

# 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:	沈子衿
组 员:	沈子衿, 林宇翔
学 院:	计算机学院
系:	软件工程
专 业:	软件工程
学 号:	3160104734
指导教师:	董玮

2018 年 10 月 20 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： 林宇翔（负责 Server） 实验地点： 计算机网络实验室

### 一、 实验目的

- 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
- 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
- Visual C++、gcc 等 C++集成开发环境。

#### 四、 操作方法与实验步骤

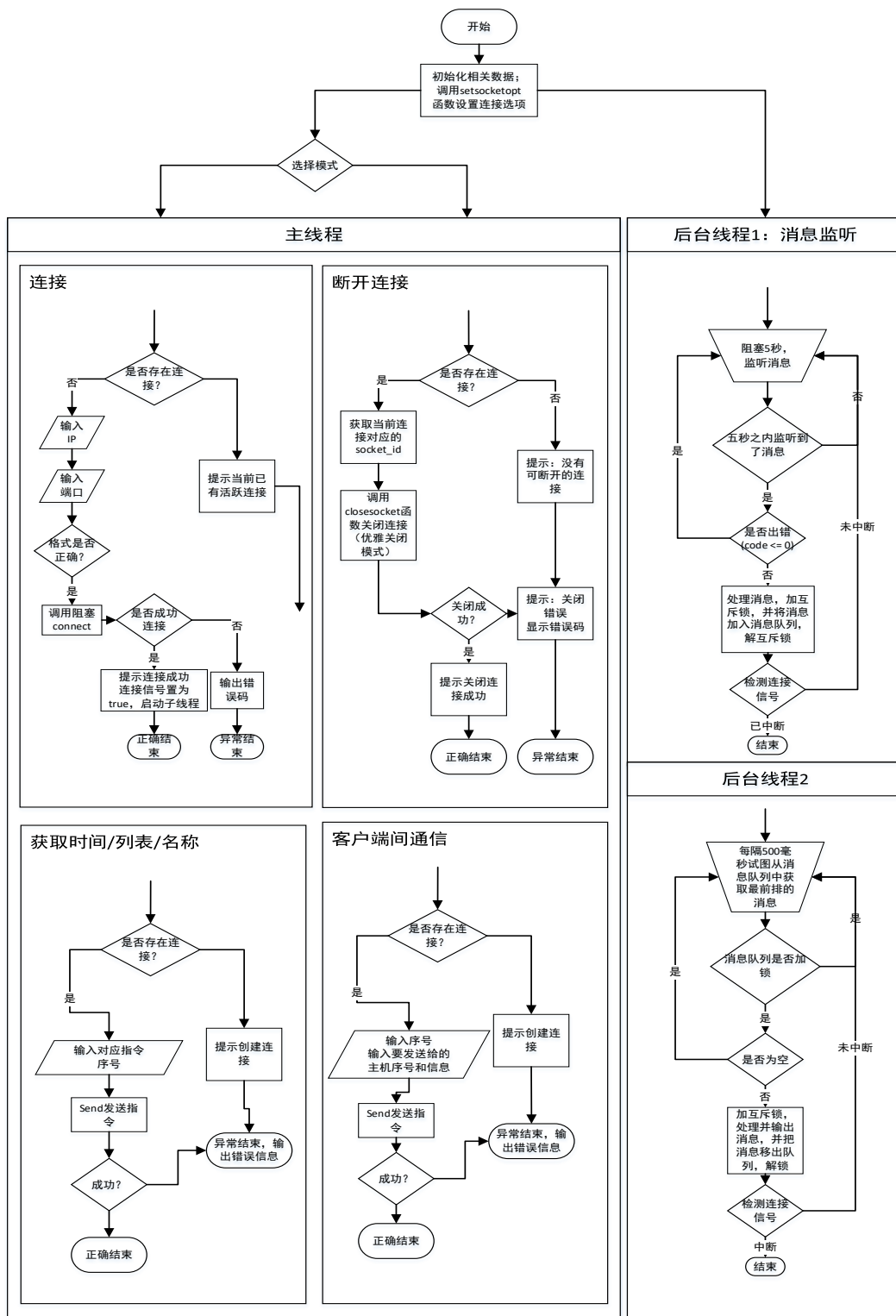
- 小组分工：1 人负责编写服务端，1 人负责编写客户端
- 客户端编写步骤（**需要采用多线程模式**）
  - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
  - b) 编写一个菜单功能，列出 7 个选项
  - c) 等待用户选择
  - d) 根据用户选择，做出相应的动作（未连接时，只能选连接功能和退出功能）
    1. 选择连接功能：请用户输入服务器 IP 和端口，然后调用 `connect()`，等待返回结果并打印。连接成功后设置连接状态为已连接。**然后创建一个接收数据的子线程，循环调用 `receive()`，直至收到主线程通知退出。**
    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：调用 `send()`将获取时间请求发送给服务器，**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：调用 `send()`将获取名字请求发送给服务器，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：调用 `send()`将获取客户端列表信息请求发送给服务器，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，然后调用 `send()`将数据发送给服务器，观察另外一个客户端是否收到数据。
    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，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，并调用 `send()`发给客户端
      2. 请求类型为获取名字：调用 `GetComputerName` 获取本机名，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、端口等数据通过调用 `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，将要转发的消息组装通过调用 `send()`发给接收客户端（使用接收客户端的 `socket` 句柄）。

- 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

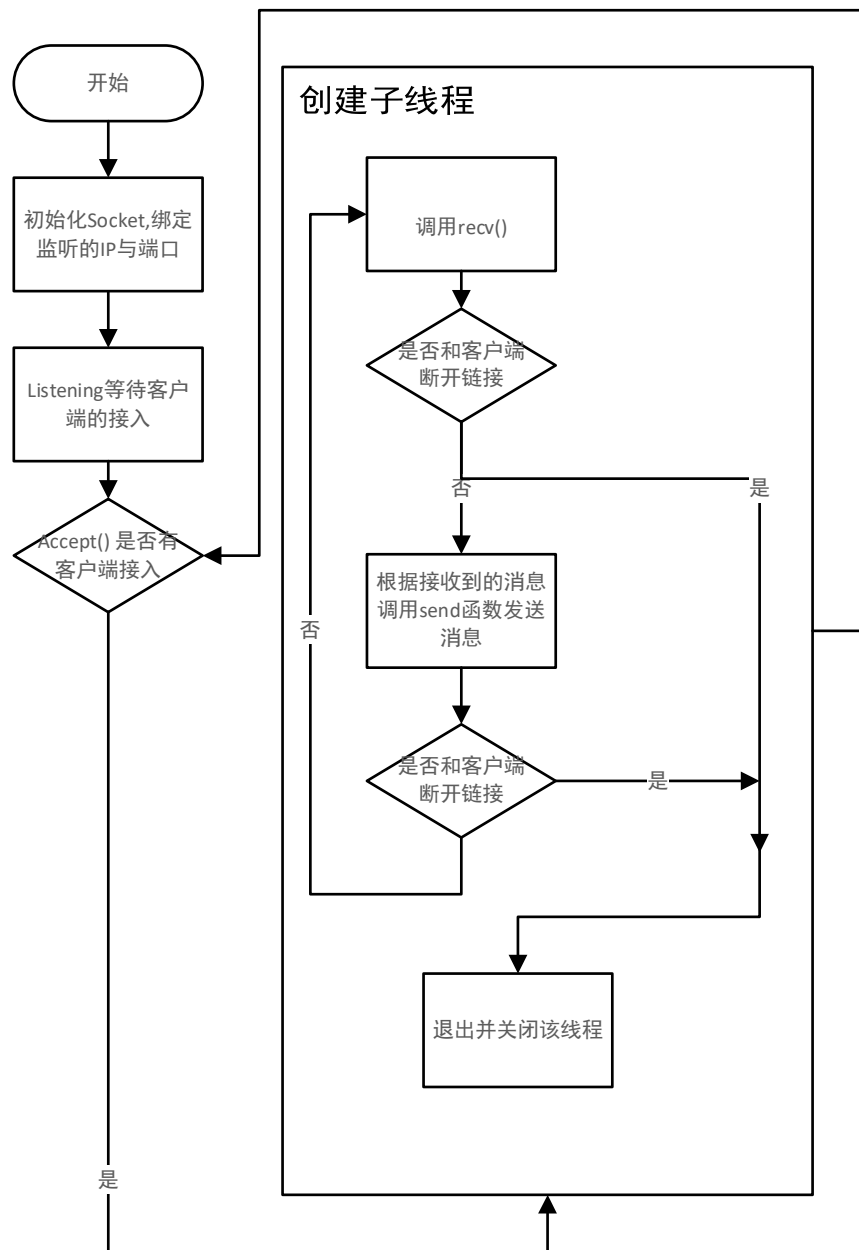
## 五、实验数据记录和处理

- 客户端和服务端框架图（用流程图表示）

■ 客户端总体框架（由沈子衿负责）



## ■ 服务端总体框架（由林宇翔负责）



## ● 客户端初始运行后显示的菜单选项

```

connect successfully. input 'q' to quit, 'help' for instructions
1. connect
2. disconnect
3. get server name
4. get client list
5. get current time
6. send info to client

localhost/>

```

其中输入 'q' 可以退出客户端，输入 'help' 可以获取帮助。

- 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）

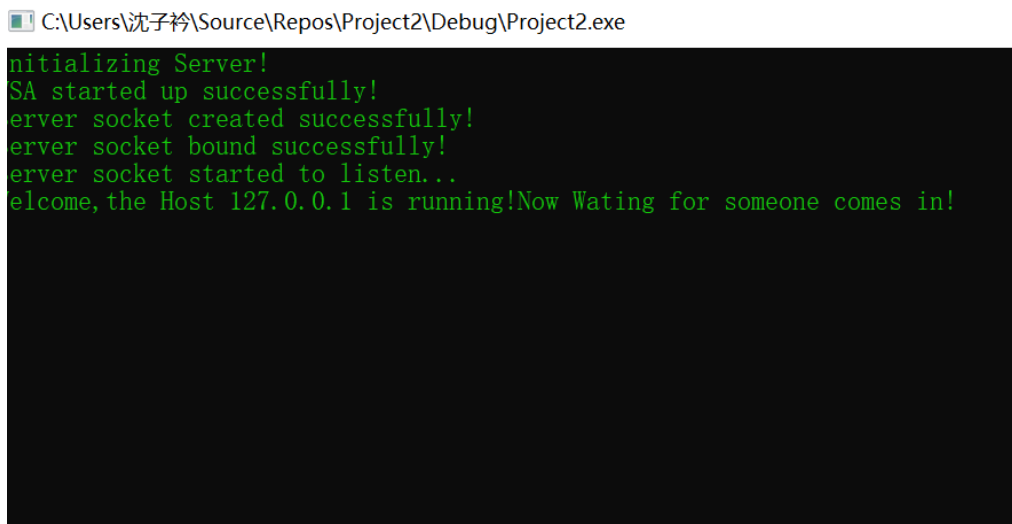
```
void ClientNet::ProcessQueue(char* listenBuffer, bool *isConnect) {
    while (1) {
        // 安全地控制线程的中止
        int iErrMsg = 0;
        iErrMsg = recv(m_sock, listenBuffer, 4096, 0);
        // 出错但连接正常，往往是因为超时
        if (iErrMsg < 0 && (errno == EINTR || errno == EWOULDBLOCK || errno == EAGAIN)) continue;
        else if (iErrMsg < 0) {
            // 未连接
            continue;
        }
        // 连接途中网络突然中断
        else if (iErrMsg == 0) { cout << "error: Network Suddenly Disconnected." << endl; continue; };
        qutex.lock();
        infoQueue.Enqueue(listenBuffer);
        qutex.unlock();
    }
}
```

首先是消息队列处理功能。线程调用有 timeout 设定的阻塞 recv 函数，监听是否有来自服务器的消息，如果有消息且非错误信息，结束阻塞，给队列互斥锁加锁，将新消息加入消息队列，然后结束队列互斥锁的加锁。

```
void ClientNet::ClientListenStart(bool *isConnect) {
    while (1) {
        // 安全地控制线程的中止
        string res;
        if (qutex.try_lock() == false) {
            Sleep(500);
            continue;
        }
        if ((res = infoQueue.Dequeue()) == "") {
            qutex.unlock();
            Sleep(500);
            continue;
        }
        cout << "\nSERVER/> ";
        cout << res.c_str() << endl << endl;
        qutex.unlock();
        if (real_test_lock() == false) {
            iotex.unlock();
        }
        // 每500毫秒中读取一次数据，确保receive要比他的频率快
        else
            cout << "\nlocalhost/> ";
        Sleep(500);
        // 使用之后清空
    }
}
```

然后是消息回显功能。子线程每隔一定时间尝试对消息队列进行 `dequeue` 操作，这一尝试包含当前队列是否为空以及互斥锁是否打开。如果成功，则打印出队节点中包含的信息，否则，进行下一次轮询。每次轮询之间间隔 500ms，比监听线程频率慢，最大限度防止消息丢失。

- 服务器初始运行后显示的界面



```
C:\Users\沈子衿\Source\Repos\Project2\Debug\Project2.exe
nitializing Server!
SA started up successfully!
erver socket created successfully!
erver socket bound successfully!
erver socket started to listen...
elcome,the Host 127.0.0.1 is running!Now Wating for someone comes in!
```

- 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）

```
while (true)
{
    socketClient = accept(socketServer, (SOCKADDR *)&addrClient, &addrLength);
    if (socketClient == INVALID_SOCKET)
    {
        printf("Accept Failed\n");
        continue;
    }
    printf("Accept Success\n");
    InetNtopW(addrClient.sin_family, &addrClient, bufferForIP, IP_SIZE);
    cout << "A new client connected! The IP address: " << inet_ntoa(addrClient.sin_addr) << ",
    port number: " << ntohs(addrClient.sin_port) << endl;
    addNode(Head, socketClient, inet_ntoa(addrClient.sin_addr), ntohs(addrClient.sin_port));
    hThread = CreateThread(NULL, 0, CreateClientThread, (LPVOID)socketClient, 0, NULL);
    if (hThread == NULL)
    {
        cerr << "Failed to create a new thread!Error code: " << ::WSAGetLastError() << endl;
        WSACleanup();
        system("pause");
        exit(1);
    }
    CloseHandle(hThread);
}
return;
```

首先调用 `Accept()` 函数连接子线程，若失败则阻塞当前程序，若连接成功，首先输出当前连接客户端的 IP 地址与监听端口，然后创建子线程。并进行下一轮循环。

- 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：

```
connect successfully. input 'q' to quit, 'help' for instructions

1. connect
2. disconnect
3. get server name
4. get client list
5. get current time
6. send info to client

localhost/> 1
IP: 127.0.0.1
setsockopt(2): No error
Connection success!

localhost/>
```

服务端:

```
Initializing Server!
WSA started up successfully!
Server socket created successfully!
Server socket bound successfully!
Server socket started to listen...
Welcome, the Host 127.0.0.1 is running! Now Waiting for someone comes in!
Accept Success
A new client connected! The IP address: 127.0.0.1, port number: 10182
The Client has added to the list
```

- 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:



```
connect successfully. input `q` to quit, `help` for instructions

1. connect
2. disconnect
3. get server name
4. get client list
5. get current time
6. send info to client

localhost/> 1
IP: 127.0.0.1
setsockopt(2): No error
Connection success!

localhost/> 5

SERVER/> 2018-10-27 13-41-17
```

服务端:

```
Initializing Server!
WSA started up successfully!
Server socket created successfully!
Server socket bound successfully!
Server socket started to listen...
Welcome, the Host 127.0.0.1 is running! Now Waiting for someone comes in!
Accept Success
A new client connected! The IP address: 127.0.0.1, port number: 10182
The Client has added to the list
Message received: time
Client requests to get the time
```

可以看到服务端显示: Client requests to get the name.

- 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:

```
localhost/> 3

SERVER/> The name of the computer: LAPTOP-I210TIEM
```

服务端:

```
Message received: GetComputerName
The name of the computer: LAPTOP-I210TIEM
The name of the computer: LAPTOP-I210TIEM
```

相关的服务器的处理代码片段:

```

else if (strstr(bufMessage, "GetComputerName") != NULL)
{
    cout << "Message received: " << bufMessage << endl;
    memset(bufMessage, 0, MaxSize);
    char host[256];
    if (gethostname(host, sizeof(host)) == SOCKET_ERROR)
    {
        cout << "无法获取主机名..." << endl;
        break;
    }
    else
    {
        cout << "The name of the computer: " << host << endl;
        const char* str = "The name of the computer: ";
        strcpy(bufMessage, str);
        strcat(bufMessage, host);
        cout << bufMessage << endl;
    }
}
}

```

- 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：

```

localhost/> 4
SERVER/> 1.bmp 127.0.0.1 10703.bmp

```

服务端：

```

The name of the computer: 127.0.0.1
Message received: GetClientList
1.bmp 127.0.0.1 10703.bmp

```

可以看出，此时只有一个客户端连接了。如果我们打开两个客户端，都连接到服务器之后再执行获取列表的操作，则可以看到如下的结果

```

localhost/> 1
IP: 127.0.0.1
Connection success!

localhost/> 4

SERVER/> 1.bmp 127.0.0.1 10703.bmp
2.bmp 127.0.0.1 11361.bmp

```

这证明服务器可以同时接受两个以上客户端的连接。

相关的服务器的处理代码片段：

调用代码

```

else if (strstr(bufMessage, "GetClientList") != NULL)
{
    cout << "Message received: " << bufMessage << endl;
    memset(bufMessage, 0, MaxSize);
    getTheList(Head, bufMessage);
    cout << bufMessage << endl;
}

```

函数代码

```

int getTheList(Clit Head, char * Message)
{
    Clit ptr = Head->Next;
    char Current[10];
    while (ptr != NULL)
    {
        memset(Current, 0, 10);
        sprintf_s(Current, "%d.bmp", ptr->Num);
        strcat(Message, Current);
        strcat(Message, " ");
        strcat(Message, ptr->IP);
        strcat(Message, " ");
        memset(Current, 0, 10);
        sprintf(Current, "%d.bmp", (int)ptr->Port);
        strcat(Message, Current);
        strcat(Message, "\n");
        ptr = ptr->Next;
    }
    return 0;
}

```

getTheList 函数在客户端链表里读取各个客户端的编号 IP, Port 并将其组织成有序的字符串, copy 进 Buffer 里。

- 客户端选择发送消息功能时, 两个客户端和服务端 (如果有的话) 显示内容截图。

发送消息的客户端:

```

localhost/> 6
Please select the client No. : 2
Please enter the info: dasfa
SERVER/> Your Message send success

```

服务器端（可选）：

```
Message received: Send 2 dasfa
The Message send: The Message From Client 1 : dasfa
Your Message send success
Send Success
```

接收消息的客户端：

```
localhost/>
SERUER/> The Message From Client 1 : dasfa
```

相关的服务器的处理代码片段：

```
else if (strstr(bufMessage, "Send") != NULL)
{
    cout << "Message received: " << bufMessage << endl;
    int n = bufMessage[5] - '0';
    if (!(1 <= n && n <= 9))
    {
        memset(bufMessage, 0, MaxSize);
        strcpy(bufMessage, "The Number is Error");
        cout << bufMessage << endl;
    }
    else
    {
        Mark = getSock(Head, n);
        if (Mark == INVALID_SOCKET)
        {
            memset(bufMessage, 0, MaxSize);
            strcpy(bufMessage, "The Number is Error,Please check the
client list again");
            cout << bufMessage << endl;
        }
        else
        {
            char * Message = &(bufMessage[7]);
            memset(AllMessage, 0, MaxSize);
            strcpy(AllMessage, "The Message to Client ");
```

```

strcpy(AllMessage, "The Message to Client ");
char a[5];
a[0] = (int)5 ;
a[1] = ,
a[2] = ':';
a[3] = ' ';
a[4] = '\0';
strcat(AllMessage, a);
strncat(AllMessage, Message, MaxSize - 28);
cout << "The Message send: " << AllMessage << endl;
sendResult = send(Mark, AllMessage, MaxSize, 0);
if (sendResult == SOCKET_ERROR)
{
    memset(bufMessage, 0, MaxSize);
    strcpy(bufMessage, "Your Message send Failed,please Check
the Clinet list again");
    cout << bufMessage << endl;
}
else
{
    memset(bufMessage, 0, MaxSize);
    strcpy(bufMessage, "Your Message send success");
    cout << bufMessage << endl;
}

```

相关的客户端（发送和接收消息）处理代码片段：

发送消息

```

/*客户端发送消息*/
int ClientNet::ClientSend(const char* msg, int len)
{
    int rlt = 0;

    int iErrMsg = 0;

    // 指定sock发送消息
    iErrMsg = send(m_sock, msg, len, 0);
    if (iErrMsg < 0)
        // 发送失败
    {
        printf("send msg failed with error: %d\n", iErrMsg);
        rlt = 1;
        return rlt;
    }
    else if (iErrMsg == 0)
    {
        rlt = 3;
        printf("connection timeout.\n");
        return rlt;
    }
    //printf("send msg successfully\n");
    iotex.lock();
    return rlt;
}

```

接收消息

```
void ClientNet::ProcessQueue(char* listenBuffer, bool *isConnect) {
    while (1) {
        // 安全地控制线程的中止
        int iErrMsg = 0;
        iErrMsg = recv(m_sock, listenBuffer, 4096, 0);
        // 出错但连接正常, 往往是因为超时
        if (iErrMsg < 0 && (errno == EINTR || errno == EWOULDBLOCK ||
            errno == EAGAIN)) continue;
        else if (iErrMsg < 0) {
            // 未连接
            continue;
        }
        // 连接途中网络突然中断
        else if (iErrMsg == 0) { cout << "error: Network Suddenly
            Disconnected." << endl; continue; };
        qutex.lock();
        infoQueue.Enqueue(listenBuffer);
        qutex.unlock();
    }
}
```

## 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？

答 1：不需要。操作系统会根据服务器的 IP 自动选择符合条件的网络连接，而不必由客户端指定；

答 2：我们首先观测客户端 2 的端口状态：

```
Message received: GetClientList
1 127.0.0.1 9564
2 127.0.0.1 9785
```

发现当前使用端口是 9785。

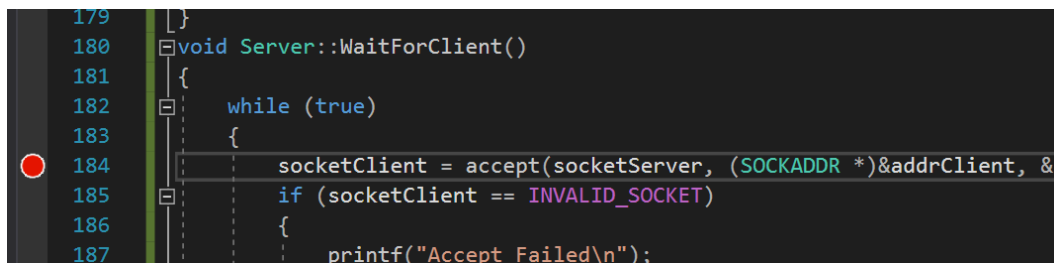
然后我们断开客户端 2 和服务器的连接，再执行重连，查看端口状态如下：

```
Message received: GetClientList
1 127.0.0.1 9564
2 127.0.0.1 9792
```

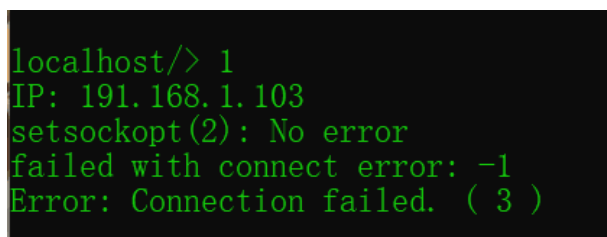
发现当前使用端口变化为 9792。这一结果表明，connect 时客户端的端口并不是不变的，而是动态分配的。操作系统会根据当前端口的占用情况和服务器连接情况选择最优端口提供给客户端。客户端也可以自己指定端口，但不是必需的。

- 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？

答：首先尝试设置断点：



设置断点后。客户端尝试连接结果如下：



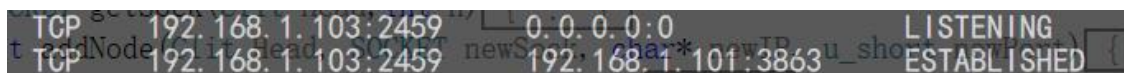
可知连接会因超时而失败。

- 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？

答：Send()函数一个很重要的参数 是 SOCKET 函数，每个 SOCKET 对应不同的客户端，可以通过接收的套接字（SOCKET 句柄）来区分数据包是属于哪个客户端的。

- 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）

答：连接之后，TCP 状态如下，为‘ESTABLISHED’。



TCP 三次握手客户端 `connect`，会发送 SYN，服务器端确认并发送 SYN，客户端确认，进入 ESTABLISHED 状态，客户端调用 `close`，发送 FIN，进入 FIN\_WAIT1 状态，服务器端确认，进入 CLOSE\_WAIT 状态，客户端进入 FIN\_WAIT2 状态。

- 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？

答 1：当客户端与服务器建立起正常的 TCP 连接后，如果客户主机网线断开、电源掉电、或系统崩溃，服务器进程将永远不会知道（通过我们常用的 `select`，`epoll` 监测不到断开或错误事件），如果不主动处理或重启系统的话对于服务端来说会一直维持着这个连接，任凭服务端进程如何望穿秋水，也永远再等不到客户端的任何回应。这种情况就是半开连接，浪费了服务器端可用的文件描述符。

答 2：配置操作系统的 `SO_KEEPALIVE` 选项，或者进行应用层心跳检测。

## 七、 讨论、心得

本次实验主要涉及 WinSocket API 和 C++多线程编程相关知识。通过本次实验，我学习了 WinSocket 的基本概念和用法，也学习了 Linux `pthread` 库和 C++ 11 跨平台线程库的用法，实现了计算机网络课程知识同操作系统课程知识的融会贯通，可谓受益匪浅。

本次实验是小组实验和探究型实验，代码量较大，且需要模块之间的对接，这无疑锻炼了我们在编程工作中的团队协作能力、自学能力信息获取能力，对未来开发类似项目有较大的参考价值。