

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。

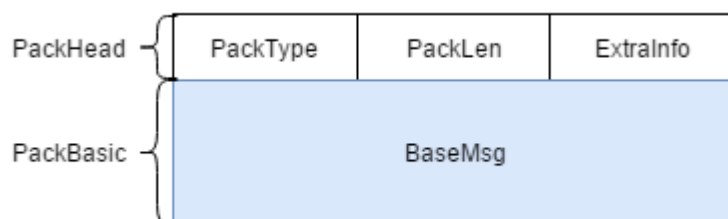
- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

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：0-3 字节用于标注数据包类型，4-7 字节记录整个数据包的长度，8-11 为保留字节。12-75 字节记录简单信息。

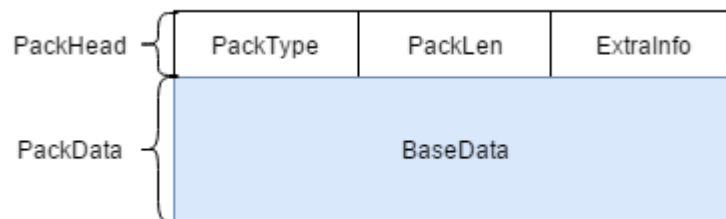


请求类型的定义：

```
1. //请求数据包类型
2. #define CONNECT_REQUEST 0X10 //连接请求
3.
4. #define REQUIRE_TIME 0X20 //获取时间
5.
6. #define REQUIRE_NAME 0X30 //获取名字
7.
8. #define REQUIRE_CLIENT_LIST 0X40 //获取客户端列表
9.
10. #define DISCONNECT_REQUEST 0X60 //断开连接请求
```

- 描述响应数据包的格式（画图说明），响应类型的定义

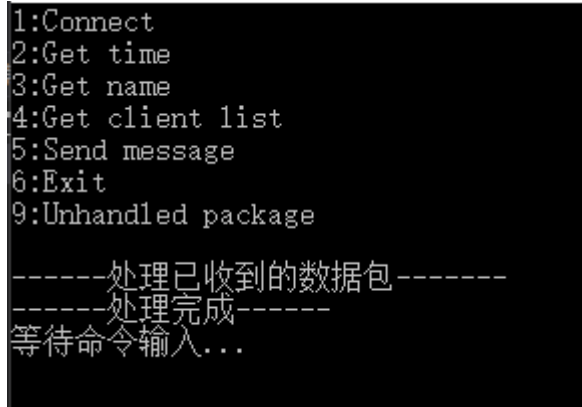
相应数据包格式：0-3 字节用于标注数据包类型，4-7 字节记录整个数据包的长度，8-11 传递额外信息。
12-2059 字节记录具体信息。



```
1. //数据包类型
2. #define CONNECT_SECC 0X11 //连接成功
3. #define CONNECT_FAIL 0X12 //连接失败
4.
5. #define SEND_TIME 0X21 //发送时间
6.
7. #define SEND_NAME 0X31 //发送名字
8.
9. #define SEND_CLIENT_LIST 0X41 //发送客户端列表
10.
11. #define MESSAGE 0X50 //收到需要转发给某个用户端的数据
12. #define MESSAGE_SUCC 0X51 //成功转发
13. #define MESSAGE_FAIL 0X52 //转发失败
14.
```

```
15. #define DISCONNECT_SECC 0X61 //断开连接成功
16. #define DISCONNECT_FAIL 0X62 //断开连接失败
```

- 客户端初始运行后显示的菜单选项



```
1:Connect
2:Get time
3:Get name
4:Get client list
5:Send message
6:Exit
9:Unhandled package

-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
```

菜单选项：1：开始一个新的连接，随后用户需按照提示输入服务端 IP 地址和端口；2：获取时间；3 获取服务器名字；4：获取客户端列表；5 发送消息，随后用户需要输入目的客户端的编码，和具体的消息内容；6：退出该客户端；9：查看已经收到的未处理的包裹。

- 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）

客户端主线程进行循环，首先检查是否存在已接收未处理的数据包，（根据变量 UnHandelPack 进行判断），若有，则打印数据包，并进行相应的处理。之后等待用户键盘输入命令，执行相应的操作。

```
1. while (true)
2.     {
3.         printf("\n-----处理已收到的数据包-----\n");
4.         //检查接收到的数据，并打印
5.         while (UnHandelPack > 0)
6.         {
7.             //对数据包进行处理
8.             if (PackList[UnHandelPack - 1].Head.PackType == MESSAGE)
9.             {
10.                //跟服务器说已经收到
11.                if (sclient == SOCKET_ERROR)
12.                {
13.                    printf("ERROR:Invalid port or ip address!");
14.                }
15.                else
16.                {
```

```

17.         printf("收到来自客户端的消息\n");
18.         PPACK MsgRec = ReMsgPackSucc();
19.         send(sclient, (char*)MsgRec, MsgRec->Head.PackLen,0);
20.     }
21. }
22. else if (PackList[UnHandelPack - 1].Head.PackType == MESSAGE_SUCC)
23. {
24.     printf("用户端%d 消息发送成功\n",WaitToConfirm);
25. }
26. //打印数据包
27. PrintPack(&PackList[UnHandelPack - 1]);
28. UnHandelPack--;
29. }
30. printf("-----处理完成-----\n");
31. cout << "等待命令输入..." << endl;
32. int CmdId;
33. cin >> CmdId;
34.
35. switch (CmdId)
36. {
37.     case CONN_SERVER:
38.     {
39.         char Address[BUFFER_SIZE] = "127.0.0.1";
40.         int port = 4112;
41.         /*cout<<"Input ip address and port"<<endl;
42.         cin >> Address >> port;*/
43.
44.         sclient = ConnectServer((char*)Address, port);
45.         if (sclient == SOCKET_ERROR)
46.         {
47.             printf("ERROR:Invalid port or ip address!");
48.         }
49.         else//创建用于通讯的子进程
50.         {
51.             HANDLE hThread;
52.             DWORD  threadId;
53.             hThread = CreateThread(NULL, 0, ThreadClient, &sclient, 0, &threadI
d); // 创建线程
54.         }
55.         break;
56.     }
57.     case TIME:
58.     {
59.         if (sclient == SOCKET_ERROR)

```

```

60.         {
61.             printf("ERROR:Connect First!");
62.         }
63.         else
64.         {
65.             PPACK pPack = RequirePack(REQUIRE_TIME);//生成一个请求时间的数据包
66.             send(sclient, (const char*)pPack, pPack->Head.PackLen, 0);
67.             printf("发送请求时间数据包...\n");
68.         }
69.         break;
70.     }
71.     case NAME:
72.     {
73.         if (sclient == SOCKET_ERROR)
74.         {
75.             printf("ERROR:Connect First!");
76.         }
77.         else
78.         {
79.             PPACK pPack = RequirePack(REQUIRE_NAME);//生成一个请求时间的数据包
80.             send(sclient, (const char*)pPack, pPack->Head.PackLen, 0);
81.             printf("发送请求名字数据包...\n");
82.         }
83.         break;
84.     }
85.     case CLIENT_LIST:
86.     {
87.         if (sclient == SOCKET_ERROR)
88.         {
89.             printf("ERROR:Connect First!");
90.         }
91.         else
92.         {
93.             PPACK pPack = RequirePack(QEQUIRE_CLIENT_LIST);//生成一个请求客户端列
表的数据包
94.             send(sclient, (const char*)pPack, pPack->Head.PackLen, 0);
95.             printf("发送请求客户端列表数据包...\n");
96.         }
97.         break;
98.     }
99.     case SEND_MSG_TO:
100.    {
101.        if (sclient == SOCKET_ERROR)
102.        {

```

```

103.             printf("ERROR:Connect First!");
104.         }
105.     else
106.     {
107.         int ClientIndex;
108.         string Msg;
109.         int confirm;
110.         cout << "Input the ClientIndex" << endl;
111.         cin >> ClientIndex;
112.         cout << "Input Message to send" << endl;
113.         cin >> Msg;
114.         cout << "1:Comfirm 0:ReInput" << endl;
115.         cin >> confirm;
116.
117.         if (confirm == 1)
118.         {
119.             //发送消息数据包
120.             int success;
121.             PPACK MsgPack = DataPack(MESSAGE, (char*)Msg.c_str(), ClientIndex);
122.             PrintPack(MsgPack);
123.             send(sclient, (const char*)MsgPack, MsgPack->Head.PackLen, 0);
124.
125.             cout << "发送消息数据包..." << endl;
126.             //cout << "Send message successfully." << endl;
127.             WaitToConfirm = ClientIndex;
128.         }
129.     }
130. }
131. case EXIT:
132. {
133.     //发送断开数据包
134.     PPACK pPack = RequirePack(DISCONNECT_REQUEST); //生成一个请求时间的数据包
135.     send(sclient, (const char*)pPack, pPack->Head.PackLen, 0);
136.     printf("发送请求断开连接数据包...\n");
137.
138.     //先断开连接，再退出
139.     closesocket(sclient);
140.     WSACleanup();
141.     return 0;
142. }
143. case UNHANDLED:

```



```

144.         {
145.             printf("跳过, 查看收到的数据包");
146.             break;
147.         }
148.     }
149. }

```

- 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）

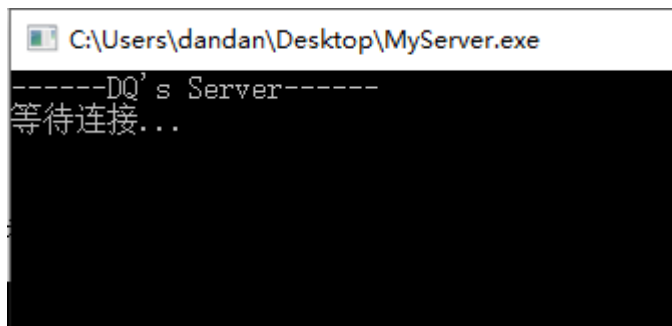
子线程调用 `recv` 函数，持续接收新的数据。将数据包拷贝到全局变量 `PackList` 中，并按照未处理数据包接受时间进行排列。本质就是将接收到的数据包存在列表中，并用整数 `UnHandelPack` 来记录数量，以此达成和主线程的信息交流。

```

1.  //循环等待消息并作出响应
2.  while (true)
3.  {
4.      //接收数据
5.      char revData[4096];
6.      int ret = recv(s, revData, 4096, 0);
7.      if (ret > 0)
8.      {
9.          PPACK pPack1 = (PPACK)revData;
10.         PackList[UnHandelPack].Head.PackType = pPack1->Head.PackType;
11.         PackList[UnHandelPack].Head.PackLen = pPack1->Head.PackLen;
12.         PackList[UnHandelPack].Head.ExtraInfo = pPack1->Head.ExtraInfo;
13.         if(pPack1->Head.PackType== SEND_CLIENT_LIST)
14.             memcpy((char*)&PackList[UnHandelPack].PackData, (char*)&pPack1->PackData, sizeof(Client)*pPack1->Head.ExtraInfo);
15.         else
16.         {
17.             memcpy((char*)&PackList[UnHandelPack].PackData, (char*)&pPack1->PackData, sizeof(PACKDATA));
18.         }
19.         UnHandelPack++;
20.     }
21. }

```

- 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）

服务器主线程调用 `ConnClient` 函数，持续等待新的连接出现。一旦有新的连接出现，创建新的线程，用于监听该连接发送过来的数据包。

```
1. while (true)
2. {
3.     //等待连接
4.     SOCKET sClient = ConnClient(slisten, 4112);
5.
6.     //创建新的线程用于通信
7.     HANDLE hThread;
8.     DWORD  threadId;
9.     hThread = CreateThread(NULL, 0, ThreadClient, &sClient, 0, &threadId); // 创建线程
10.    ThreadHandleList[ThreadNum] = hThread;
11.    ThreadNum++;
12.    printf("子进程%d 等待接受消息...\n", threadId);
13. }
```

- 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）

子线程等待该连接发送过来的数据包，通过检查数据包中 `PackType` 数值，做出相应的反应。

类型包括 1) 请求时间 `REQUIRE_TIME`，则发送一个时间数据包；2) 请求名字 `REQUIRE_NAME`，发送名字数据包；3) 请求客户端列表 `REQUIRE_CLIENT_LIST`，发送客户端列表；4) 收到转发信息请求 `MESSAGE`，检查目的客户端连接序号，转发该数据包；5) 转发成功 `MESSAGE_SUCC`，表明上一个转发给目的客户端的数据包已经被收到，发送“发送成功”数据包给源客户端；6) 断开连接请求 `DISCONNECT_REQUEST`，断开相应的连接，并退出子线程。

```

1. //循环等待消息并作出响应
2. while (true)
3. {
4.     //接收数据
5.     char revData[4096];
6.     int ret = recv(s, revData, 4096, 0);
7.     if (ret > 0)
8.     {
9.         //printf("%d\n", ret);
10.        revData[ret] = 0x00;
11.        PPACK pPack = (PPACK)revData;
12.        //test
13.        printf("收到数据包, 正在处理...\n");
14.        PrintPack(pPack);
15.        int CmdID = pPack->Head.PackType;
16.        switch (CmdID)
17.        {
18.            case REQUIRE_TIME:
19.            {
20.                cout << "发送时间..." << endl;
21.                time_t timep;
22.                struct tm *p;
23.                time(&timep);
24.                p = gmtime(&timep);
25.                string Time;
26.                Time += "time:" + to_string(p->tm_mon)
27.                    + "-" + to_string(p->tm_mday) + " " + to_string(p->tm_hour)
28.                    + ":" + to_string(p->tm_min) + ":" + to_string(p->tm_sec);
29.                //cout << Time << endl;
30.                PPACK TimeDataPack = DataPack(SEND_TIME, (char*)Time.c_str(),0);
31.
32.                //PrintPack(TimeDataPack);
33.                send(s, (char*)TimeDataPack, TimeDataPack->Head.PackLen, 0);
34.                break;
35.            }
36.            case REQUIRE_NAME:
37.            {
38.                cout << "发送名字..." << endl;
39.                string Name = "DQ's Server";
40.                PPACK NamePack = DataPack(SEND_NAME, (char*)Name.c_str(),0);
41.                //PrintPack(NamePack);
42.                send(s, (char*)NamePack, NamePack->Head.PackLen, 0);
43.                break;
44.            }

```

```

45.         case QEQUIRE_CLIENT_LIST:
46.         {
47.             cout << "发送客户端列表" << endl;
48.             char ClientListData[4096];
49.             memcpy(ClientListData, &ClientList, sizeof(CLIENT));
50.
51.             PPACK ListPack = DataPack(SEND_CLIENT_LIST, ClientListData, ConnClientNum);
52.
53.             //printf("发送出去的数据包\n");
54.             //PrintPack(ListPack);
55.             send(s, (char*)ListPack, ListPack->Head.PackLen, 0);
56.             break;
57.         }
58.         case MESSAGE:
59.         {
60.             //转发给相应的客户端
61.             int SendToClient = pPack->Head.ExtraInfo;
62.             if (SendToClient <= 0 || SendToClient > ConnClientNum)
63.             {
64.                 //发送转发失败的数据包回去
65.                 PPACK MsgFailPack = ReMsgPackFail();
66.                 printf("转发失败, 通知用户端...\n");
67.                 //PrintPack(MsgFailPack);
68.                 send(s, (char*)MsgFailPack, MsgFailPack->Head.PackLen, 0);
69.             }
70.             else
71.             {
72.                 cout << "转发 ing..." << endl;
73.                 SOCKET sendTo = ClientList[SendToClient - 1].Socket;
74.                 send(sendTo, (char*)pPack, pPack->Head.PackLen, 0);
75.                 WaitToConfirm = sendTo;
76.             }
77.             break;
78.         }
79.         case MESSAGE_SUCC:
80.         {
81.             cout << "转发成功, 通知用户端..." << endl;
82.             PPACK confirm = ReMsgPackSucc();
83.             send(WaitToConfirm, (char*)confirm, confirm->Head.PackLen, 0);
84.             break;
85.         }
86.         case DISCONNECT_REQUEST:
87.         {
88.             cout << "正在断开连接..." << endl;

```

```

88.          //清除已连接客户端信息
89.          for (int i = 0; i < ConnClientNum; i++)
90.          {
91.              if (ClientList[i].Socket == s)
92.              {
93.                  ClientList[i].IsConn = false;
94.                  break;
95.              }
96.          }
97.          return 0;
98.      }
99.  }
100. }
101. }

```

- 客户端选择连接功能时，客户端和服务端显示内容截图。

客户端截图：

```

1:Connect
2:Get time
3:Get name
4:Get client list
5:Send message
6:Exit
9:Unhandled package
-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
1
Input ip address and port
10.181.241.128
4112

-----HEAD-----
CdmID:17
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:A successful connection.

连接客户端成功
-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
子线程, pid = 23196

```

说明：进入客户端程序，首先显示菜单，然后处理已经收到的数据包（一开始为空），进入等待用户命令的循环。

当用户输入连接请求，提示用户输入连接地址和端口。随后等待服务器发回连接状态信息数据包，并打印消息。若连接成功，提示“连接客户端成功”。

主线程继续处理由子进程接受的数据包和“等待命令输入”的循环，新创建的子进程打印其 pid 并持续接收数据包。

服务端：

```
等待连接...
正在处理收到的数据包...

-----HEAD-----
CdmID:16
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:Require a new connection.

----客户端列表增加，目前列表----
Index  Socket  Isconn  Port    IP
1      196     1       4112    10.181.241.128
2      172     1       4112    10.180.90.166

子进程18724等待接受消息...
等待连接...
收到数据包，正在处理...
```

说明：服务端接收到一个请求连接的数据包时，打印该消息，并将客户端的信息加入到客户端列表中，并打印更新后的客户端列表。

创建一个子进程用于接受该连接上的数据包，并根据数据包的类型做出不同的处理，主进程持续等待接受新的连接。

Wireshark 抓取的数据包截图：

客户端发送给服务端的请求连接的数据包：

- 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：输入命令 2，客户端开始发送时间数据包，随后进入新一轮处理已经接收到的数据包，在第一轮处理中，时间数据包还没有送达，故处理处显示空白，可按 9 跳过新一轮的命令输入，进入新一轮的数据包处理。可以看到在第二轮中收到了类型为 33 的数据包，对应宏定义中 `#define SEND_TIME 0X21 //发送时间` 该项，打印数据包，可以查看数据包中的时间。

```
等待命令输入...
2
发送请求时间数据包...

-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
9
跳过，查看收到的数据包
-----处理已收到的数据包-----

-----HEAD-----
CdmID:33
PackLen:2060
DataLen:0
-----BODY-----
TimeData:time:11-23 5:51:9

-----处理完成-----
等待命令输入...
```

服务器：子线程提示收到数据包，正在处理。打印数据包内容，显示数据包类型为 32，对应

`#define REQUIRE_TIME 0X20 //获取时间` 十六进制的宏定义中的获取时间一项。根据该数据包类型，服务端发送给源客户端一个时间数据包，显示“发送时间...”提示。

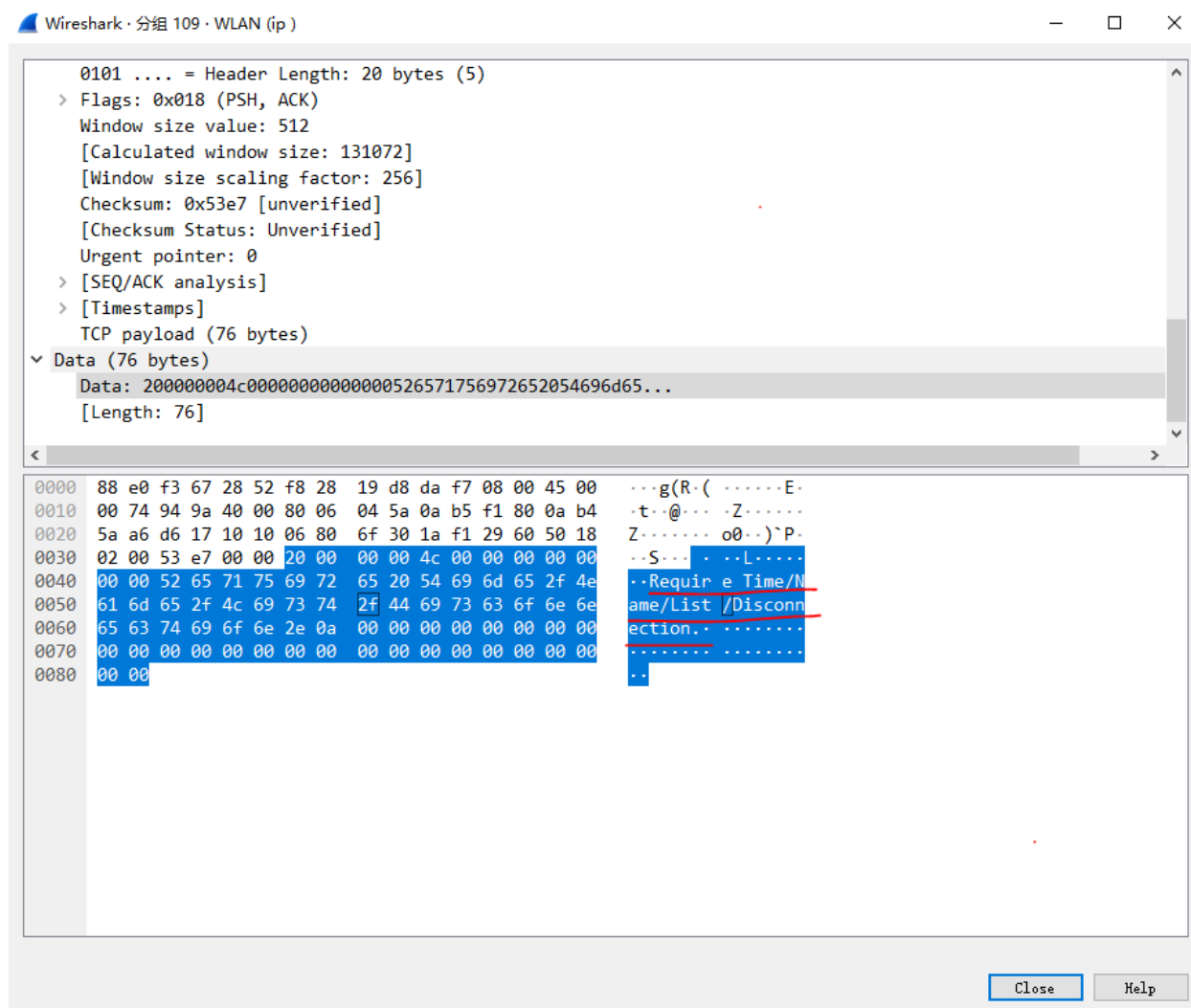
```
收到数据包，正在处理...

-----HEAD-----
CdmID:32
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:Require Time/Name/List/Disconnection.

发送时间...
```

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）：

客户端发送给服务端的时间数据包：请求类型位于非标蓝部分。



服务端向客户端发送的时间数据包：数据包 data 部分，未标蓝部分为响应或者请求类型，标蓝部分为时间信息。

```
> Frame 110: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
> Ethernet II, Src: JuniperN_67:28:52 (88:e0:f3:67:28:52), Dst: LiteonTe_d8:da:f7 (f8:28:19:d8:da:f7)
> Internet Protocol Version 4, Src: 10.180.90.166 (10.180.90.166), Dst: 10.181.241.128 (10.181.241.128)
> Transmission Control Protocol, Src Port: apple-vpns-rp (4112), Dst Port: 54807 (54807), Seq: 77, Ack: 153, Le
< Data (1460 bytes)
  Data: 210000000c0800000000000074696d653a31312d32332034...
  [Length: 1460]
```

0000	f8 28 19 d8 da f7 88 e0	f3 67 28 52 08 00 45 00	.(.....g(R..E-
0010	05 dc bf f0 40 00 7e 06	d5 9b 0a b4 5a a6 0a b5@~.Z...
0020	f1 80 10 10 d6 17 1a f1	29 60 06 80 6f 7c 50 10)`.o P.
0030	01 00 3f 11 00 00 21 00	00 00 0c 08 00 00 00 00	..?..!
0040	00 00 74 69 6d 65 3a 31	31 2d 32 33 20 34 3a 33	..time:1 1-23 4:3
0050	31 3a 30 00 00 00 00 00	00 00 00 00 00 00 00 00	1:0.....
0060	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0070	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0100	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

- 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：输入命令 3，客户端开始发送请求名字数据包，随后进入新一轮处理已经接收到的数据包，在第一轮处理中，名字数据包还没有送达，故处理处显示空白，可按 9 跳过新一轮的命令输入，进入新一轮的数据包处理。可以看到在第二轮中收到了类型 49 的数据包，对应宏定义中

```
#define SEND_NAME 0X31 //发送名字
```

一项，打印数据包，可以查看数据包中的名字信息

```

3
发送请求名字数据包...

-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
9
跳过，查看收到的数据包
-----处理已收到的数据包-----

-----HEAD-----
CdmID:49
PackLen:2060
DataLen:0
-----BODY-----
Name:DQ's Server
-----处理完成-----

```

服务端：子线程提示收到数据包，正在处理。打印数据包内容，显示数据包类型为 48，对应

`#define REQUIRE_NAME 0X30 //获取名字` 十六进制的宏定义中的获取名字一项。根据该数据包类型，服务端发送给源客户端一个时间数据包，显示“发送名字...”提示。

```

收到数据包，正在处理...

-----HEAD-----
CdmID:48
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:Require Time/Name/List/Disconnection.

发送名字...

```

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）：

客户端向服务端请求名字的数据包：

相关的服务器的处理代码片段：服务器检测到数据包中 `PackType` 为获取名字类型。创建新的数据包，将名字信息放入数据区域，发送数据包。

```
case REQUIRE_NAME:
{
    cout << "发送名字..." << endl;
    string Name = "DQ's Server";
    PPACK NamePack = DataPack(SEND_NAME, (char*)Name.c_str(), 0);
    //PrintPack(NamePack);
    send(s, (char*)NamePack, NamePack->Head.PackLen, 0);
    break;
}
```

- 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：输入命令 4，客户端开始发送请求客户端列表数据包，随后进入新一轮处理已经接收到的数据包，在第一轮处理中，数据包还没有送达，故处理处显示空白，可按 9 跳过新一轮的命令输入，进入新一轮的数据包处理。可以看到在第二轮中收到了类型 65 的数据包，对应宏定义中

`#define SEND_CLIENT_LIST 0X41 //发送客户端列表` 一项，打印数据包，可以查看数据包中的客户端列表信息。

```
等待命令输入...
4
发送请求客户端列表数据包...

-----处理已收到的数据包-----
-----处理完成-----
等待命令输入...
9
跳过，查看收到的数据包
-----处理已收到的数据包-----

-----HEAD-----
CdmID:65
PackLen:2060
DataLen:1
-----BODY-----
Index   Socket  Isconn  Port    IP
1       208     1       4112    10.181.171.55

-----处理完成-----
等待命令输入...
```

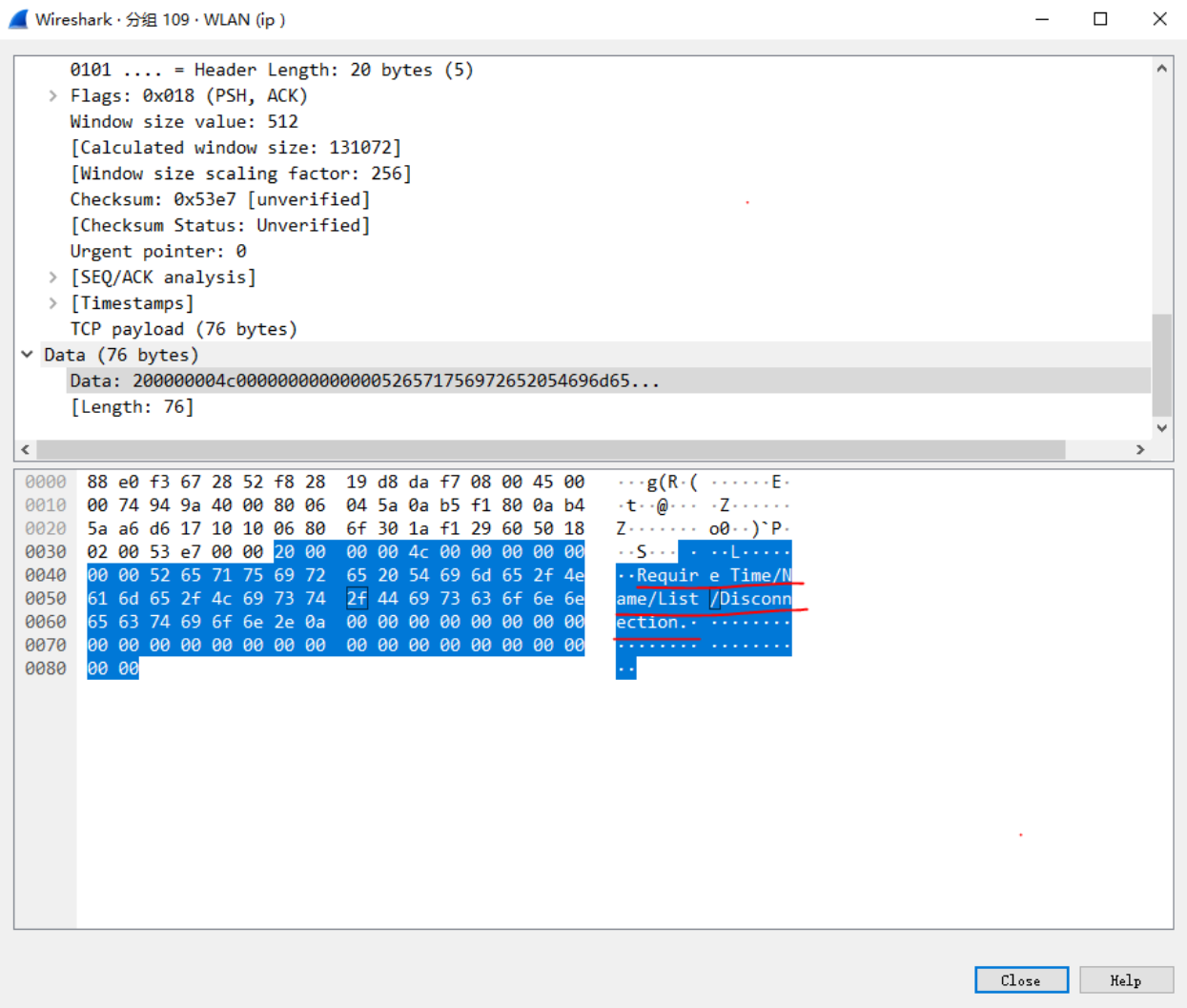
服务端：子线程提示收到数据包，正在处理。打印数据包内容，显示数据包类型为 64，对应

```
#define QEQUIRE_CLIENT_LIST 0x40 //获取客户端列表
```

十六进制的宏定义中的获取客户端列表一项。

根据该数据包类型，服务端发送给源客户端一个客户端列表数据包，显示“发送请求客户端列表数据包”提示。

客户端向服务端请求客户端列表的数据包：



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）：

服务端向客户端发送的客户端列表数据包：其中标蓝部分表示客户端列表数据，其余部分标示了数据包类型、列表长度等信息。

看收到的数据包，可以看到提示“用户端 1 消息发送成功”，表示刚刚输入的发送给客户端序号为 2 的信息发送成功。

```
-----处理完成-----  
等待命令输入...  
5  
Input the ClientIndex  
2  
Input Message to send  
肖战祝您新年快乐!!!  
1:Comfirm 0:ReInput  
1
```

```
-----处理已收到的数据包-----  
用户端2消息发送成功  
  
-----HEAD-----  
CdmID:81  
PackLen:76  
DataLen:0  
-----BODY-----  
BaseMsg:A successful message.  
  
-----处理完成-----
```

服务器：服务器收到信息类数据包，打印消息，转发对相应的目的客户端，显示“转发 ing...”（图 1）

等收到目的客户端的成功通知，发送转发成功数据包给源客户端。（图 2）

```
收到数据包，正在处理...  
  
-----HEAD-----  
CdmID:80  
PackLen:2060  
DataLen:2  
-----BODY-----  
Msg:肖战祝您新年快乐!!!  
  
转发ing...
```

```
收到数据包，正在处理...  
  
-----HEAD-----  
CdmID:81  
PackLen:76  
DataLen:0  
-----BODY-----  
BaseMsg:A successful message.  
  
转发成功，通知用户端...
```

接收消息的客户端：目的客户端收到消息，进行打印，并发送接受成功数据包给服务端。

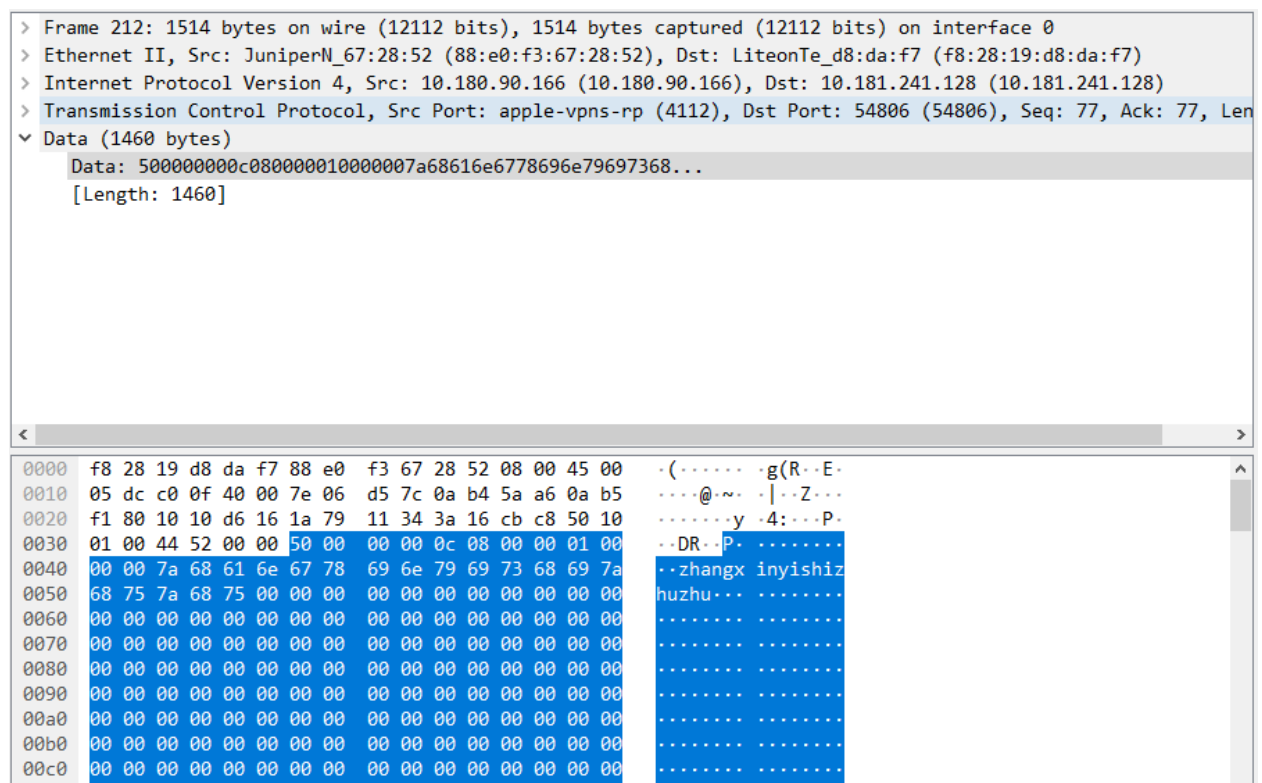

```

收到来自客户端的消息
-----HEAD-----
CdmID:80
PackLen:2060
DataLen:2
-----BODY-----
Msg:肖战祝您新年快乐!!!
-----处理完成-----

```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

客户端发送给服务器的消息数据包：其中标蓝部分为消息的具体内容，其他部分标识了数据包类型等信息。



相关的服务器的处理代码片段：分为两个过程：服务器收到信息类数据包，打印消息，转发到相应的目的客户端，显示“转发 ing...”

等收到目的客户端的成功通知，发送转发成功数据包给源客户端。

```

case MESSAGE:
{
    //转发给相应的客户端
    int SendToClient = pPack->Head.ExtraInfo;
    if (SendToClient <= 0 || SendToClient > ConnClientNum)
    {
        //发送转发失败的数据包回去
        PPACK MsgFailPack = ReMsgPackFail();
        printf("转发失败, 通知用户端...\n");
        //PrintPack(MsgFailPack);
        send(s, (char*)MsgFailPack, MsgFailPack->Head.PackLen, 0);
    }
    else
    {
        cout << "转发ing..." << endl;
        SOCKET sendTo = ClientList[SendToClient - 1].Socket;
        send(sendTo, (char*)pPack, pPack->Head.PackLen, 0);
        WaitToConfirm = sendTo;
    }
    break;
}
case MESSAGE_SUCC:
{
    cout << "转发成功, 通知用户端..." << endl;
    PPACK confirm = ReMsgPackSucc();
    send(WaitToConfirm, (char*)confirm, confirm->Head.PackLen, 0);
    break;
}
}

```

相关的客户端（发送和接收消息）处理代码片段：如果收到来自客户端的消息数据包，通知服务器已经收到，打印数据包（接受方）；如果收到来自服务器的发送成功通知，打印通知用户发送给某客户端的信息已成功（发送方）。

```

printf("\n-----处理已收到的数据包-----\n");
//检查接收到的数据，并打印
while (UnHandelPack > 0)
{
    //对数据包进行处理
    if (PackList[UnHandelPack - 1].Head.PackType == MESSAGE)
    {
        //跟服务器说已经收到
        if (sclient == SOCKET_ERROR)
        {
            printf("ERROR:Invalid port or ip address!");
        }
        else
        {
            printf("收到来自客户端的消息\n");
            PPACK MsgRec = ReMsgPackSucc();
            send(sclient, (char*)MsgRec, MsgRec->Head.PackLen, 0);
        }
    }
    else if (PackList[UnHandelPack - 1].Head.PackType == MESSAGE_SUCC)
    {
        printf("用户端%d消息发送成功\n", WaitToConfirm);
    }
    //打印数据包
    PrintPack(&PackList[UnHandelPack - 1]);
    UnHandelPack--;
}
printf("-----处理完成-----\n");

```

- 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。

没有发出 TCP 连接释放的消息。在较长时间后，TCP 连接断开了，TCP 套接字的保持存活选项 SO_KEEPALIVE，如果在两个消失之内在该套接字的任何一个方向上都没有数据交换，TCP 就自动给对端发送一个保持存活探测分节。如果没有对此 TCP 探测分节的任何响应，该套接字的处理错误就被置为 ETIMEOUT，套接字本身则被关闭。

- 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？

还存在。发送消息之后，由于目的客户端已经断开，TCP 协议由于没有收到 ACK 包，于是协议栈会自动进行重传，当重传达到一定次数后，就会发 RST 包，重置连接。

- 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。

服务器处理了 44 次请求：

```
41发送时间...
43
42发送时间...
44
```

客户端接收到 22 个包：

```
21
-----HEAD-----
CdmID:33
PackLen:2060
DataLen:0
-----BODY-----
TimeData:time:11-23 14:19:27
22
```

- 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：

```
-----HEAD-----
CdmID:32
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:Require Time/Name/List/Disconnection.

收到数据包，正在处理...

-----HEAD-----
CdmID:32
PackLen:76
DataLen:0
-----BODY-----
BaseMsg:Require Time/Name/List/Disconnection.

123发送时间...
125
124发送时间...
126
```

客户端:

19	19	19
-----HEAD-----	-----HEAD-----	-----HEAD-----
CdmID:33	CdmID:33	CdmID:33
PackLen:2060	PackLen:2060	PackLen:2060
DataLen:0	DataLen:0	DataLen:0
-----BODY-----	-----BODY-----	-----BODY-----
TimeData:time:11-23 14:25	TimeData:time:11-23 14:25:46	TimeData:time:11-23 14:25:43
20	20	20
-----HEAD-----	-----HEAD-----	-----HEAD-----
CdmID:33	CdmID:33	CdmID:33
PackLen:2060	PackLen:2060	PackLen:2060
DataLen:0	DataLen:0	DataLen:0
-----BODY-----	-----BODY-----	-----BODY-----
TimeData:time:11-23 14:25	TimeData:time:11-23 14:25:46	TimeData:time:11-23 14:25:43
21	21	21
-----HEAD-----	-----HEAD-----	-----HEAD-----
CdmID:33	CdmID:33	CdmID:33
PackLen:2060	PackLen:2060	PackLen:2060
DataLen:0	DataLen:0	DataLen:0
-----BODY-----	-----BODY-----	-----BODY-----
TimeData:time:11-23 14:25	TimeData:time:11-23 14:25:46	TimeData:time:11-23 14:25:43
-----处理完成-----	-----处理完成-----	-----处理完成-----
等待命令输入...	等待命令输入...	等待命令输入...

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，使用 `SOCKET` `sclient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);` 语句创建源端口，同一个套接字调用 connect 时，端口保持不变，如果创建了新的连接，端口可以发生改变。

- 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？

不能。

- 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？

每一个连接都创建了一个子进程用于接受数据，与子进程相关的连接是唯一的，子进程将收到的数据包放在共享内存区域，并加上进程或者连接标识符，主线程在处理数据时就可以区分。

- 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）

第一次挥手:客户端向服务器端发送 FIN，表示要断开连接(客户端不再发送数据了)，然后进入 FIN_WAIT_2 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？

没有变化，服务器不知道客户端的异常情况，产生了一个半开连接。在较长时间后，TCP 连接断开了，TCP 套接字的保持存活选项 `SO_KEEPALIVE`，如果在两个消失之内在该套接字的任何一个方向上都没有数据交换，TCP 就自动给对端发送一个保持存活探测分节。如果没有对此 TCP 探测分节的任何响应，该套接字的处理错误就被置为 `ETIMEOUT`，套接字本身则被关闭。

七、 讨论、心得

在做实验的过程中，对 socket 网络编程的基本用法有了更深入的认识，对 tcp 协议也有了更多了解。