

# 浙江大学

## 本科实验报告

课程名称:	计算机网络
实验名称:	基于 Socket 接口实现自定义协议通信
姓 名:	张溢弛
学 院:	计算机学院
系:	软件工程
专 业:	软件工程
学 号:	3180103772
指导教师:	邱劲松

2020 年 12 月 26 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: 聂俊哲 3180103501 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（画图或者绘制表格说明），请求类型的定义

请求类型	关键字
获取当前时间	<code>GetTime</code>
获取服务器的机器名字	<code>GetName</code>
获取当前所有用户信息	<code>GetList</code>
给另一个用户发消息	<code>Send</code>

前三个数据包的格式内容比较简单，我主要说明一下第四个数据包的格式

send	空格	用户 编号	空格	message
------	----	----------	----	---------

- 描述响应数据包的格式（画图说明），响应类型的定义

请求类型	关键字
返回当前时间	GetTime
返回服务器的机器名字	GetName
返回当前所有用户信息	GetList
信息发送是否成功	Send

GetTime 和 Send 的格式内容都比较简单，重点描述第二第三个数据包。

返回服务器的机器名字：

The name of the computer:	空格	主机名
---------------------------	----	-----

- 描述指示数据包的格式（画图说明），指示类型的定义

The Message From Client	空格	发送客户端 编号	:	空格	发送的消息
----------------------------	----	-------------	---	----	-------

- 客户端初始运行后显示的菜单选项

```
C:\Users\74096\Desktop\计算机网络\Client\Debug\Client.exe
Welcome to the DIY Client!
Author: Zhang Each & Nie Junzhe

You can choose the following operations:

1. Connect to the server
2. Close the server connect
3. Get the time
4. Get the server name
5. Get the client list
6. Send a message to server
7. Quit
```

- 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) {
    while (!client.testLock());
    cout << "LocalHost>>>";
    cin >> choice;
    // 按照输入的选择来进行不同的操作
    if (choice == 1) {
        if (isConnect) {
            cout << "Error! You have been connect to the server!\n" << endl;
        }
        // 没有连接到服务端的时候尝试连接到服务端
        else { ... }
    }
    else if (choice == 2) { ... }
    // 以下三种是向服务器获取数据的方式
    else if (choice == 4) { ... }
    else if (choice == 5) { ... }
    else if (choice == 3) { ... }
    // 客户端间的通信
    else if (choice == 6) { ... }
    else if (choice == 7) { ... }
    else {
        cout << "Error: No such operation\n" << endl;
    }
}
```

客户端的主线程代码主要是用一个无限循环来进行请求的发送，每次循环将会处理一条请求，用户根据上面的提示选择对应的操作，比如建立连接，断开连接，客户端之间通信等等，然后根据对应的选项，按照程序的提示输入关键的数据和信息即可完成操作，当输入7的时候就会退出，并结束客户端的运行

- 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::queueProcess(char* infomation, bool& isConnect) {
    while (true) {
        int err = 0;
        err = recv(sock, infomation, 4096, 0);
        if (err < 0) {
            continue;
        }
        else if (err == 0) {
            cout << "Network Interrupted!" << endl;
        }
        queue_mutex.lock();
        this->info.Enqueue(infomation);
        queue_mutex.unlock();
    }
}
```


队列处理子线程会将收到的消息存放到消息队列中

```
void Client::clientListenStart(bool& isConnect) {
    while (true) {
        string res;
        // 尝试上锁失败
        if (!queue_mutex.try_lock()) {
            Sleep(500);
            continue;
        }
        res = info.Dequeue();
        // 消息队列是空的
        if (res == "") {
            queue_mutex.unlock();
            Sleep(500);
            continue;
        }
        cout << "\nServer>>>" << res << endl;
        queue_mutex.unlock();
        if (!testLock()) {
            io_mutex.unlock();
        }
        else {
            cout << "\nLocalHost>>>" << endl;
        }
        // 每次检查锁的时候检查完休息500ms, 不让这个子线程一直运行跑炸了
        Sleep(500);
    }
}
```

客户端的监听需要单独开设一个子线程来运行，在一个无限循环中每隔 500ms 就检测一次消息队列是否出现了新的消息，如果收到了新的消息就将其打印出来，我们在这个循环中使用了

一些锁来保证同步，防止死锁

- 服务器初始运行后显示的界面

 C:\Users\15888\source\repos\ConsoleApplication2\x64\Debug\ConsoleApplication2.exe

```
initializing Server!
WSA started up successfully!
Server socket created successfully!
Server socket bound successfully!
Server socket started to listen...
Welcome,the Host 192.168.43.78 is running!Now Wating for someone comes in!
```

- 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)
{
    //调用Accept函数，若链接到客户端则继续，否则阻塞该程序
    socketClient = accept(socketServer, (SOCKADDR *)&addrClient, &addrLength);
    //连接失败
    if (socketClient == INVALID_SOCKET)
    {
        printf("Accept Failed\n");
        continue;
    }
    printf("Accept Success\n");
    //转换网络二进制结构到ASCII类型的地址
    InetNtopW(addrClient.sin_family, &addrClient, bufferForIP, IP_SIZE);
    //输出客户端的IP和监听端口
    cout << "A new client connected! The IP address: " << inet_ntoa(addrClient.sin_addr) << ", port number: " << ntohs(addrClient.sin_port) << endl;
    //将客户端加入到客户端链表中
    addNode(Head, socketClient, inet_ntoa(addrClient.sin_addr), ntohs(addrClient.sin_port));
    //创建子线程
    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;
```

循环保持监听，调用 Accept 函数，若链接到客户端则继续，否则阻塞该程序。若连接成功，先转换网络二进制结构到 ASCII 类型的地址，再将客户端加入到客户端链表中并创建子线程，之后进入下一次循环保持持续监听。

- 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）

-



```

//连上客户端，发送HELLO
memset(bufMessage, 0, MaxSize);
strcpy(bufMessage, "Hello!\n");
sendResult = send(socketClient, bufMessage, MaxSize, 0);
do
{
    memset(bufMessage, 0, MaxSize);
    retValue = recv(socketClient, bufMessage, MaxSize, 0);
    if (retValue > 0)
    {
        if (strcmp(bufMessage, "exit") == 0) { ... }
        else if (strstr(bufMessage, "GetTime") != NULL) { ... }
        else if (strstr(bufMessage, "GetName") != NULL) { ... }
        else if (strstr(bufMessage, "GetList") != NULL) { ... }
        else if (strstr(bufMessage, "Send") != NULL) { ... }
        else { ... }
        if(temp==0)
            sendResult = send(socketClient, bufMessage, MaxSize, 0);
        else { ... }

        if (sendResult == SOCKET_ERROR)
        {
            cerr << "Failed to send message to client!Error code: " << ::GetLastError() << endl;
            ::closesocket(socketClient);
            system("pause");
            return 1;
        }
    }
} while (retValue > 0);
}

```

服务端的客户端处理代码主要是用循环来处理客户端的请求，retValue 是判断变量，用来判断接收与发送的操作是否完成。只有当 retValue > 0 时说明成功接收客户端的请求，retValue == 0 时说明客户端请求断开连接，retValue < 0 说明发生错误。每次循环将会处理一条请求，比如建立连接，断开连接，客户端之间通信等等，然后根据对应的选项，执行相关代码。

- 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

```

LocalHost>>>1
Please input the IP address:
10.181.149.240
setsocket:: No error

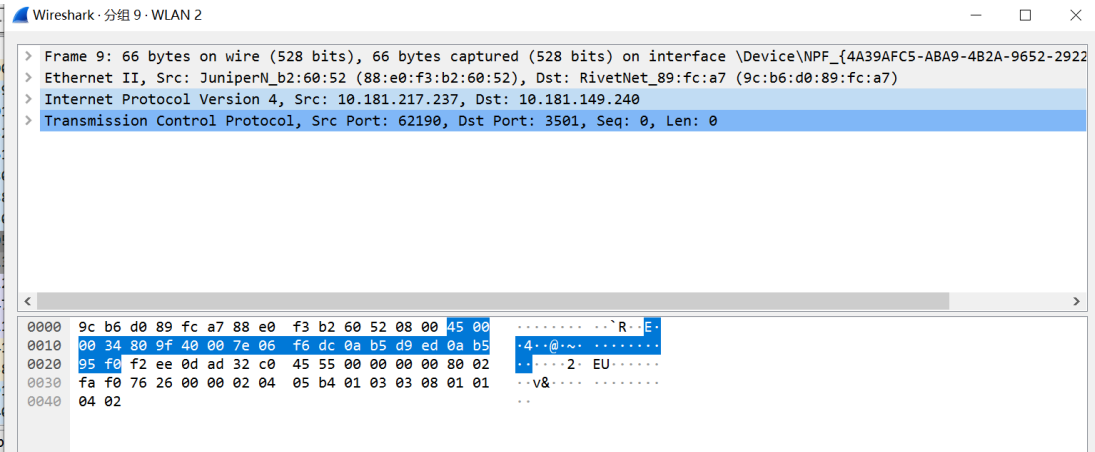
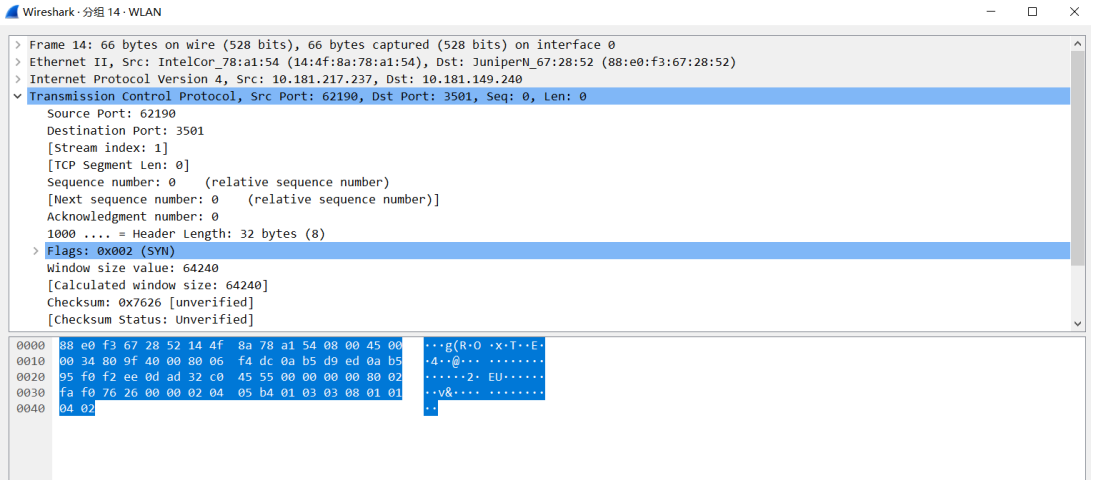
Success to connect!
LocalHost>>>
Server>>>Hello!

```

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 10.181.149.240 is running!Now Wating for someone comes in!
Accept Success
A new client connected! The IP address: 10.181.217.237, port number: 62230
The Client has added to the list
-
```

Wireshark 抓取的数据包截图：



9	4.403053	10.181.217.237	10.181.149.240	TCP	66	62190 → 3501 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
10	4.403239	10.181.149.240	10.181.217.237	TCP	66	3501 → 62190 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
11	4.409122	10.181.217.237	10.181.149.240	TCP	56	62190 → 3501 [ACK] Seq=1 Ack=1 Win=131328 Len=0
12	4.772479	10.181.149.240	40.90.189.152	TCP	55	61553 → 443 [ACK] Seq=1 Ack=1 Win=514 Len=1 [TCP segment of a reassembled...
13	5.138218	40.90.189.152	10.181.149.240	TCP	66	443 → 61553 [ACK] Seq=1 Ack=2 Win=7971 Len=0 SLE=1 SRE=2
14	5.219413	JuniperN_67:28:52	Broadcast	ARP	60	Who has 10.181.251.144? Tell 10.0.2.2
15	6.139588	JuniperN_67:28:52	Broadcast	ARP	60	Who has 10.181.208.78? Tell 10.0.2.2

- 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

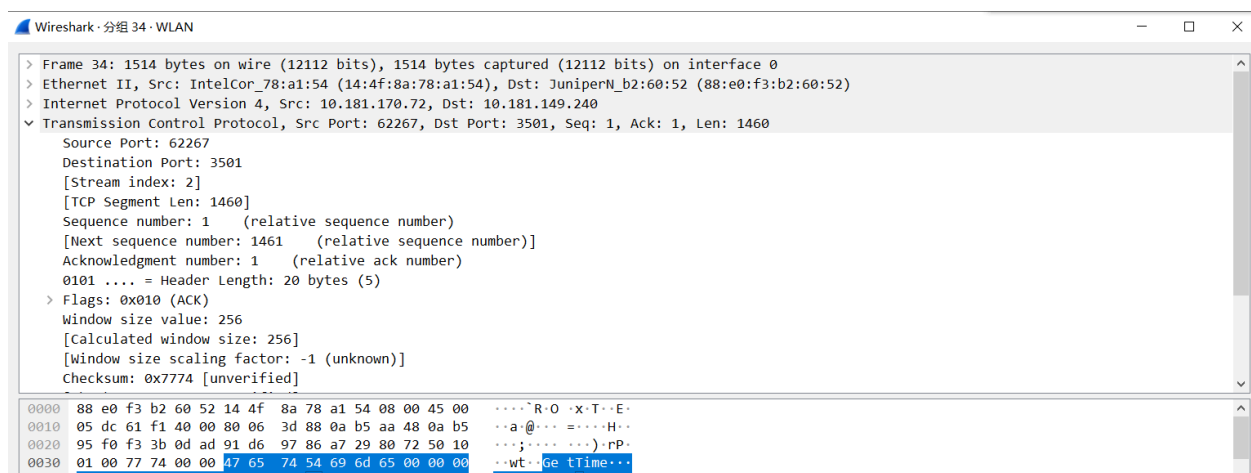
```
LocalHost>>>
3
Server>>>2020-12-16 16-52-13
```

服务端

```
Message received: GetTime
Client requests to get the time
Send Success
```

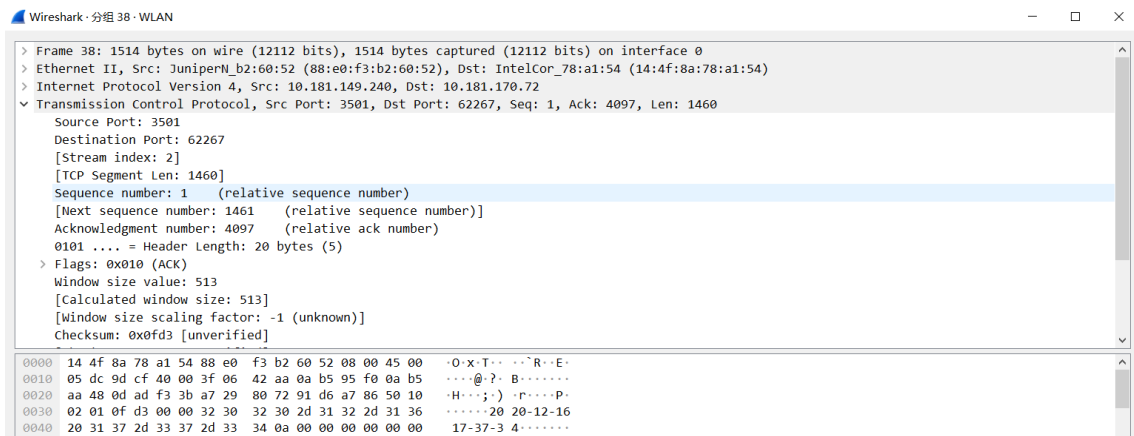
Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的时间数据对应的位置）：

客户端的抓包图，成功抓取了有 GetTime 信息的 TCP 包

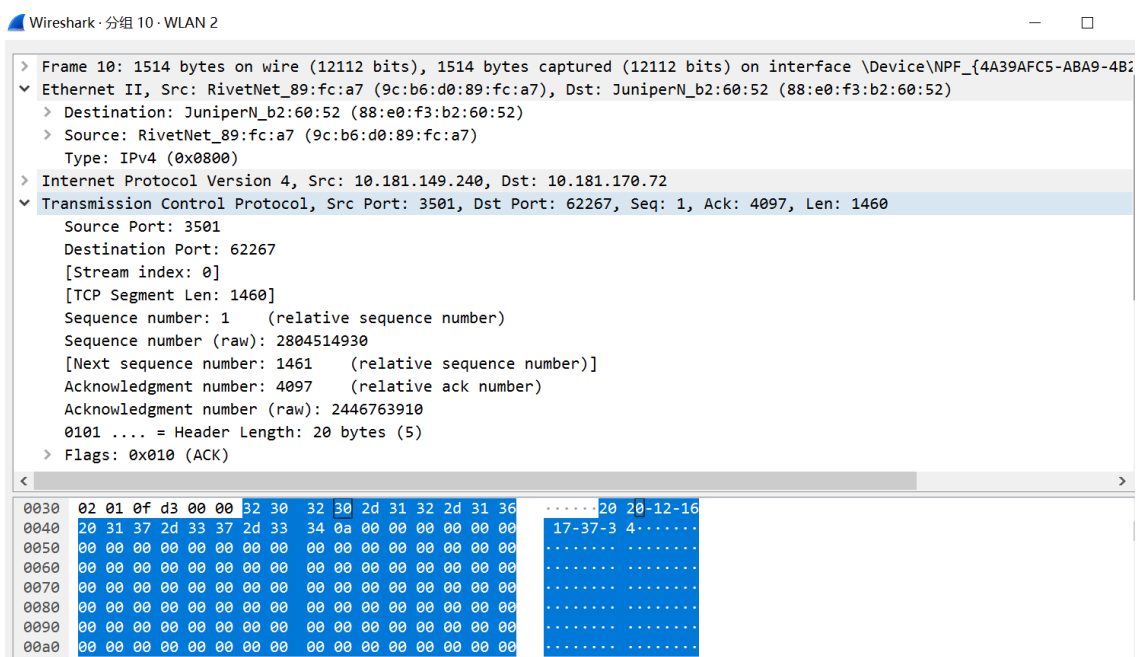


客户端抓包截图

抓到服务端发送回来的包，含有时间信息



服务端的抓包截图



- 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

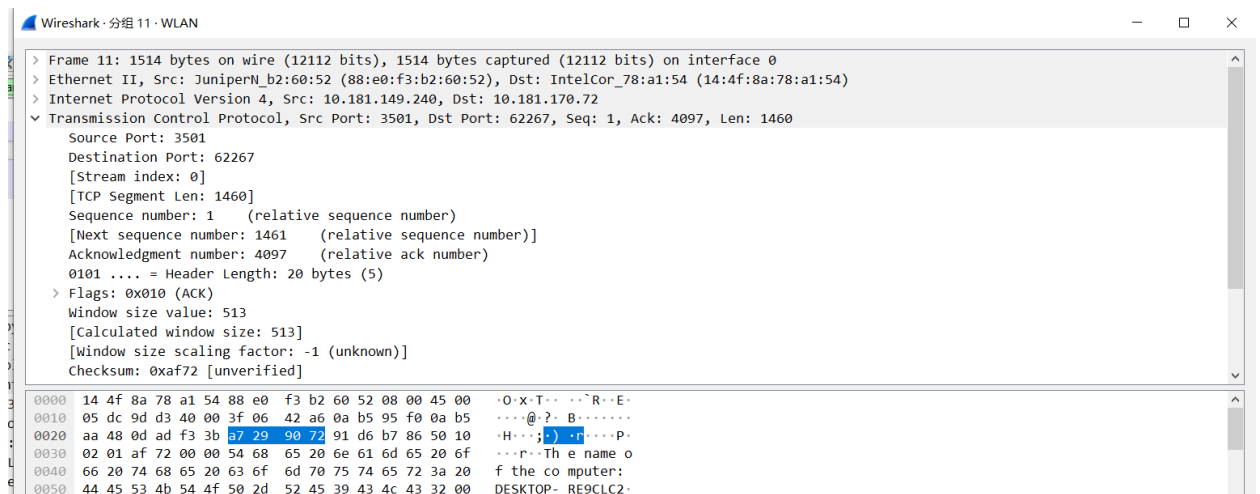
```
LocalHost>>>>4
Server>>>>The name of the computer: DESKTOP-RE9CLC2
```

服务端

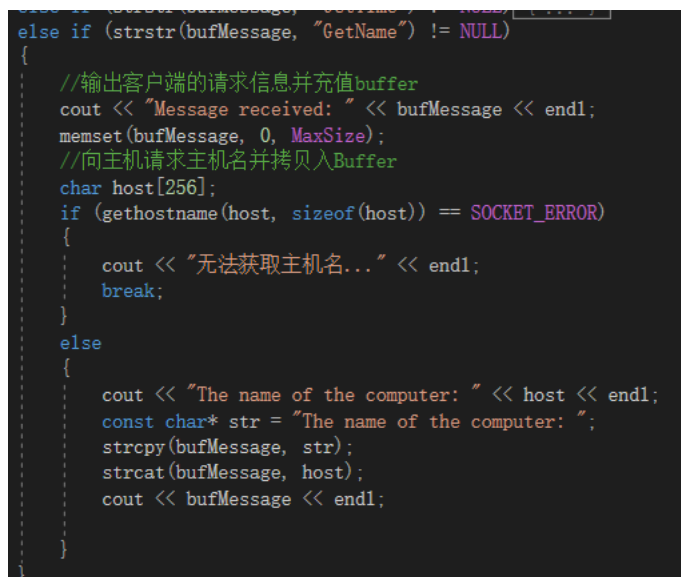
```
Message received: GetName
The name of the computer: DESKTOP-RE9CLC2
The name of the computer: DESKTOP-RE9CLC2
Send Success
```

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):

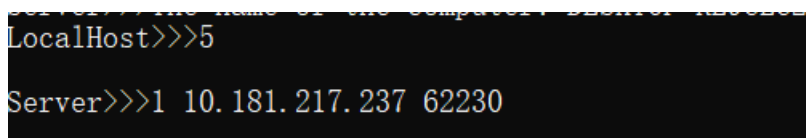


相关的服务器的处理代码片段:



- 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: 当前只连接了我一台客户端

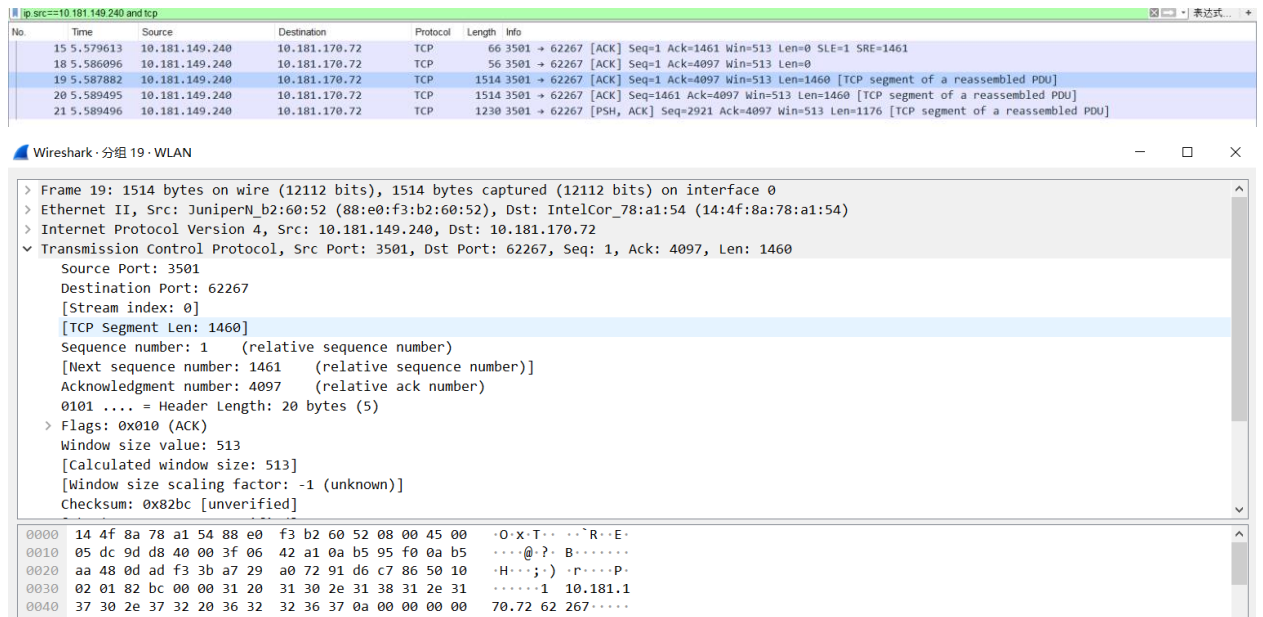


服务端:

```
Message received: GetList
1 10.181.217.237 62230

Send Success
```

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：

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

```

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

```

- 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：

```

LocalHost>>>
5
Server>>>1 10.181.170.72 62431
2 10.181.170.72 62432

LocalHost>>>6
Please input the client number:
1
Please input your message:
hello123

Server>>>Your Message send success

```

服务器：

```

Message received: GetList
1 10.181.170.72 62431
2 10.181.170.72 62432

Send Success
Message received: Send 1 hello123
The Message send: The Message From Client 2 : hello123
Your Message send success
Send Success

```

接收消息的客户端：

```

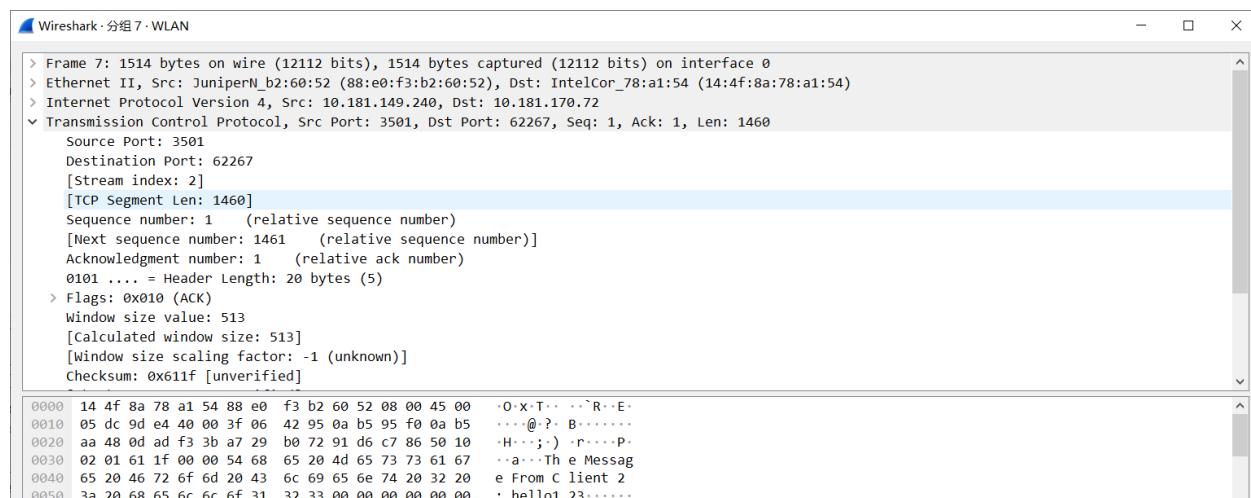
LocalHost>>>
5

Server>>>1 10.181.170.72 62431

LocalHost>>>_
Server>>>The Message From Client 2 : hello123

```

Wireshark 抓取的数据包截图（发送和接收分别标记）：



相关的服务器的处理代码片段：



```

else if (strstr(bufMessage, "Send") != NULL)
{
    //收到send 命令，先打印接收到的消息
    cout << "Message received: " << bufMessage << endl;
    //得到接收客户端的编号并提取 Socket
    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
        {
            //编辑发送的信息
            char * Message = &(bufMessage[7]);
            memset(AllMessage, 0, MaxSize);
            strcpy(AllMessage, "The Message From Client ");
            char a[5];
            //得到发送客户端的编号
            n = getNum(Head, socketClient);
            a[0] = n + '0';
            a[1] = ' ';
            a[2] = ':';
            a[3] = ' ';
            a[4] = '\0';
            strcat(AllMessage, a);
            //将信息拷贝入ALLMessafe 发送Buffer

```

```

            strcat(AllMessage, a);
            //将信息拷贝入ALLMessafe 发送Buffer
            strncat(AllMessage, Message, MaxSize - 28);
            cout << "The Message send: " << AllMessage << endl;
            //发送消息
            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
            {
                //发送成功，给发送的客户端发一个发送成功的信息
                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：

```
// 客户端间的通信
else if (choice == 6) {
    string num, info;
    cout << "Please input the client number: \n";
    cin >> num;
    cout << "Please input your message: \n";
    cin >> info;
    string pack = "Send " + num + " " + info;
    strcpy(msg, pack.c_str());
    result = client.clientSend(msg, 4096);
    msg[0] = '\0';
}
```

```
/*
 * 向服务端发送信息，不同的返回值分别代表发送失败，连接超时和成功发送三种情况
 * 调用了socket中的send函数来进行消息的发送
 */
int Client::clientSend(const char* msg, int length) {
    int err = send(sock, msg, length, 0);
    if (err < 0) {
        cout << "Fail to send the message for error " << err << endl;
        return 1;
    }
    else if (err == 0) {
        cout << "Connect time out!" << endl;
        return 3;
    }
    io_mutex.lock();
    return 0;
}
```

接收消息的代码片段就是前面提到的消息队列，每隔一段时间检测消息队列中是否收到了新的消息。

- 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。

wireshark 没有抓到对应的释放连接的包，但是通过一段时间之后服务器会发现连接已经断开并释放线程

- 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？

```
LocalHost>>>1
Please input the IP address:
10.181.149.240
setsocket:: No error

Success to connect!
LocalHost>>>
Server>>>Hello!

LocalHost>>>
5

Server>>>1 10.181.170.72 62198
2 10.181.170.72 62226

LocalHost>>>6
Please input the client number:
1
Please input your message:
hello1
```

客户端发送之后没有收到服务器的响应，服务器发现 1 号客户端已经消失，就断开了连接

- 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。

No.	Time	Source	Destination	Protocol	Length	Info
315	5.744394	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=386133 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
316	5.744394	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=387593 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
317	5.744394	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [PSH, ACK] Seq=389053 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
318	5.744395	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=390513 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
319	5.744395	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [PSH, ACK] Seq=391973 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
320	5.744396	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=393433 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
321	5.744396	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=394893 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
322	5.744396	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [PSH, ACK] Seq=396353 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
323	5.744396	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=397813 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
324	5.744396	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=399273 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
325	5.744397	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [PSH, ACK] Seq=400733 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
327	5.744649	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=402193 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
328	5.744649	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=403653 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
329	5.744651	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [PSH, ACK] Seq=405113 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
330	5.744651	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=406573 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
331	5.744651	10.181.149.240	10.181.170.72	TCP	1514	3501 → 62248 [ACK] Seq=408033 Ack=4097 Win=513 Len=1460 [TCP segment of a reassembled PDU]
332	5.744651	10.181.149.240	10.181.170.72	TCP	162	3501 → 62248 [PSH, ACK] Seq=409493 Ack=4097 Win=513 Len=108 [TCP segment of a reassembled PDU]

Frame 332: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface 0

Ethernet II, Src: JuniperR\_b2:60:52 (88:e0:f3:b2:60:52), Dst: IntelCor\_78:a1:54 (14:4f:8a:78:a1:54)

Internet Protocol Version 4, Src: 10.181.149.240, Dst: 10.181.170.72

Transmission Control Protocol, Src Port: 3501, Dst Port: 62248, Seq: 409493, Ack: 4097, Len: 108

Source Port: 3501

Destination Port: 62248

[Stream Index: 2]

[TCP Segment Len: 108]

Sequence number: 409493 (relative sequence number)

[Next sequence number: 409601 (relative sequence number)]

Acknowledgment number: 4097 (relative ack number)

0101 .... = Header Length: 20 bytes (5)

Flags: 0x01B (PSH, ACK)

Window size value: 513

[calculated window size: 513]

[Window size scaling factor: -1 (unknown)]

Checksum: 0xc6a [unverified]

[Checksum Status: Unverified]

分送: 368 已显示: 293 (79.6%) 已选择: 0 (0.0%) Profile: Default

实际收到的数据包不足 100 个

- 多个客户端同时连接服务器，同时发送时间请求（程序内自动连续调用 100 次 send），服务器和客户端的运行截图

修改后的客户端代码

```

else if (choice == 3) {
    strcpy(msg, "GetTime");
    for (int i = 0; i < 100; i++) {
        result = client.clientSend(msg, 4096);
    }

    msg[0] = '\0';
}

```

运行时候的客户端截图，成功收到了 100 次回复

```

Server>>>2020-12-16 17-33-40

Server>>>2020-12-16 17-33-40

Server>>>2020-12-16 17-33-41

Server>>>2020-12-16 17-33-42

Server>>>2020-12-16 17-33-42

Server>>>2020-12-16 17-33-43

Server>>>2020-12-16 17-33-43

Server>>>2020-12-16 17-33-44

Server>>>2020-12-16 17-33-44

Server>>>2020-12-16 17-33-45

```

```
Server>>>2020-12-16 17-34-17  
  
Server>>>2020-12-16 17-34-18  
  
Server>>>2020-12-16 17-34-18  
  
Server>>>2020-12-16 17-34-19  
  
Server>>>2020-12-16 17-34-19  
  
Server>>>2020-12-16 17-34-20  
  
Server>>>2020-12-16 17-34-20  
  
Server>>>2020-12-16 17-34-21  
  
Server>>>2020-12-16 17-34-21  
LocalHost>>>
```

服务器截图

## 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？

不需要进行 bind 操作，系统自动产生一个源端口，每次调用 connect 时客户端的端口可能会发生变化，系统自动分配了一个空闲的端口

- 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，可能会少于 send 的次数

- 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？

socket 是由唯一标记的，send 函数中的 sockfd 字段是每个 socket 的标志

- 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）

状态时 TIME\_WAIT，稍微过一小段事件之后就会消失

- 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？

此时的服务器的该 TCP 连接状态不会发生变化

查阅资料得知，可以在代码中使用 heartbeat 来验证连接是否已经断开，当客户端断开超过一定时间之后服务器就会断开连接，TCP 就会释放

## 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）

本次实验中我和队友学习了 C/C++ 的 socket 编程，掌握了 Windows 环境下 socket API 的基本用法和一些 Windows API 的用法，比如获取当前时间和本机信息，收获比较大。

我认为本次实验的工作量比较大，并且由于客户端和服务端是由两个人分别编写的，因此 debug 的难度比较大，我们预先商量好了各个数据包的格式之后就分头开始写了，但是 debug 似乎花费了比写代码更长的时间，最后的试运行和 wireshark 抓包环节也碰到了比较多的问题，比如数据包格式对不上，线程的调度存在问题等等，但是最后还是完成了整个实验。

这个实验给我的印象比较好，我认为这样的实验比之前的 GNS3 相关实验更具有挑战性和趣味性，同时这样的实验也更像是计算机学院专业课的课程实验。