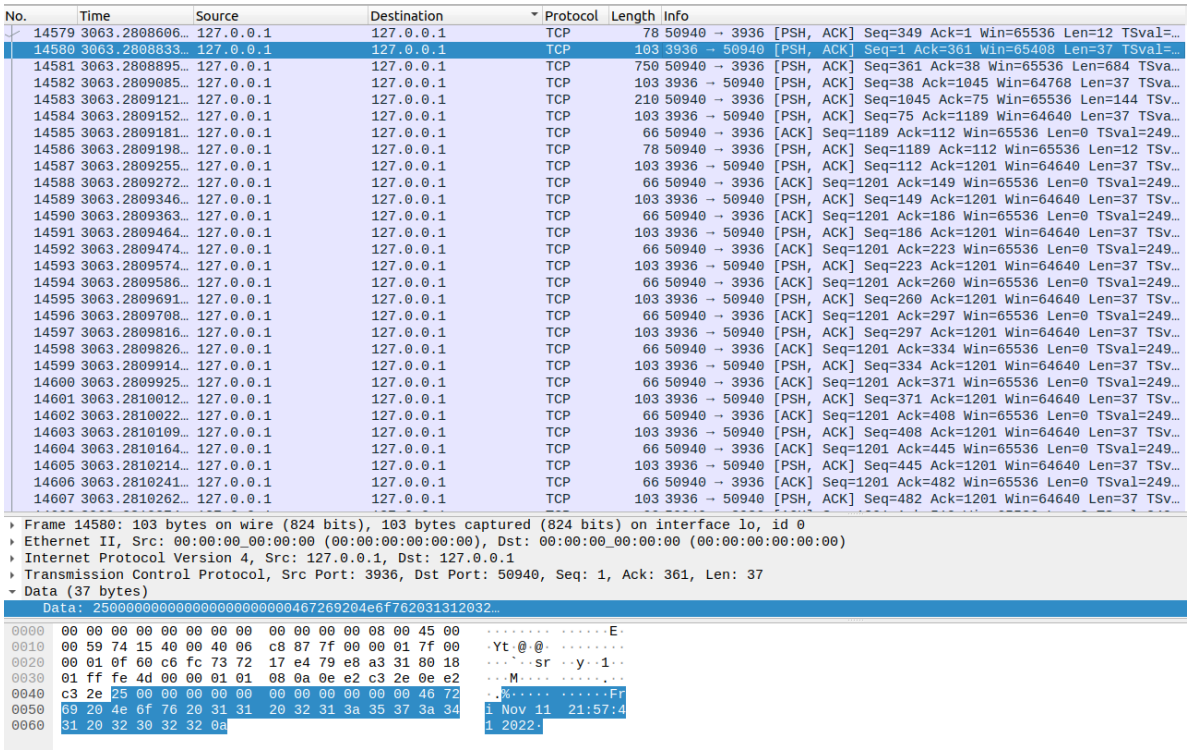
- 修改获取时间功能，改为用户选择 1 次，程序内自动发送 100 次请求。服务器是否正常处理了 100 次请求，截取客户端收到的响应（通过程序计数一下是否有 100 个响应回来），并使用 Wireshark 抓取数据包，观察实际发出的数据包个数。

客户端收到的响应：

```
 67   Time: Fri Nov 11 21:57:41 2022
 68   Time: Fri Nov 11 21:57:41 2022
 69   Time: Fri Nov 11 21:57:41 2022
 70   Time: Fri Nov 11 21:57:41 2022
 71   Time: Fri Nov 11 21:57:41 2022
 72   Time: Fri Nov 11 21:57:41 2022
 73   Time: Fri Nov 11 21:57:41 2022
 74   Time: Fri Nov 11 21:57:41 2022
 75   Time: Fri Nov 11 21:57:41 2022
 76   Time: Fri Nov 11 21:57:41 2022
 77   Time: Fri Nov 11 21:57:41 2022
 78   Time: Fri Nov 11 21:57:41 2022
 79   Time: Fri Nov 11 21:57:41 2022
 80   Time: Fri Nov 11 21:57:41 2022
 81   Time: Fri Nov 11 21:57:41 2022
 82   Time: Fri Nov 11 21:57:41 2022
 83   Time: Fri Nov 11 21:57:41 2022
 84   Time: Fri Nov 11 21:57:41 2022
 85   Time: Fri Nov 11 21:57:41 2022
 86   Time: Fri Nov 11 21:57:41 2022
 87   Time: Fri Nov 11 21:57:41 2022
 88   Time: Fri Nov 11 21:57:41 2022
 89   Time: Fri Nov 11 21:57:41 2022
 90   Time: Fri Nov 11 21:57:41 2022
 91   Time: Fri Nov 11 21:57:41 2022
 92   Time: Fri Nov 11 21:57:41 2022
 93   Time: Fri Nov 11 21:57:41 2022
 94   Time: Fri Nov 11 21:57:41 2022
 95   Time: Fri Nov 11 21:57:41 2022
 96   Time: Fri Nov 11 21:57:41 2022
 97   Time: Fri Nov 11 21:57:41 2022
 98   Time: Fri Nov 11 21:57:41 2022
 99   Time: Fri Nov 11 21:57:41 2022
100   Time: Fri Nov 11 21:57:41 2022
```

确实有 100 个响应

抓包如下：



No.	Time	Source	Destination	Protocol	Length	Info
14579	3063.2808606…	127.0.0.1	127.0.0.1	TCP	78	50940 → 3936 [PSH, ACK] Seq=349 Ack=1 Win=65536 Len=12 TSval=…
14580	3063.2808833…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=1 Ack=361 Win=65408 Len=37 TSval=…
14581	3063.2808895…	127.0.0.1	127.0.0.1	TCP	750	50940 → 3936 [PSH, ACK] Seq=361 Ack=38 Win=65536 Len=684 TSva…
14582	3063.2809085…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=38 Ack=1045 Win=64768 Len=37 TSva…
14583	3063.2809121…	127.0.0.1	127.0.0.1	TCP	210	50940 → 3936 [PSH, ACK] Seq=1045 Ack=75 Win=65536 Len=144 TSv…
14584	3063.2809152…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=75 Ack=1189 Win=64640 Len=37 TSva…
14585	3063.2809181…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1189 Ack=112 Win=65536 Len=0 TSval=249…
14586	3063.2809198…	127.0.0.1	127.0.0.1	TCP	78	50940 → 3936 [PSH, ACK] Seq=1189 Ack=112 Win=65536 Len=12 TSv…
14587	3063.2809255…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=112 Ack=1201 Win=64640 Len=37 TSv…
14588	3063.2809272…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=149 Win=65536 Len=0 TSval=249…
14589	3063.2809346…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=149 Ack=1201 Win=64640 Len=37 TSv…
14590	3063.2809363…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=186 Win=65536 Len=0 TSval=249…
14591	3063.2809464…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=186 Ack=1201 Win=64640 Len=37 TSv…
14592	3063.2809474…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=223 Win=65536 Len=0 TSval=249…
14593	3063.2809574…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=223 Ack=1201 Win=64640 Len=37 TSv…
14594	3063.2809586…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=260 Win=65536 Len=0 TSval=249…
14595	3063.2809691…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=260 Ack=1201 Win=64640 Len=37 TSv…
14596	3063.2809708…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=297 Win=65536 Len=0 TSval=249…
14597	3063.2809816…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=297 Ack=1201 Win=64640 Len=37 TSv…
14598	3063.2809826…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=334 Win=65536 Len=0 TSval=249…
14599	3063.2809914…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=334 Ack=1201 Win=64640 Len=37 TSv…
14600	3063.2809925…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=371 Win=65536 Len=0 TSval=249…
14601	3063.2810012…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=371 Ack=1201 Win=64640 Len=37 TSv…
14602	3063.2810022…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=408 Win=65536 Len=0 TSval=249…
14603	3063.2810109…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=408 Ack=1201 Win=64640 Len=37 TSv…
14604	3063.2810164…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=445 Win=65536 Len=0 TSval=249…
14605	3063.2810214…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=445 Ack=1201 Win=64640 Len=37 TSv…
14606	3063.2810241…	127.0.0.1	127.0.0.1	TCP	66	50940 → 3936 [ACK] Seq=1201 Ack=482 Win=65536 Len=0 TSval=249…
14607	3063.2810262…	127.0.0.1	127.0.0.1	TCP	103	3936 → 50940 [PSH, ACK] Seq=482 Ack=1201 Win=64640 Len=37 TSv…

```
▸ Frame 14580: 103 bytes on wire (824 bits), 103 bytes captured (824 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 3936, Dst Port: 50940, Seq: 1, Ack: 361, Len: 37
▾ Data (37 bytes)
    Data: 250000000000000000000000467269204e6f762031312032…

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ·····E·
0010  00 59 74 15 40 00 40 06  c8 87 7f 00 00 01 7f 00   ·Yt·@·@·
0020  00 01 0f 60 c6 fc 73 72  17 e4 79 e8 a3 31 80 18   ···`··sr··y··1··
0030  01 ff fe 4d 00 00 01 01  08 0a 0e e2 c3 2e 0e e2   ···M····
0040  c3 2e 25 00 00 00 00 00  00 00 00 00 00 00 46 72   ··%·····
0050  69 20 4e 6f 76 20 31 31  20 32 31 3a 35 37 3a 34   i Nov 11  21:57:4
0060  31 20 32 30 32 32 0a                              1 2022·
```

实际发出的个数 100，说明服务器响应速度较快

- 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：

```
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[0]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
[PROMPT] Time has been sent to client with id[1]
```

客户端 1：

```
t
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
Time: Fri Nov 11 22:20:39 2022
```

客户端 2：

```
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
Time: Fri Nov 11 22:20:40 2022
```

在很短时间内就已经完成了 100 次响应，所以没有体现出交替执行。

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？

不需要调用 bind 操作。由操作系统内核分配。客户端的端口可能会发生变化，因为这是由系统分配的空闲端口。

- 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？

可以连接成功

- 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？

取决于服务器响应速度，对于我编写的程序，连续 100 次 send 后二者是一致的。

- 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？

在客户端发送信息的 send 函数中，会有一项 sockfd 字段。该字段唯一确定地标记了 socket 连接，所以可以通过 sockfd 来区分客户端。

- 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）

  TIME_WAIT，保持了不到一分钟。

- 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？

客户端会发送一个结束报文给服务器，客户端连接将变为 FIN_WAIT_1 状态。若此时客户端收到服务器专门用于确认的 ACK 报文,则连接转移至 FIN_WAIT_2 状态。当客户端处于 FIN_WAIT_2 状态时，服务器处于 CLOSE_WAIT 状态，这一对状态是有可能发生半关闭的状态。可以采用一种服务器检验连接是否有效的方法：客户端每隔 3 秒向用户器发送一次请求，同样的，服务器会不断监听来自客户端的“心跳包”，并在屏幕上显示。如果超过 5s 没有收到包的话，则会断开连接与此客户端的连接。

七、 讨论、心得

通过本次实验，我学习了 linux 下的 socket 编程，熟悉了建立 socket 连接的流程，也对网络连接的建立有了更深的理解。Socket 编程本身是比较范式化的，所以学习了一些样例后就能自己使用，难点主

要是在多线程管理上。为了实现主线程通知子线程关闭，自己再安全关闭的功能，我学习了 signal 的用法，通过 signal 可以触发中断，从而设置一些全局/原子变量，使子线程从循环中退出。但是在子线程退出前，必须将其设置为 detach 或者使其被 join，否则可能会导致程序崩溃。虽然子线程可以退出，但服务端主线程被 accept 阻塞，如何安全退出呢？为此我学习了 setjmp 和 longjmp，利用该机制，使得主线程被 signal 中断时通过 longjmp 跳出阻塞函数，然后在释放资源后 return 0 退出。其实最花时间的在于异常情况的处理，比如实现 client 突然关闭时，服务器不受影响，服务器突然关闭时客户端不会直接退出等。同时有时候进程内线程通信比较困难时，可以利用 syn 和 ack 通过端之间的通信辅助线程间通信。总而言之，通过代码实践，更好地理解了 socket 通信，也学到了很多知识。推荐一本 C++多线程相关的开源书籍：xiaoweiChen/Cpp_Concurrency_In_Action: 作为对《C++ Concurrency in Action》英文版的中文翻译。 (github.com)