

**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： | 魏伯翼/王逸君 |
| 学 院： | 计算机学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3180103616/ |
| 指导教师： | 陆系群 |

2020年 11 月 29 日

**因为代码文件太大，上传到github上了，请到github上下载源代码https://github.com/nujiy/Network**

**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： 魏伯翼/王逸君 实验地点： 计算机网络实验室

# 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

# 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：
   1. 连接：请求连接到指定地址和端口的服务端
   2. 断开连接：断开与服务端的连接
   3. 获取时间: 请求服务端给出当前时间
   4. 获取名字：请求服务端给出其机器的名称
   5. 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）
   6. 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
   7. 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：
   1. 向客户端传送服务端所在机器的当前时间
   2. 向客户端传送服务端所在机器的名称
   3. 向客户端传送当前连接的所有客户端信息
   4. 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
   5. 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

# 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。

# 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：

1. 定义两个数据包的边界如何识别
2. 定义数据包的请求、指示、响应类型字段
3. 定义数据包的长度字段或者结尾标记
4. 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**）

1. 运行初始化，调用socket()，向操作系统申请socket句柄
2. 编写一个菜单功能，列出7个选项
3. 等待用户选择
4. 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）
5. 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。**
6. 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。
7. 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。
8. 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。
9. 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）。
10. 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）。
11. 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。
12. 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**）

1. 运行初始化，调用socket()，向操作系统申请socket句柄
2. 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
3. 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。
5. 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。

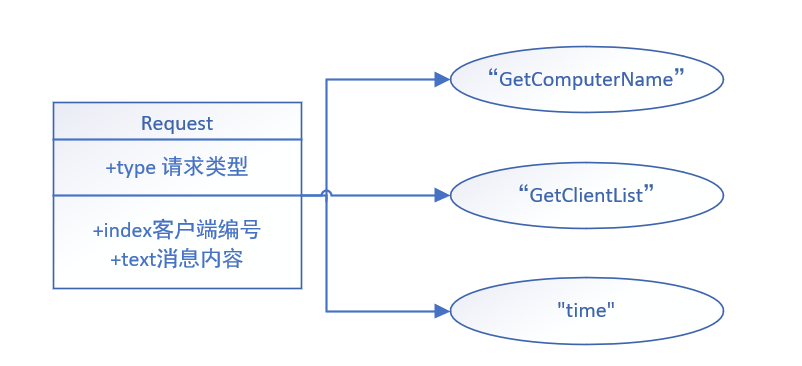
* 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
* 使用多个客户端同时连接服务端，检查并发性
* 使用Wireshark抓取每个功能的交互数据包

# 实验数据记录和处理

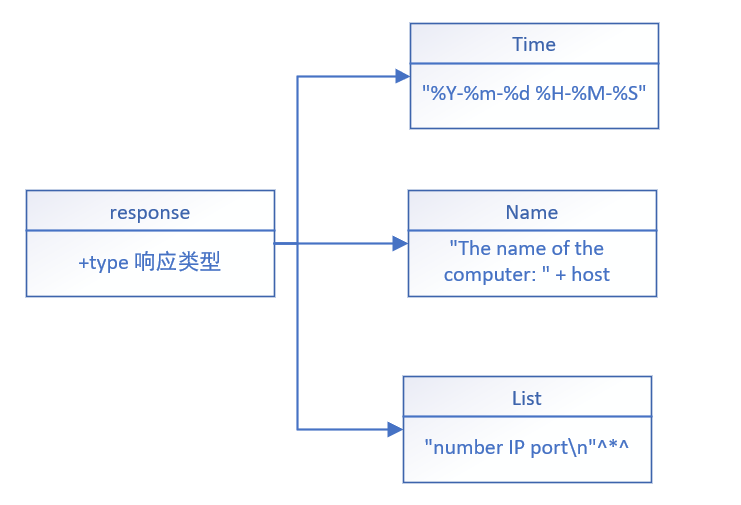
**请将以下内容和本实验报告一起打包成一个压缩文件上传：**

* **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

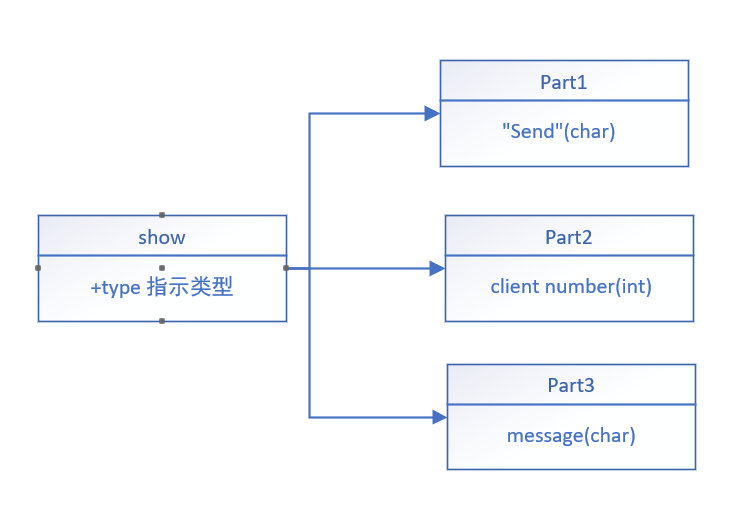
请求类型以String类型的字符串标识，当请求为消息类型时发送带有消息的请求包。



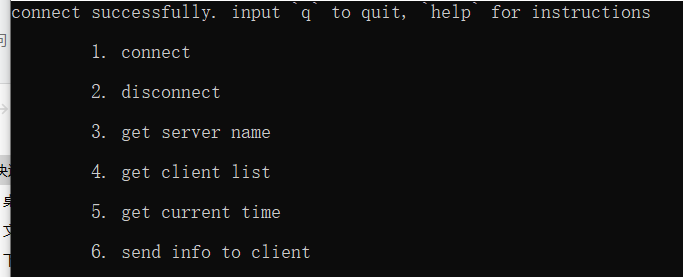
* 描述响应数据包的格式（画图说明），响应类型的定义



* 描述指示数据包的格式（画图说明），指示类型的定义



* 客户端初始运行后显示的菜单选项



* 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）

循环进行消息队列轮询，根据接收到消息的不通执行不同的操作

“q”:如果有活跃链接就断开链接

“help”:显示菜单栏

“1”：连接服务器

“2”：断开服务器

“3”：向服务器发送请求获取服务器名字

“4”：向服务器发送请求获取用户列表

“5”：向服务器发送请求获取时间

“6”：向指定用户发送指定信息

\*仅展示部分代码，完整代码见附件

  while (!client.real\_test\_lock());

            printf("localhost/> ");

            getline(cin, msg);

            if (msg == "q") {

                // 如果有活跃连接，先断开连接

                if (isConnect = true) {

                    cout << "please wait...";

                    isConnect = false;

                    Sleep(1000);

                    client.ClientClose();

                    cout << "done" << endl;

                }

                break;

            }

            else if (msg == "help") {

                cout << endl;

                cout << "\t1. connect\n" << endl;

                cout << "\t2. disconnect\n" << endl;

                cout << "\t3. get server name\n" << endl;

                cout << "\t4. get client list\n" << endl;

                cout << "\t5. get current time\n" << endl;

                cout << "\t6. send info to client\n" << endl;

                cout << "\t`q`to quit, `help`for instructions" << endl;

                cout << endl;

            }

            else

            {

                if (msg == "1") {

                    if (isConnect == false) {

                        int port;

                        string ip;

                        cout << "IP: ";

                        getline(cin, ip);

                        rlt = client.ClientConnect(2459, ip.c\_str());

                        if (rlt != 0) {

                            cout << "Error: Connection failed. ( " << rlt << " )\n" << endl;

                            continue;

                        }

                        cout << "Connection success!\n" << endl;

                        isConnect = true;

                    }

                    else {

                        cout << "Error: Can't connect to more than ONE server.\n" << endl;

                    }

                }

* 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）

（1）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::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();

    }

}

（2）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。

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/> ";

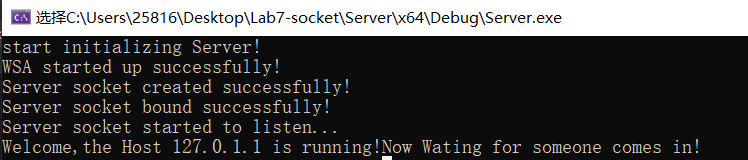
        Sleep(500);

        // 使用之后清空

    }

}

* 服务器初始运行后显示的界面



* 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）

初始化服务器，并执行WaitForClient()操作

    ServerNet svr;

    list\_client = (P\_Client)malloc(sizeof(struct Client));

    list\_client->Next = NULL;

    list\_client->Num = 0;

    //wait for clients' connection

    svr.WaitForClient();

    //server over

    getchar();

    return 0;

* 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）
  + - 1. 如果连接到客户端就继续执行，并打印相关提示。
      2. 将客户端加入列表中，然后创建子线程

while (true)

    {

        //if connect to client, then continue

        //else block

        socketClient = accept(socketServer, (SOCKADDR \*)&addrClient, &addrLength);

        //connection failed

        if (socketClient == INVALID\_SOCKET)

        {

            printf("Accept Failed\n");

            continue;

        }

        printf("Accept Success\n");

        InetNtopW(addrClient.sin\_family, &addrClient, bufferForIP, IP\_SIZE);

        //print the ip and port of client

        cout << "A new client connected! The IP address: " << inet\_ntoa(addrClient.sin\_addr) << ", port number: " << ::ntohs(addrClient.sin\_port) << endl;

        //add the client to the list

        Client::Client\_add(list\_client, socketClient, inet\_ntoa(addrClient.sin\_addr), ntohs(addrClient.sin\_port));

        //create subthread

        hThread = CreateThread(NULL, 0, CreateClientThread, (LPVOID)socketClient, 0, NULL);

        if (hThread == NULL)

        {

            cerr << "Failed to create a new thread!Error code: " << ::WSAGetLastError() << endl;

            WSACleanup();

            getchar();

            exit(1);

        }

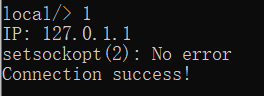
        CloseHandle(hThread);

    }

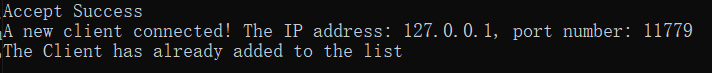
    return;

* 客户端选择连接功能时，客户端和服务端显示内容截图。

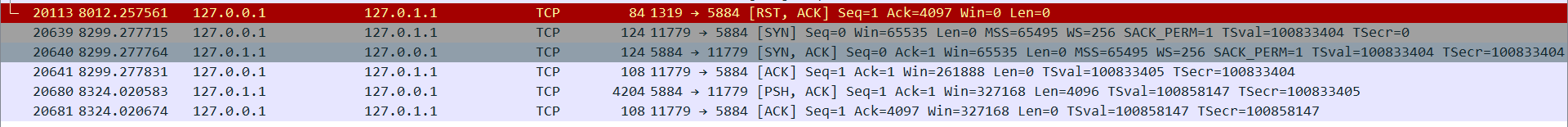
客户端：



服务端：

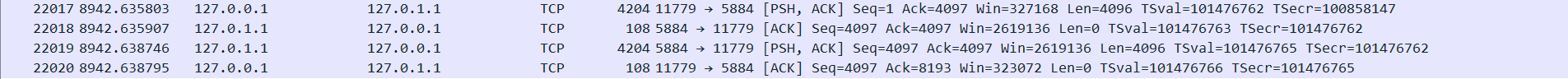


Wireshark抓取的数据包截图：



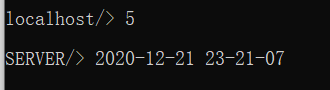
* 客户端选择获取时间功能时，客户端和服务端显示内容截图。

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）：

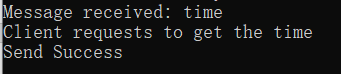


* 客户端选择获取名字功能时，客户端和服务端显示内容截图。

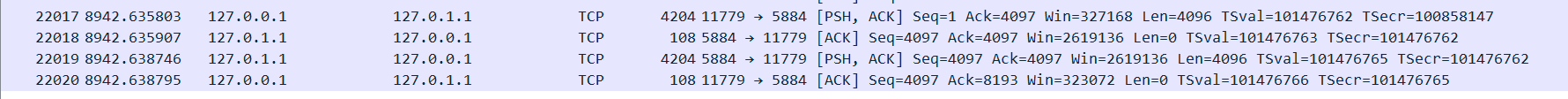
客户端：



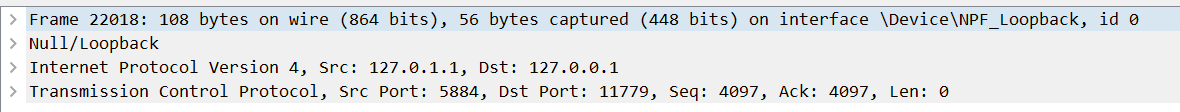
服务器：



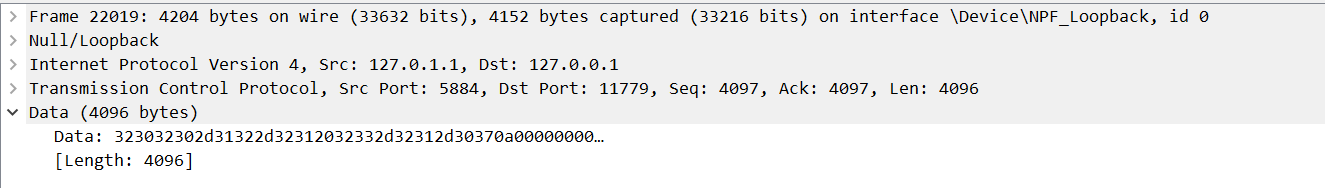
Wireshark抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的名字数据对应的位置）：



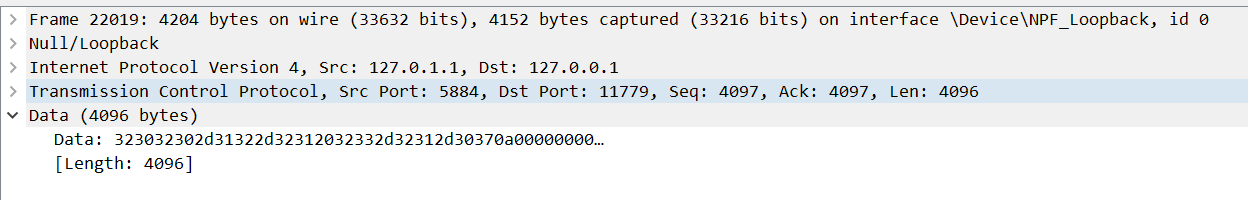
响应



请求



返回



相关的服务器的处理代码片段：

     else if (strstr(bufMessage, "time") != NULL)

   {

       //print the request msg from client

       cout << "Message received: " << bufMessage << endl;

       //reset Buffer

       memset(bufMessage, 0, MAX\_MSG\_SIZE);

       cout << "Client requests to get the time" << endl;

       //get current time and copy to buffer

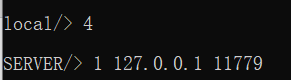
       time\_t t = time(0);

       strftime(bufMessage, sizeof(bufMessage), "%Y-%m-%d %H-%M-%S\n", localtime(&t));

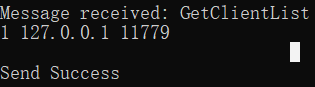
   }

* 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。

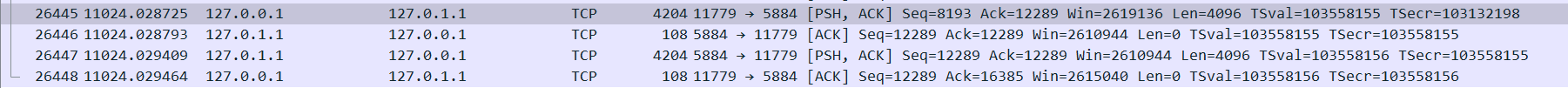
客户端：



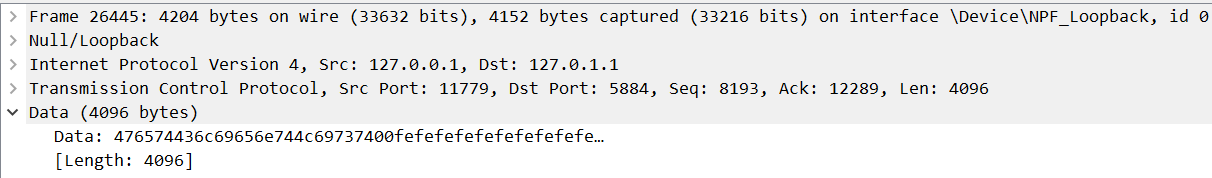
服务器：



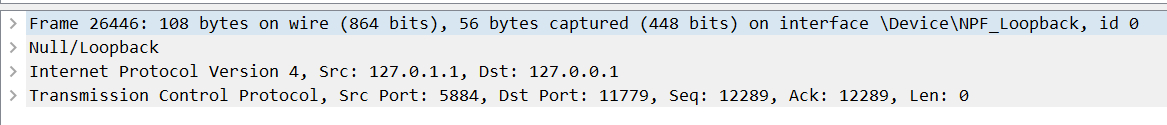
Wireshark抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：



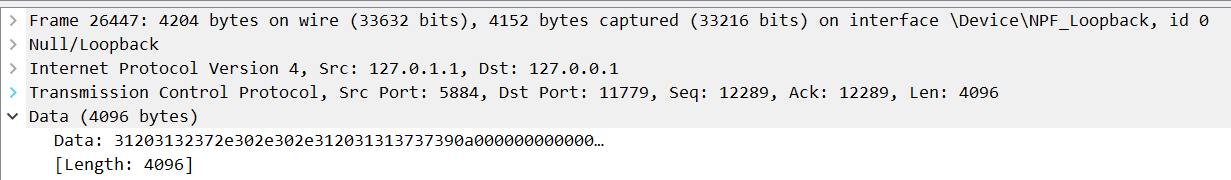
请求：



响应：



返回：



相关的服务器的处理代码片段：

   else if (strstr(bufMessage, "GetClientList") != NULL)

   {

       cout << "Message received: " << bufMessage << endl;

       memset(bufMessage, 0, MAX\_MSG\_SIZE);

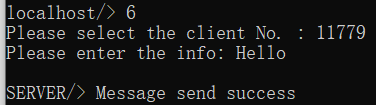
       Client::getList(list\_client, bufMessage);

       cout << bufMessage << endl;

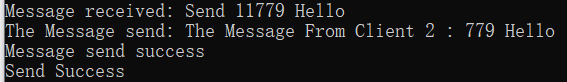
   }

* 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：



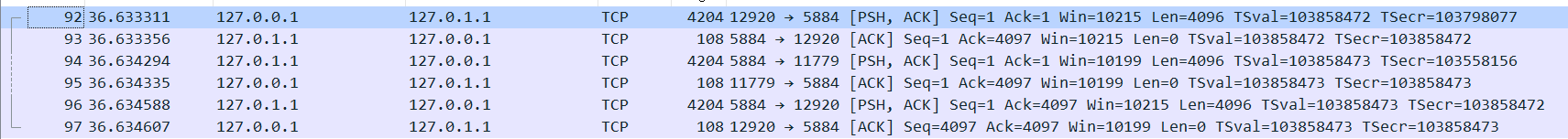
服务器：



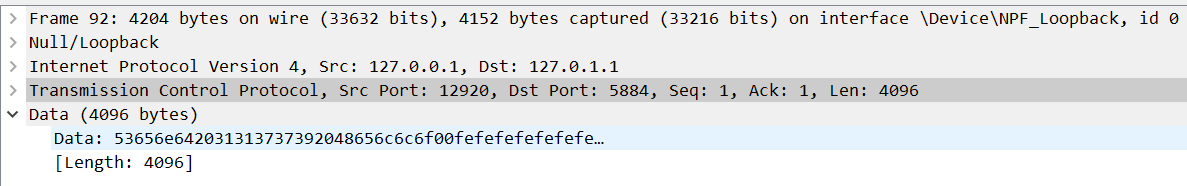
接收消息的客户端：



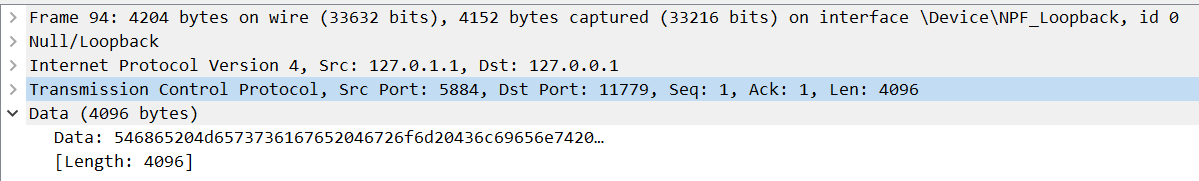
Wireshark抓取的数据包截图（发送和接收分别标记）：



发送：



接收：



相关的服务器的处理代码片段：

   else if (strstr(bufMessage, "Send") != NULL)

   {

       //receive send, print the message received

       cout << "Message received: " << bufMessage << endl;

       //get client number and get socket

       int num\_client = bufMessage[5] - '0';

       if (!(1 <= num\_client && num\_client <= 9))

       {

           memset(bufMessage, 0, MAX\_MSG\_SIZE);

           strcpy(bufMessage, "The Number is Error");

           cout << bufMessage << endl;

       }

       else

       {

           Mark = Client::getSock(list\_client, num\_client);

           if (Mark == INVALID\_SOCKET)

           {

               memset(bufMessage, 0, MAX\_MSG\_SIZE);

               strcpy(bufMessage, "The Number is Error,Please check the client list again");

               cout << bufMessage << endl;

           }

           else

           {

               char \* Message = &(bufMessage[7]);

               memset(AllMessage, 0, MAX\_MSG\_SIZE);

               strcpy(AllMessage, "The Message From Client ");

               char temp[5];

               //get the number of client

               num\_client = Client::getNum(list\_client, socketClient);

               temp[0] = num\_client + '0';

               temp[1] = ' ';

               temp[2] = ':';

               temp[3] = ' ';

               temp[4] = '\0';

               strcat(AllMessage, temp);

               //copy message to ALLMessage, send Buffer

               strncat(AllMessage, Message, MAX\_MSG\_SIZE - 28);

               cout << "The Message send: " << AllMessage << endl;

               //send message

               sendResult = send(Mark, AllMessage, MAX\_MSG\_SIZE, 0);

               if (sendResult == SOCKET\_ERROR)

               {

                   memset(bufMessage, 0, MAX\_MSG\_SIZE);

                   strcpy(bufMessage, "Message Failed,Check the Clinet list and try again");

                   cout << bufMessage << endl;

               }

               else

               {

                   //successfullly, send a message of success to client

                   memset(bufMessage, 0, MAX\_MSG\_SIZE);

                   strcpy(bufMessage, "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：

else if (msg == "6")

                {

                    string num;

                    string msg;

                    cout << "Please select the client No. : ";

                    getline(cin, num);

                    cout << "Please enter the info: ";

                    getline(cin, msg);

                    string pkg = "Send " + num + " " + msg;

                    strcpy\_s(msgBuffer, pkg.c\_str());

                    rlt = client.ClientSend(msgBuffer, MAX\_MSG\_SIZE);

                    msgBuffer[0] = '\0';

                }

int ClientNet::ClientSend(const char\* msg, int len)

{

    int rlt = 0;

    int iErrMsg = 0;

    //send message to sock

    iErrMsg = send(m\_sock, msg, len, 0);

    if (iErrMsg < 0)

        //fail

    {

        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;

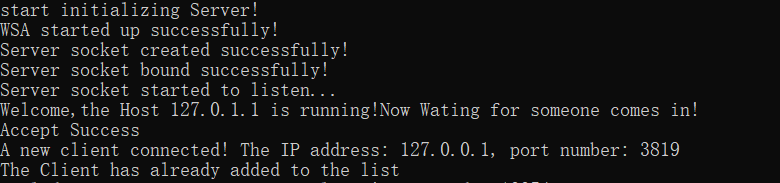
    }

    iotex.lock();

    return rlt;

}

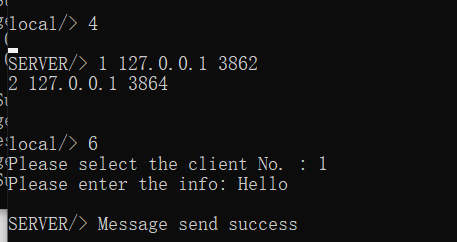
* 拔掉客户端的网线，然后退出客户端程序。观察客户端的TCP连接状态，并使用Wireshark观察客户端是否发出了TCP连接释放的消息。同时观察服务端的TCP连接状态在较长时间内（10分钟以上）是否发生变化。





说明：TCP建立连接三次握手后，当客户端断开网线时，客户端连接状态依然是ESTABLISHED，而且客户端没有发出TCP连接释放的消息。在较长时间内，服务端的TCP连接没有发生变化。通过查阅资料我们了解到TCP有一个KeepLive机制，默认情况下，当服务器2h内没有收到客户端的请求时，会发送1个探测响应，75秒后超时，总共发送10个这样的探测响应，服务器就认为客户端已经断开连接，于是就关闭连接。

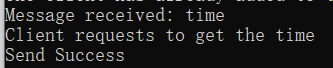
* 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？



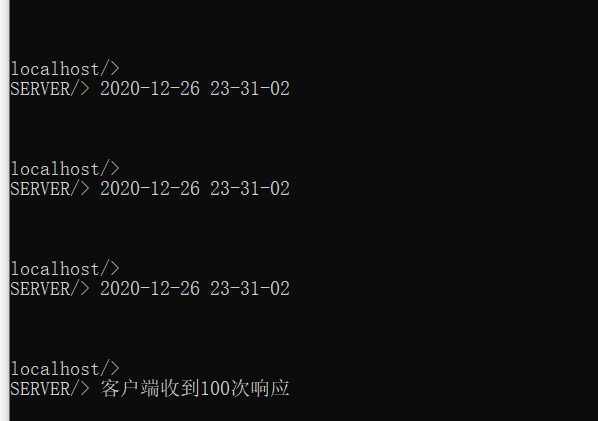
说明：之前的连接还在客户端列表中，当向之前连接发消息的时候，消息能够发送至对方客户端。

* 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。

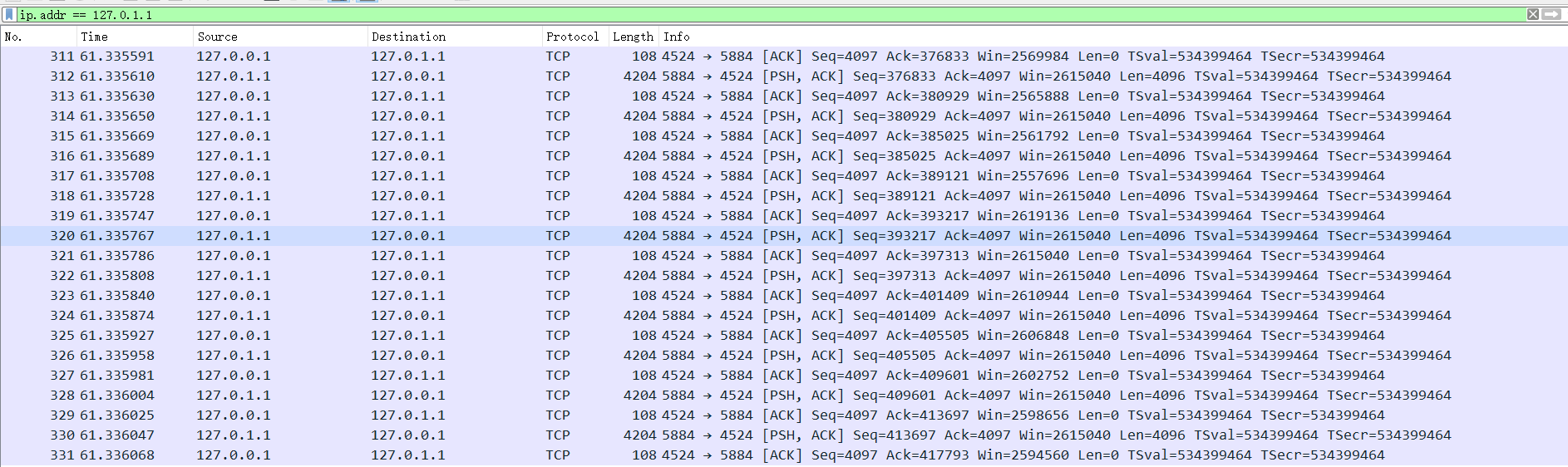
服务器：



客户端：



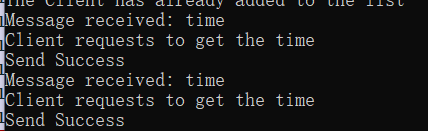
Wireshark



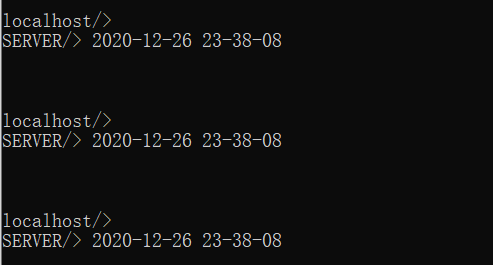
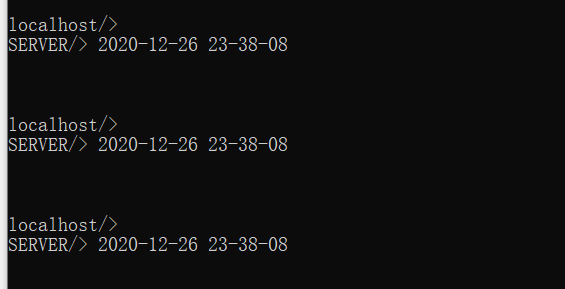
实际发出的数据包有约300个

* 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

服务器：



客户端：



# 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？

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；**
2. **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？

**1. 连接会因超时而失败**

* 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？

**1.** **不一完全一致，其中一次Tcp Segment中包含了多个请求数据。这也是因为Tcp发送请求数据大小小于发送缓冲区大小时，发送缓冲区直到满的时候，才会发送数据包。**

* 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？

**1. 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）

1. **连接之后，TCP状态如下，为‘ESTABLISHED’.**
2. **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？

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。**
2. **配置操作系统的SO\_KEEPALIVE选项，或者进行应用层心跳检测。**

# 讨论、心得

在Lab7:Socket编程的实验中，我们一开始对于TCP协议了解比较少，因此感到编程方面无法下手。通过上网查阅资料，阅读经典教材，努力不断学习相关知识，并且进行demo编写实验之后，经过了一段时间的编程后，我们的程序终于能够完成功能需求。

完成功能需求后，我们根据实验报告中的要求，使用wireshark抓包软件，对于客户端服务端的通信进行了测试，在实验过程中，我们通过具体的抓数据包展开分析，真实地看到了TCP三次握手（客户端发送SYN，服务端发送SYN和ACK，以及最后客户端回复ACK建立连接的具体现象）和四次挥手的现象，了解到TCP是面向流的协议，通过Acknowledge Number和Sequence Number来保证数据传输的顺序性，此外在实验过程中我们通过抓包也了解到了TCP会出现粘包拆包的现象，了解了原理后，对于编写代码也有很大的帮助。

虽然在编写过程中会遇到一些困难，但是我觉得对于理论课的理解会有很大的帮助，收获不少。