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：	刘轩铭
学　　院：	计算机学院
系：	计算机系
专　　业：	软件工程
学　　号：	3180106071
指导教师：	邱劲松

2020 年　　12 月　　21 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**

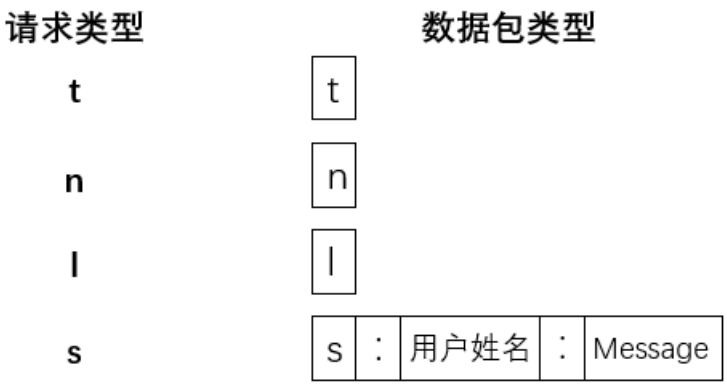
以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- **描述请求数据包的格式（画图说明），请求类型的定义**

请求类型的定义如下所示：

请求类型	定义
t	要求服务器返回当前时间
n	要求服务器返回自己的用户名
l	要求服务器返回当前连接的所有用户的信息
s	给一个特定的用户（同连在这个服务器上的）发送消息

请求数据包格式如下所示：



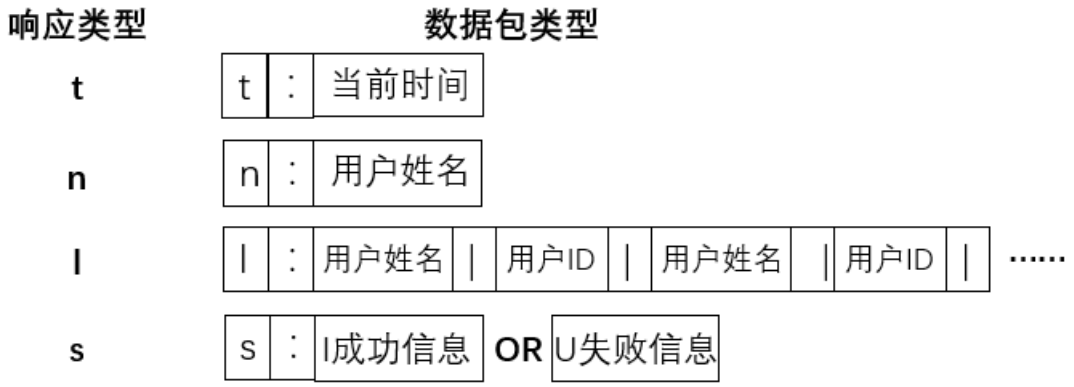
- **描述响应数据包的格式（画图说明），响应类型的定义**

响应类型的定义如下所示：

响应类型	定义
t	服务器返回的当前时间
n	服务器返回的用户名
l	服务器返回的当前连接的所有用户的信息

s	信息发送是否成功

响应数据包格式如下所示：

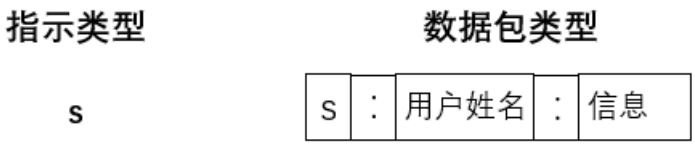


- **描述指示数据包的格式（画图说明），指示类型的定义**

指示类型的定义如下所示：

指示类型	定义
s	转发给目的客户端的信息

指示数据包格式如下所示：



- **客户端初始运行后显示的菜单选项**

```
INFO: Client of LAB7
INFO: Initializing socket...
Here is the help information!
+-------+--------------------------------------+
| Input |              Function                |
+-------+--------------------------------------+
|   c   |  Connect to a server.                |
|   d   |  Disconnect from the server.         |
|   t   |  Get the current time on the server. |
|   n   |  Get who I am.                       |
|   l   |  List and update users on the server.|
|   s   |  Send a message to a specified user. |
|   q   |  Disconnect and quit.                |
|   h   |  Show this help infomation.          |
+-------+--------------------------------------+
e.g. Input "h" will show this help information.
```

如上图所示，初始运行之后，会显示命令界面。

- **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）**

```
void Client::HandleCmd(string cmd)
{
    if(cmd.empty()) return;
    if( cmd == "c") {
        ClientConnect();
    }
    else if( cmd == "d" ) {
        ClientDisconnect();
    }
    else if(cmd == "t") {
        ClientGetTime();
    }
    else if(cmd == "n") {
        ClientGetName();
    }
    else if(cmd == "l") {
        ClientGetUserList();
    }
    else if(cmd == "s") {
        ClientSendMessage();
    }
    else if(cmd == "q") {
        ClientQuit();
    }
    else if(cmd == "h") {
        ShowHelpInfo();
    }
    else {
        cout << "INFO: Not legal command." << endl;
    }
```

如图所示，客户端的主线程循环，不断接受用户的命令请求，然后调用相应的函数，去解决相应请求。如果没有相应的请求，则输出“No legal command”结束。

- **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 Client::ProcessReplyMsg(char* msg)
{
    if ((!msg) || msg[0] == '\0')
        return;
    switch (msg[0])
    {
        case 't':
        case 'n':
            cout << msg << endl;
            break;
        case 'l': {
            cout << "Nickname" << "\t\t" << "Connfd" << endl;
            char* processMsg = msg + 2;
            int startPos = 0, endPos = 0;
            int totalNum = 0;
            for(int i = 0; i < strlen(processMsg); i++) {
                if(processMsg[i] == '|') {
                    endPos = i;
                    string subStr = string(processMsg).substr(startPos, endPos-startPos);
                    cout << subStr << "\t";
                    if(!((++totalNum) % 2)) cout << endl;
                    startPos = endPos + 1;
                }
            }
            break;
        }
        case 's':
            cout << msg << endl;
            break;
        default:
            break;
```

该部分处理服务器端发送过来的数据，并以人类可以接受的方式输出。例如，用户之间的通讯消息，服务器与客服端传输的格式是"%s|%d|",也就是信息本身字符串 + | +该用户的 ID。而客户端通过调用函数，将该消息以列表的形式展现给用户。

- **服务器初始运行后显示的界面**

```
SERVER OF LAB7
INFO: Initializing server...
INFO: Binding socket...
INFO: Listening socket...
INFO: Offering service...
```

- **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) {
    pthread_t pid;
    ClientInfo* thisClientInfoPtr = new ClientInfo();
    struct sockaddr_in clientAddr;
    socklen_t len = sizeof(clientAddr);
    SOCKET connfd = accept(sockfd, (struct sockaddr*)&clientAddr, &len);
    if(connfd == INVALID_SOCKET) {
        cout << "ERROR: Connecting client meets error." << endl;
        closesocket( sockfd );
        WSACleanup();
        exit( _Code: -1);
    }
    else {
        cout << "INFO: Creating thread with connfd = " << connfd << " ." << endl;
        this->clientNum++;
        cout << "INFO: There has been " << this->clientNum << " clients" << endl;
        thisClientInfoPtr->clientAddr = clientAddr;
        thisClientInfoPtr->connfd = connfd;
        thisClientInfoPtr->nickName = "Client Piggy " + std::to_string(connfd);
        thisClientInfoPtr->isBreak = false;
        this->clientMap[connfd] = *thisClientInfoPtr;
        pthread_create(&pid, NULL, ClientThread, &connfd);
    }
}
```

- 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* Server::ClientThread(void* args)
{
    // from args get connfd.
    int connfd = *(int*)args;
    cout << "INFO[" << connfd << "]: New thread created" << endl;

    while(true) {
        if(clientMap[connfd].isBreak) {
            cout << "INFO[" << connfd << "]: Client disconnect." << endl;
            cout << "INFO[" << connfd << "]: Changing client list..." << endl;
            clientMap.erase(connfd);
            clientNum--;
            cout << "INFO[" << connfd << "]: Closing socket..." << endl;
            close(connfd);
            break;
        }
        char buf[1024] = { 0 };
        int ret = recv(connfd, buf, sizeof(buf),  flags: 0);
        if (ret > 0) {
            ProcessMessage(connfd, buf);
        }
        else if (ret == 0) {
            // client exit
            cout << "INFO[" << connfd << "]: Client disconnect." << endl;
            cout << "INFO[" << connfd << "]: Changing client list..." << endl;
            clientMap.erase(connfd);
            clientNum--;
            cout << "INFO[" << connfd << "]: Closing socket..." << endl;
            close(connfd);
            break;
        }
```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

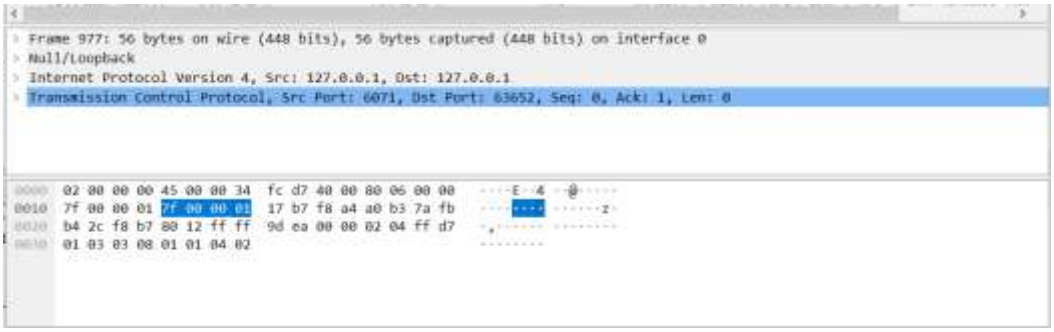
客户端：

```
c
INFO: Please input server address:
127.0.0.1
INFO: Please input server port number:
6071
INFO: Successfully connect to 127.0.0.1 : 6071
```

服务端：

```
SERVER OF LAB7
INFO: Initializing server...
INFO: Binding socket...
INFO: Listening socket...
INFO: Offering service...
INFO: Creating thread with connfd = 240 .
INFO: There has been 1 clients
INFO[240]: New thread created
```

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。**

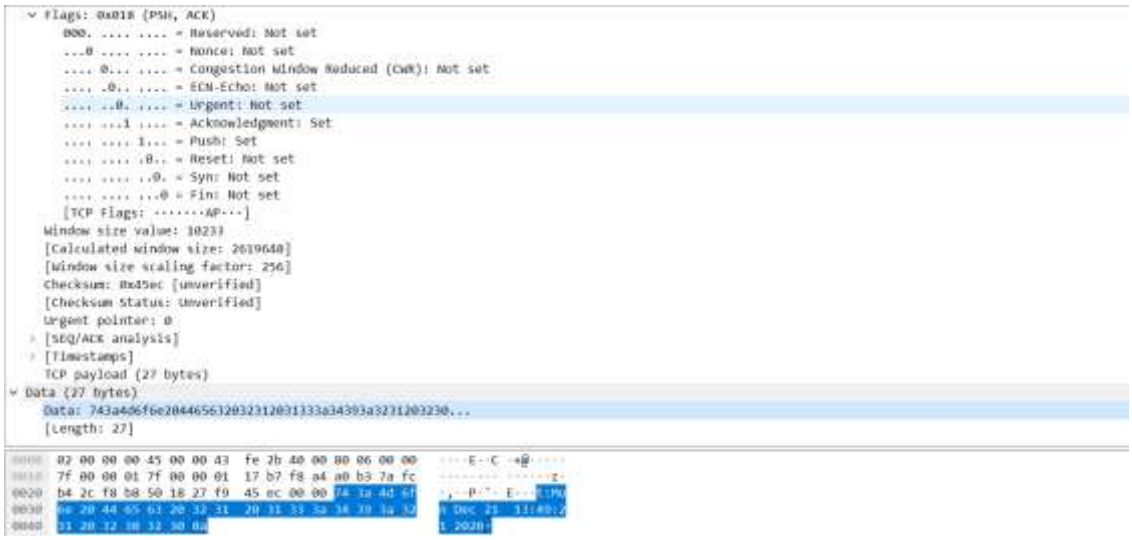
客户端：

```
t
t:Mon Dec 21 13:49:21 2020
```

服务端：

```
INFO[236]: Time information t:Mon Dec 21 13:49:21 2020
 has been sent out.
```

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）：



- **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。**

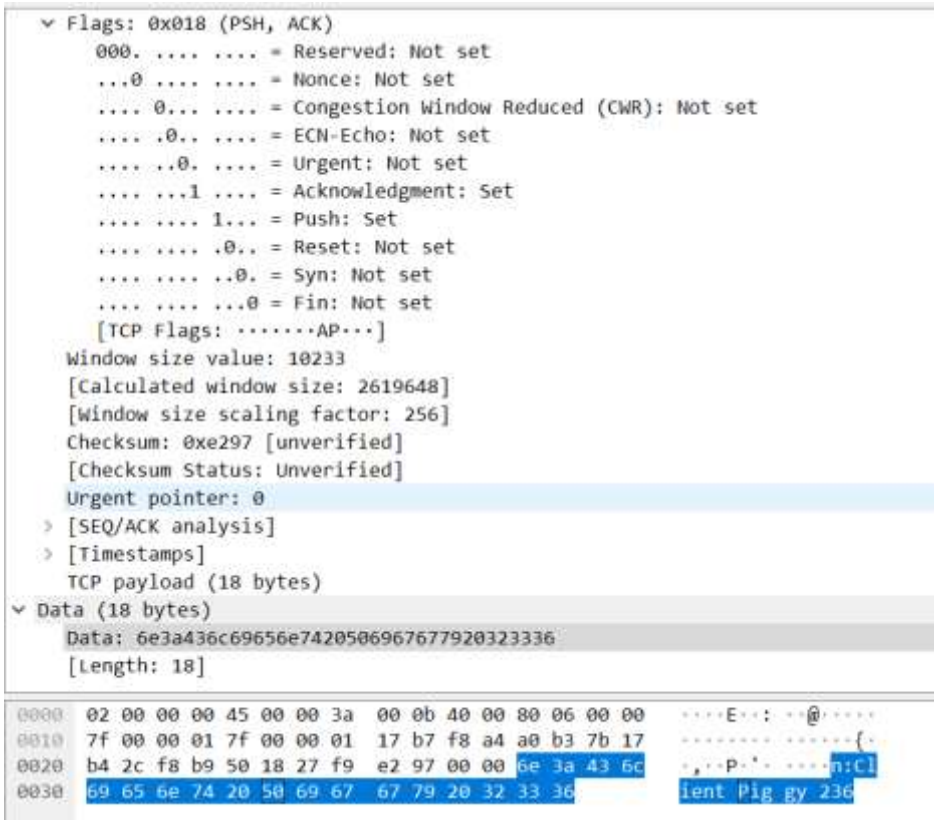
客户端：

```
n
n:Client Piggy 236
```

服务端：

```
INFO[236]: Name information n:Client Piggy 236 has been sent out.
```

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：

```
void Server::SendOutName(int connfd)
{
    char retMsg[MAX_MSG_LEN];
    sprintf(retMsg, format: "n:%s", clientMap[connfd].nickName.c_str());
    SendOutMsg(connfd, retMsg);
    cout << "INFO[" << connfd << "]: Name information " << retMsg <<" has been sent out." << endl;
}
```

```
void Client::SendOutMsg(const string& msg)
{
    send(sockfd, msg.c_str(), strlen(msg.c_str()), flags: 0);
}
```

- **客户端选择获取客户端列表功能时，客户端和服务端显示内容截图。**

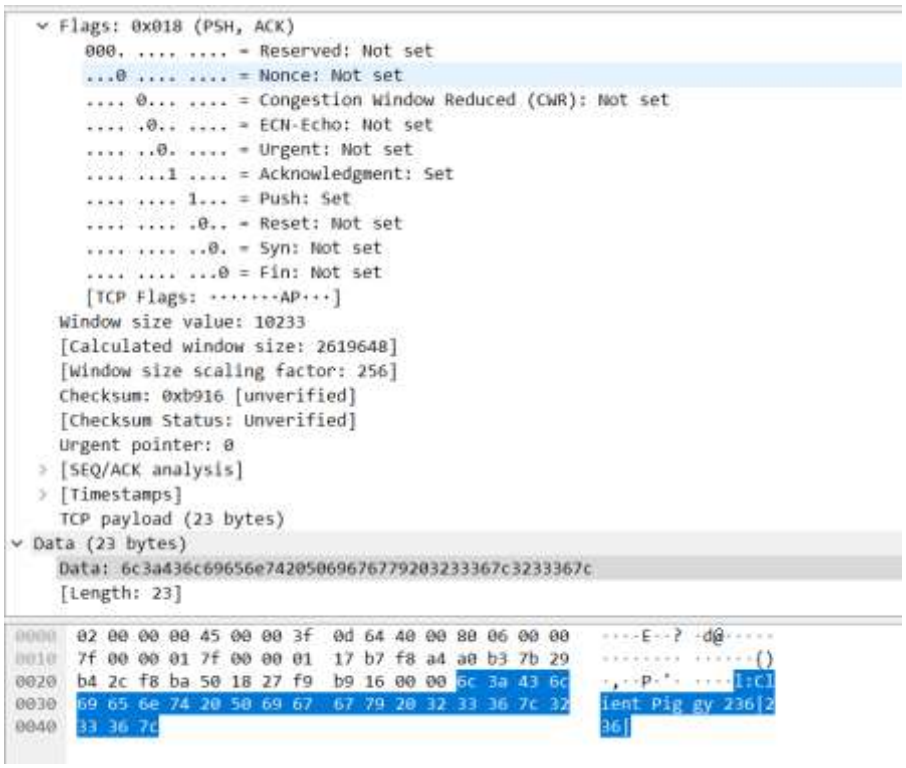
客户端：

```
Nickname                Connfd
Client Piggy 236        236
```

服务端：

```
INFO[236]: Users list information sent.
```

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：

```
void Server::SendOutUserList(int connfd)
{
    string retMsg = "l:";
    for (map<int, ClientInfo>::iterator i = clientMap.begin(); i != clientMap.end(); i++) {
        char userInfoStr[MAX_MSG_LEN];
        sprintf(userInfoStr, format: "%s|%d|", i->second.nickName.c_str(), i->second.connfd);
        retMsg += userInfoStr;
    }
    SendOutMsg(connfd, retMsg);
    cout << "INFO[" << connfd << "]: Users list information sent." << endl;
}
```

- **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
Nickname                Connfd
Client Piggy 236        236
Client Piggy 256        256
s
INFO: Please input the nickname of the user to send information:
Client Piggy 256
INFO: Please input the message to send:
Hello 256
INFO: Message sending out:
s:Information sent out.
```

服务器：

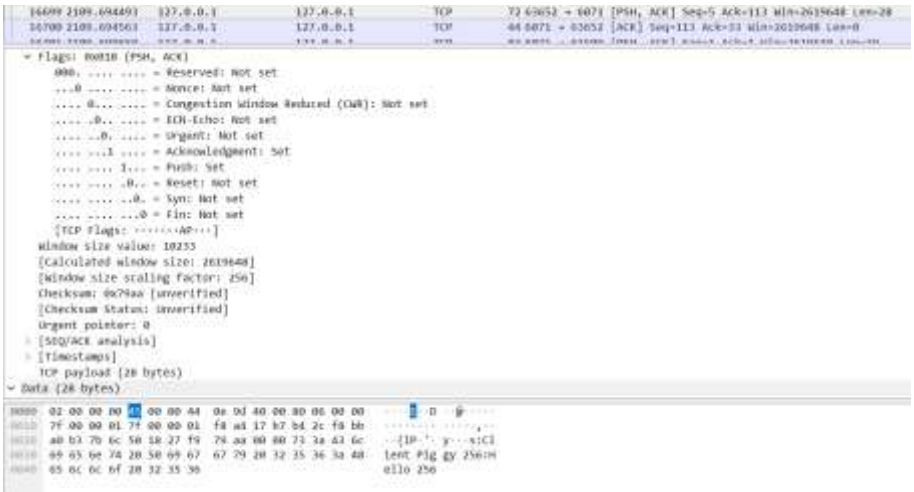
```
INFO[236] : Sending message to 256
INFO[236]: Information sent to 256 successfully
```

接收消息的客户端：

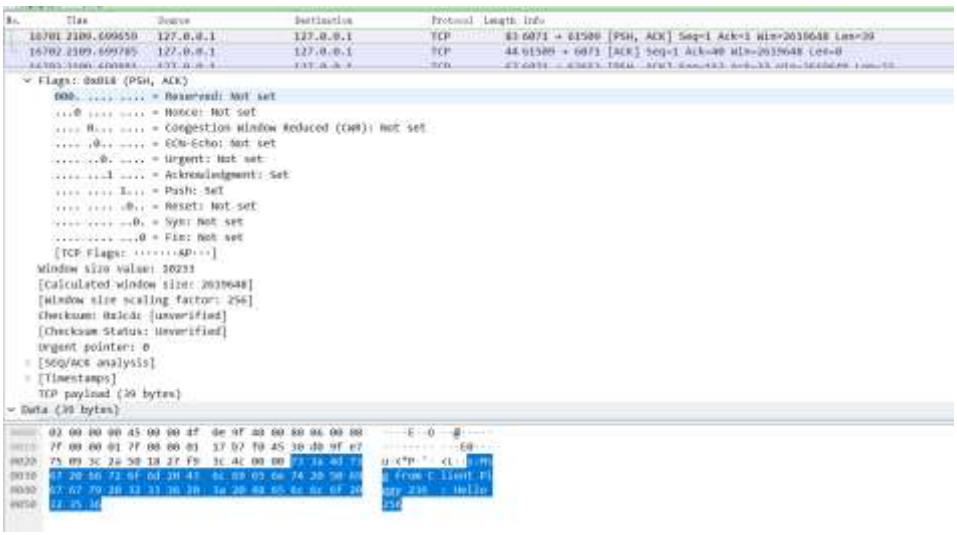
```
c
INFO: Please input server address:
127.0.0.1
INFO: Please input server port number:
6071
INFO: Successfully connect to 127.0.0.1 : 6071
s:Msg from Client Piggy 236 : Hello 256
```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送方：



接收方：



相关的服务器的处理代码片段：

```
void Server::SendOutTextMsg(int connfdFrom, int connfdTo, char* textMsg)
{
    cout << "INFO[" << connfdFrom << "] : Sending message to " << connfdTo << endl;
    string retMsg = "s:Msg from " + clientMap[connfdFrom].nickName + " : " + textMsg;
    SendOutMsg(connfdTo, retMsg);
    SendOutMsg(connfdFrom,  msg: "s:Information sent out.");
    cout << "INFO[" << connfdFrom << "]: Information sent to " << connfdTo << " successfully" << endl;
}
```

相关的客户端（发送和接收消息）处理代码片段：

```
void Client::ClientSendMessage()
{
    if (!isConnected) {
        cout << "INFO: You have not connected yet!" << endl;
        return;
    }
    char msgOut[1024];
    fflush( _File: stdin);
    cout << "INFO: Please input the nickname of the user to send information: " << endl;
    char nickName[80];
    gets(nickName);
    cout << "INFO: Please input the message to send:" << endl;
    char message[80];
    gets(message);
    sprintf(msgOut,  format: "%s:%s:%s", "s", nickName, message);
    SendOutMsg(msgOut);
    cout << "INFO: Message sending out: " << 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。

发送了消息：

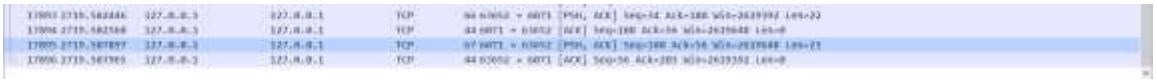
但是该消息并不会让服务端改变用户列表，通过另一个客户端再次查看发现：

```
s:Information sent out.
l
Nickname                Connfd
Client Piggy 236        236
Client Piggy 256        256
```

该连接仍然存在

- **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，发送消息不成功



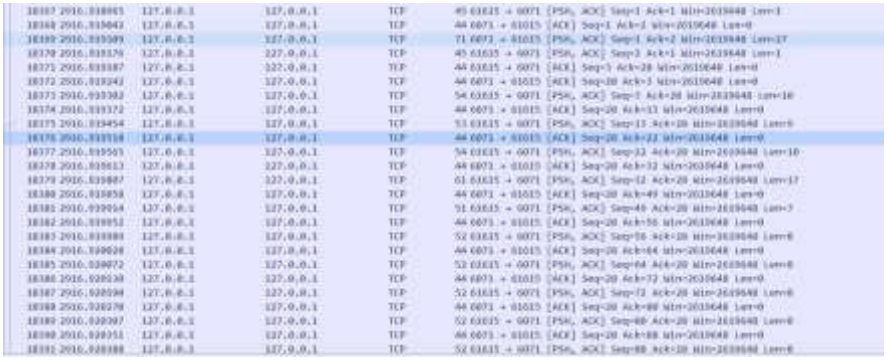
- **修改获取时间功能，改为用户选择 1 次，程序内自动发送 100 次请求。服务器是否正常处理了 100 次请求，截取客户端收到的响应（通过程序计数一下是否有 100 个响应回来），并使用 Wireshark 抓取数据包，观察实际发出的数据包个数。**

  客户端只收到了两个响应，说明不会有 100 个响应回来

```
t
t:Mon Dec 21 14:26:51 2020

t:Mon Dec 21 14:26:51 2020
```

Wireshark 观察：



实际发出的数据包也没有 100 个，具体个数，与服务器的响应速度有关。

- **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：

```
INFO[300]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
INFO[300]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
INFO[300]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
INFO[300]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
INFO[320]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
INFO[320]: Time information t:Mon Dec 21 14:32:04 2020
 has been sent out.
```

客户端 1：

```
t
t:Mon Dec 21 14:32:04 2020

t:Mon Dec 21 14:32:04 2020

t:Mon Dec 21 14:32:04 2020

t:Mon Dec 21 14:32:04 2020
```

客户端 2：

```
t
t:Mon Dec 21 14:32:04 2020

t:Mon Dec 21 14:32:04 2020
```

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？**

  不需要调用 bind 操作

  由系统内核产生源端口

  每一次调用 connect 时客户端的端口理论上都有可能变化，因为是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？**

  不完全一致。

- **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，即通过 sockfd 可以区分是哪个客户端。

- **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

  这个状态保持了不到 1 分钟就不复存在了。

- **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包给服务器，该连接状态变化，客户端连接进入

FIN_WAIT_1 状态。若此时客户端收到服务器专门用于确认目的的确认报文段(ACK)，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

实验过程中遇到的困难，得到的经验教训，对本实验安排的更好建议（看完请删除本句）

在本次实验中，对于 socket 编程有了深入的认识，send（）和 recv（）没有固定的对应关系，不定数目的 send()可以触发不定数目的 recv（）。send（）只负责拷贝，拷贝到内核就返回，此次 send（）调用所触发的程序错误，可能会在本次返回，也可能在下次调用网络 IO 函数的时候被返回。

在编写网络程序之前,对 socket 通信可以说一无所知,根本不知道服务器和客户端是如何进行通信的。为此我查阅了很多资料。一点点学会 socket 函数库的使用。并且结合操作系统中学到的多线程的知识，能够让服务器对应多个客户端，响应多个客户端的命令。

此外，我还了解到，在 Windows 和 Linux 下进行 Socket 编程的接口有些区别，而且流程也有区别。例如，Windows 下需要 WSADATA wsaData 数据段保存报文信息，而 Linux 下是没有这样的服务的。

在使用 Socket 的过程中会遇到读取响应消息超时的问题，在服务端还没有关闭连接前，客户端读取响应消息就会一直等待，直到超时。为此，我了解了 Socket 提供的 setSoTimeout(int timeout) 方法，在获得 Socket 的实例后，可以设置下超时时间，然后当 read 或是 readLine 完最后一个字节或是字符串后，或 break,或关掉连接。

该实验与操作系统的多线程相结合，可以有效的学习 socket 编程，具有很好的实践意义。实验的安排也做得非常好。