

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:	猜猜
学 院:	计算机科学与技术学院
系:	计算机科学与技术学院
专 业:	数字媒体技术
学 号:	猜猜
指导教师:	

2019 年 10 月 1 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: 猜猜 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

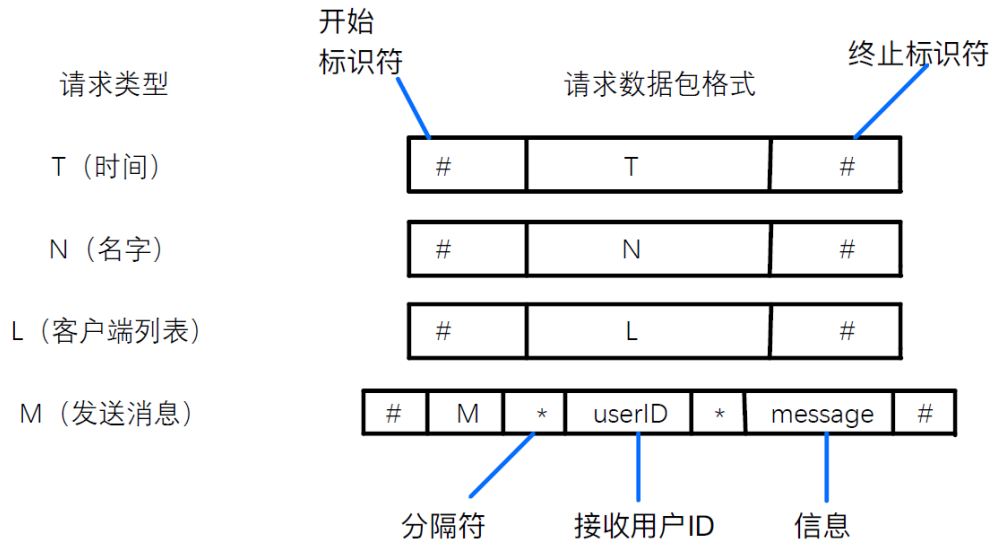
五、实验数据记录和处理

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**
 1. 请求类型的定义如下所示：

请求类型	定义
T（时间）	要求服务器返回当前时间
N（名字）	要求服务器返回自己的用户名
L（客户端列表）	要求服务器返回当前连接的所有用户的信息
M（发送消息）	给一个特定的用户（同连在这个服务器上的）发送消息

2. 请求数据包的格式如下所示：



● 描述响应数据包的格式（画图说明），响应类型的定义

1. 响应的类型定义如下：

响应类型		定义
T（时间）		服务器返回当前时间
N（名字）		服务器返回用户名
L（客户端列表）		服务器返回当前连接的所有用户的信息
M（发送消息）	M-Y（发送信息成功）	信息发送是否成功
	M-N（发送消息失败）	

2. 响应数据包的格式如下：

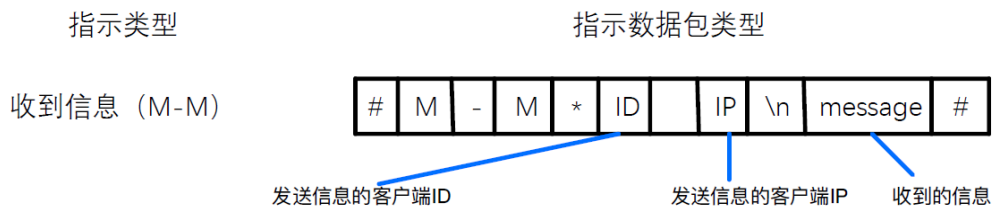


● 描述指示数据包的格式（画图说明），指示类型的定义

1. 指示的类型定义如下：

相应类型		定义
M	M-M（发送给目的客户端的信息）	转发给目的客户端的信息

2. 指示数据包的格式如下：



● 客户端初始运行后显示的菜单选项

```

Please input the operation number:
+-----+-----+
| Input | Function |
+-----+-----+
| 1     | Connect to a server. |
| 2     | Close the connect.  |
| 3     | Get the server time. |
| 4     | Get the server name. |
| 5     | Get the client list. |
| 6     | Send a message.     |
| 7     | Exit.                |
+-----+-----+
  
```

如图所示，客户端启动后，会出现指示菜单栏，包含 7 个功能：连接到服务器、关闭连接、获取时间、获取服务器名字、获取客户端列表、发送信息、退出。用户可输入相应的命令编号来执行操作。

- 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）

```
1. while (1)
2. {
3.     //开始界面
4.     HelpInfo();
5.
6.     int c;
7.     cin >> c;
8.     if (c == 2 || c == 7)
9.     { //退出
10.        return 0;
11.    }
12.
13.    else if (c == 1)
14.    { //连接服务器（省略）
15.        if (flag == 1)
16.        {
17.            continue;
18.        }
19.        else
20.        {
21.            freeaddrinfo(result);
22.            if (connect_socket == INVALID_SOCKET)
23.            {
24.                cout << "Unable to connect to server!" << endl;
25.                WSACleanup();
26.                continue;
27.            }
28.            else
29.            { //连接成功
30.
31.                //获得客户端信息（省略）
32.                //获得现在的 ip 和 port（省略）
33.
34.                thread(ReceivePacket, move(connect_socket)).detach(); //接收
                数据
35.
```

```
36.         while (1)
37.         {
38.             HelpInfo();
39.             int a;
40.             cin >> a;
41.             if (a == 2)
42.             { //请求结束连接
43.                 closeConnect(connect_socket);
44.                 break;
45.             }
46.             else if (a == 3)
47.             { //请求获得时间
48.                 getTime(send_buf, connect_socket);
49.                 continue;
50.             }
51.             else if (a == 4)
52.             { //请求获得服务器名字
53.                 getName(send_buf, connect_socket);
54.                 continue;
55.             }
56.             else if (a == 5)
57.             { //请求获得客户端列表
58.                 getList(send_buf, connect_socket);
59.                 continue;
60.             }
61.             else if (a == 6)
62.             { //请求向其它客户端发送信息
63.                 sendMessage(send_buf, connect_socket);
64.                 continue;
65.             }
66.             else if (a == 7)
67.             {
68.                 dropout(connect_socket);
69.                 return 0;
70.             }
71.             else
72.             {
73.                 continue;
74.             }
75.         }
76.         continue;
77.     }
78. }
79. }
```



```

80.
81.     else
82.     { //如果在尚未连接服务端时就企图发送请求
83.         cout << "You haven't connect to a server!" << endl;
84.         continue;
85.     }
86. }

```

- 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. int ReceivePacket(SOCKET connect_socket)
2. {
3.     do
4.     {
5.         result = recv(connect_socket, recv_buf, recv_buflen, 0); //接收数据，
接收到的数据包存在 recv_buf 中
6.         if (result > 0) //接收成功，
返回接收到的数据长度
7.         {
8.             //接收到的数据包类型 (packet_type)
9.             //接收到的数据包内容 (response_content)
10.
11.             switch (packet_type)
12.             {
13.                 case 'T':
14.                     Output(response_content);
15.                     Time.notify_one();
16.                     break;
17.                 case 'N':
18.                     Output(response_content);
19.                     ;
20.                     name.notify_one();
21.                     break;
22.                 case 'L':
23.                     Output(response_content);
24.                     list.notify_one();
25.                     break;
26.                 case 'M':
27.                     switch (type) //type: Message 中具体的类型 (Y: 发送成功, N: 发送
失败, M: 收到信息)
28.                     {

```

```

29.             case 'Y':
30.                 /*.....省略.....*/
31.                 Output(response_content);
32.                 message.notify_one();
33.                 break;
34.             case 'N':
35.                 Output(response_content);
36.                 message.notify_one();
37.                 break;
38.             case 'M':
39.                 /*.....省略.....*/
40.                 mtx_output.lock();
41.                 cout << "Message from " << to_string(from_ID) << " " <<
    from_IP << endl;
42.                 cout << receive_message << endl;
43.                 mtx_output.unlock();
44.                 break;
45.             }
46.         }
47.     }
48.     else if (result == 0) //连接结束
49.         printf("Connection closed\n");
50.     else //连接失败
51.         printf("recv failed with error: %d\n", WSAGetLastError());
52.
53. } while (result > 0);
54. return 0;
55. }

```

- 服务器初始运行后显示的界面

```

Welcome! Here is a server created by 3170102473 & 3170104579
Server initilizing...
Success! Waiting for clients...

```

如图所示，服务器启动后会显示欢迎信息和服务器初始化信息。初始化成功后，服务器将持续监听请求。

- 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）

```

1.     //绑定:
2.     ret = bind(sListen, (struct sockaddr*) & saServer, sizeof(saServer));
3.     if (ret == SOCKET_ERROR){

```

```

4.     printf("bind() failed! code:%d\n", WSAGetLastError());
5.     closesocket(sListen); //关闭套接字
6.     WSACleanup();
7.     ch = getchar();
8.     return 0;
9. }
10. //侦听连接请求,设置连接等待队伍长度为 5
11. ret = listen(sListen, 5);
12. if (ret == SOCKET_ERROR){
13.     printf("listen() failed! code:%d\n", WSAGetLastError());
14.     closesocket(sListen); //关闭套接字
15.     WSACleanup();
16.     ch = getchar();
17.     return 0;
18. }
19. printf("Success! Waiting for clients...\n");
20. //循环调用 accept 来等待客户端
21. for (;;) {
22.     sClient = accept(sListen, NULL, NULL);
23.     if (sClient == INVALID_SOCKET) { // accept 出错
24.         printf("accept() failed! code:%d\n", WSAGetLastError());
25.         closesocket(sClient);
26.         WSACleanup();
27.         ch = getchar();
28.         return 0;
29.     }
30.     //创建一个新的子线程,并以引用的方式传入 socket
31.     thread(ClientThread, ref(sClient)).detach();
32. }

```

- 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. // 获取线程编号
2. thread::id thread_id = this_thread::get_id();
3. // 获取客户端的 IP 与端口
4. getpeername(sClient, (struct sockaddr*) & client_info, &addrsz);
5. IP = inet_ntoa(client_info.sin_addr);
6. port = client_info.sin_port;
7. printf("Connected to client: %s\n", IP);
8. //创建一个新的 Client 数据
9. //将这个 Client 加入现有的列表,在写入数据前加锁
10. thread_lock.lock();
11. client_list.push_back(this_client);
12. thread_lock.unlock(); //写完数据后解锁

```

```

13.         //循环调用 recv()等待客户端的请求数据包
14.     int ret = 0;
15.     bool is_close = false;
16.     while (!is_close) {
17.         ZeroMemory(rec_packet, DEFAULT_BUFLen);
18.         ret = recv(sClient, rec_packet, DEFAULT_BUFLen, 0);
19.         if (ret > 0) { //正常
20.             printf("Packet received! Processing...\n");
21.             //处理 T,N,L 类型请求,即请求时间、服务器姓名、客户端列表
22.             int flag = ProcessRequestPacket(sClient, rec_packet);
23.             if (flag==0) { //处理 M 类型请求,即发送信息
24.                 // 向指定客户端发送信息
25.                 ...细节省略...
26.                 if (isAlive(to_id)) {
27.                     SOCKET sDestination = SearchSocket(to_id); // 获取目标客
客户端的 socket
28.                     //封装指示数据包,转发信息
29.                     ...细节省略...
30.                     SendString(sDestination, response);
31.                     printf("Send a message to destination client!\n");
32.                     //发送响应数据包,表示发送成功
33.                     ...细节省略...
34.                     SendString(sClient, response);
35.                     printf("Send a message to start client!\n");
36.                 }
37.                 else {
38.                     // 发送响应数据包,表示发送失败
39.                     string response = "#M-N*Destination doesn't exist!#";
40.                     SendString(sClient, response);
41.                     printf("Send message failed!\n");
42.                 }
43.             }
44.         }
45.         else if (ret == 0) { //连接中断
46.             printf("Connection closed!\n");
47.             is_close = true;
48.         }
49.         else { //运行出错
50.             printf("recv() failed! code:%d\n", WSAGetLastError());
51.             closesocket(sClient);
52.             WSACleanup();
53.             return;
54.         }
55.     }

```

● 客户端选择连接功能时，客户端和服务端显示内容截图。

1. 客户端

```
1
please input the server ip:
10.180.188.1
usage: 10.180.188.1 server-name
Connection succeed!
current IP: 10.180.188.1
```

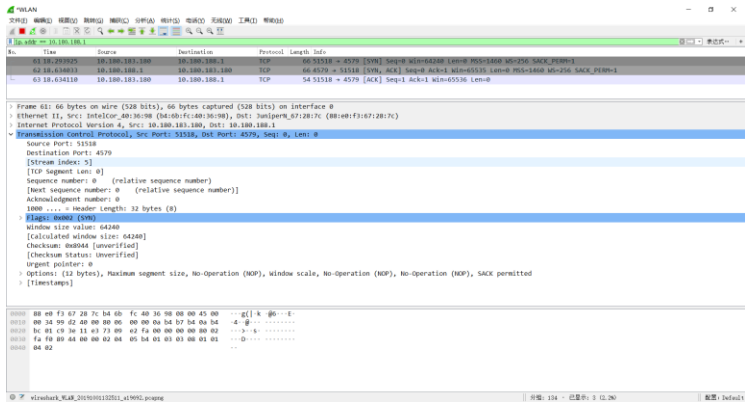
选择连接功能后，用户需输入目标服务器的 IP 地址。连接成功后，显示当前连接到的服务器 IP。

2. 服务端

```
Connected to client: 10.180.183.180
```

服务端成功处理连接请求后，会在屏幕上打印该连接的客户端 IP。

3. Wireshark 抓取的数据包截图



在连接过程中，共有三次数据包的发送，符合 TCP 协议。

● 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。

1. 客户端

```
3
Tue Oct 1 13:26:59 2019
```

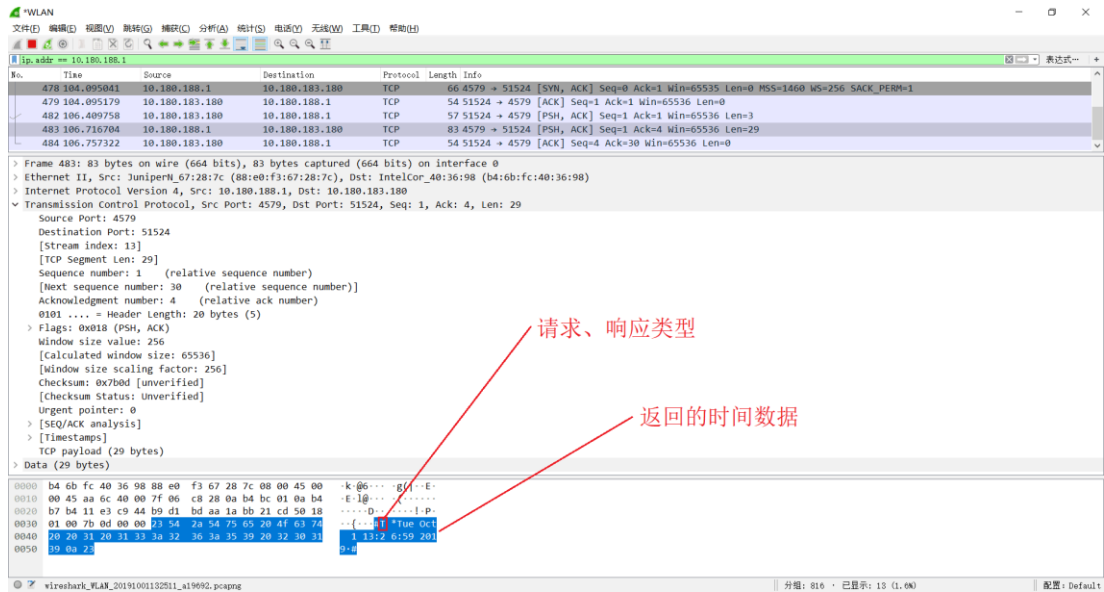
客户端输入指令 3，获取当前时间，打印在屏幕上。

2. 服务端

```
Packet received! Processing...
Time send.
```

服务端收到客户端发来的请求数据包，进行处理后，发送了时间信息。

3. 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）



可以看到，从服务端发往客户端的响应数据包中包含我们自定义的数据包边界符号，响应数据包指示头，以及当前时间信息。

● 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。

1. 客户端

```
4
2473-13107FF
```

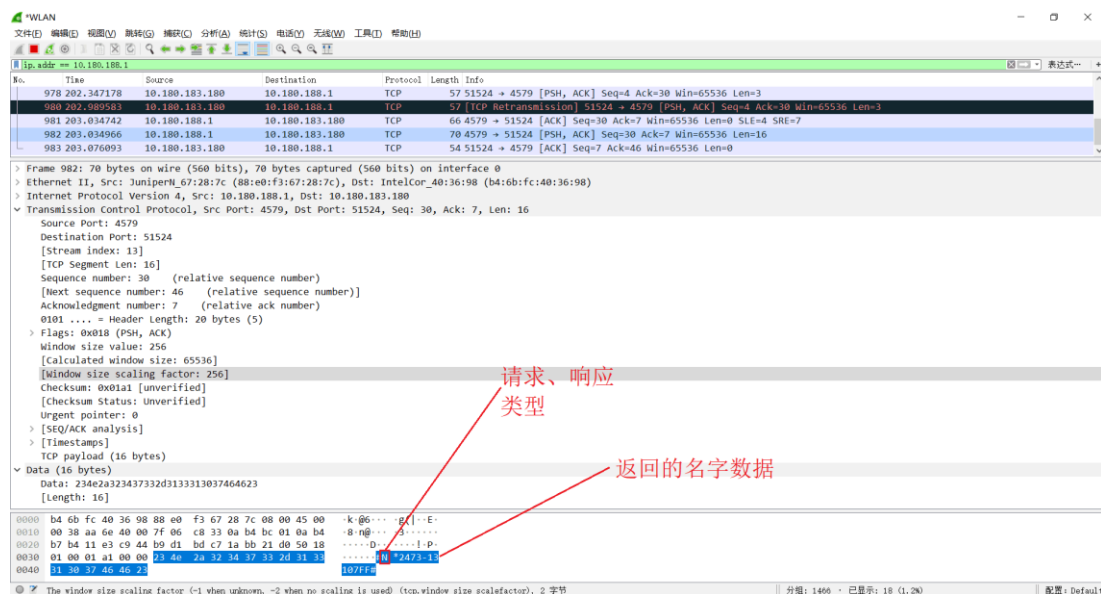
客户端输入指令 4，获取当前服务器的名字，打印在屏幕上。

2. 服务端

```
Packet received! Processing...
Server name send.
```

服务端收到客户端发来的请求数据包，进行处理后，发送了名字信息。

3. 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）



可以看到，从服务端发往客户端的响应数据包中包含我们自定义的数据包边界符号，响应数据包指示头，以及当前名字信息。

4. 相关的服务器的处理代码片段：

```
1. // 处理获取名字请求
2. void GetServerName(SOCKET s) {
3.     string response = "#N*";
4.     response += SERVER_NAME;
5.     response += "#\0";
6.     SendString(s, response);
7.     printf("Server name send.\n");
8. }
9. // 传入 string 类型的数据包，通过 SOCKET s 发送出去
10. void SendString(SOCKET s, string str) {
11.     char content[DEFAULT_BUFLen];
12.     strcpy(content, str.c_str());
13.     int len = str.length();
14.     send(s, content, len, 0);
15. }
```

● 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。

在这里我们让服务端的 PC 启动了客户端程序，从本机 IP 连接到本机服务器，这样列表中就有了两台客户端。

1. 客户端

```
5
0 10.180.183.180
1 10.180.188.1
```

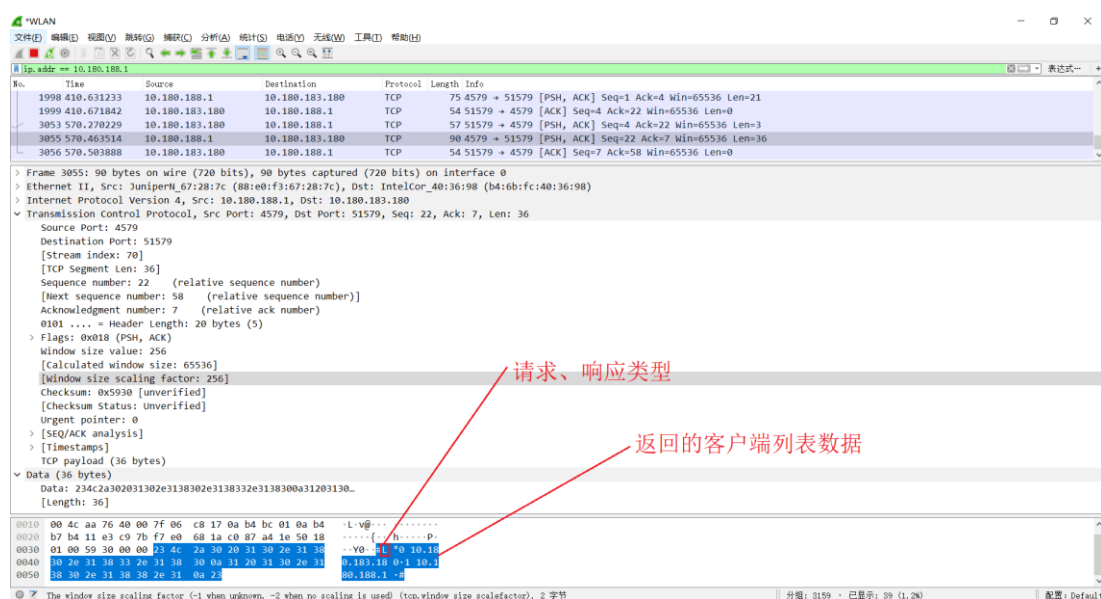
用户输入指令 5，获取当前在线的客户端列表。

2. 服务端

```
Packet received! Processing...
Client list send.
```

服务端收到客户端发来的请求数据包，进行处理后，发送了客户端列表信息。

3. 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）



可以看到，从服务端发往客户端的响应数据包中包含我们自定义的数据包边界符号，响应数据包指示头，以及客户端列表信息。其中客户端列表信息以编号和 IP 的顺序依次排列。

4. 相关的服务器的处理代码片段：

```
1. void GetClientList(SOCKET s) {
2.     string response = "#L*";
3.     thread_lock.lock();
4.     vector<struct Client*>::iterator it;
5.     for (it = client_list.begin(); it != client_list.end(); it++) {
6.         response += to_string((*it)->id);
7.         response += ' ';
8.         response += (*it)->IP;
9.         response += '\n';
10.    }
```



```

11.     response += "#\0";
12.     thread_lock.unlock();
13.     SendString(s, response);
14.     printf("Client list send.\n");
15. }

```

● 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。

1. 发送消息的客户端

```

6
please input the destination id:
1
please input the message:
Hello world!
the message has been sent to 1

```

用户输入指令 6，输入目标客户端的编号（在上一功能中获取的客户端列表里有各个客户端的编号），以及要发送的消息内容。发送后屏幕上会有确认信息。

2. 服务器

```

Packet received! Processing...
Send a message to destination client!
Send a message to start client!

```

服务器接收到客户端发来的请求数据包，进行解析处理后，向目的地客户端发送一个指示数据包，向源客户端发送一个响应数据包表示转发成功或者失败。

3. 接收消息的客户端

```

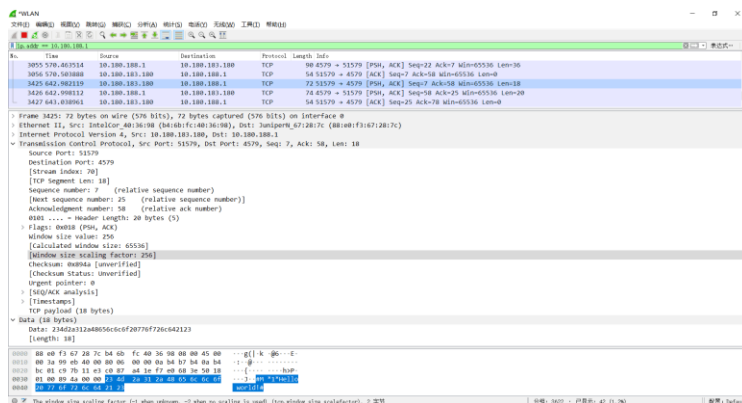
Message from 0 10.180.183.180
Hello world!

```

目标客户端会收到服务器转发来的指示数据包，解析后将消息发送方的编号与 IP 和消息内容打印在屏幕上。

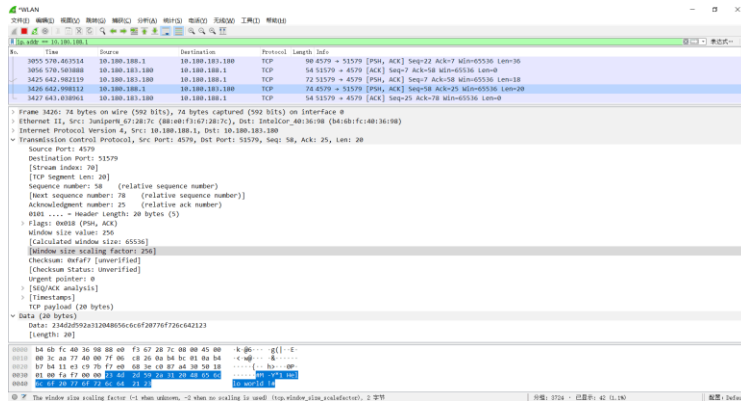
4. Wireshark 抓取的数据包截图（发送和接收分别标记）

a. 发送信息的客户端发送给服务端的数据包



该数据包中包含目的地与消息内容。

b. 服务端发送给原客户端表示成功收到信息



该数据包中包含了发送成功的指示头，以及消息内容。

c. 接收信息的客户端与服务端在同一个 PC 上，无法抓取数据包

5. 相关的服务器的处理代码片段

```
1. if (flag==0) { //处理 M 类型请求，即发送信息
2.     // 向指定客户端发送信息
3.     string str = rec_packet; //"#M*0*haha#"
4.
5.     int pos1, pos2;
6.     pos1 = str.find("*") + 1;
7.     pos2 = str.find("#", pos1);
8.     int to_id = stoi(str.substr(pos1, pos2 - pos1));
9.
10.    pos1 = pos2 + 1;
11.    pos2 = str.find("#", pos1);
12.    string message = str.substr(pos1, pos2 - pos1);
13.    //cout << to_id << ' ' << message << endl;
14.    if (isAlive(to_id)) {
15.        SOCKET sDestination = SearchSocket(to_id); // 获取目标客户端的
socket
16.        //封装指示数据包，转发信息
17.        string response = "#M-M*";
18.        response += to_string(this_client->id);
19.        response += " ";
20.        response += this_client->IP;
21.        response += "\n";
22.        response += message;
23.        response += "#\0";
24.        SendString(sDestination, response);
25.        printf("Send a message to destination client!\n");
```

```

26.         //发送响应数据包, 表示发送成功
27.         response = "#M-Y*";
28.         response += to_string(to_id);
29.         response += " ";
30.         response += message;
31.         response += "#\0";
32.         SendString(sClient, response);
33.         printf("Send a message to start client!\n");
34.     }
35.     else {
36.         // 发送响应数据包, 表示发送失败
37.         string response = "#M-N*Destination doesn't exist!#";
38.         SendString(sClient, response);
39.         printf("Send message failed!\n");
40.     }
41. }

```

6. 相关的客户端（发送和接收消息）处理代码片段：

a. 发送消息

```

7.     else if (a == 6)
8.     { //请求向其它客户端发送信息
9.         string sendMessage;
10.        char buffer[1024] = "\0";
11.        int dst_ID = 0;
12.
13.        //客户端的 ID
14.        cout << "please input the destination id:" << endl;
15.        cin >> dst_ID;
16.        cin.getline(buffer, 1024);
17.
18.        //要发送的消息内容
19.        cout << "please input the message:" << endl;
20.        cin.getline(buffer, 1024);
21.        sendMessage = buffer;
22.
23.        //发送的数据包
24.        string packet = "#M*";
25.        packet += to_string(dst_ID);
26.        packet += "*";
27.        packet += sendMessage;
28.        packet += "#";
29.        ZeroMemory(send_buf, DEFAULT_BUFLen);

```

```

30.     memcpy(send_buf, packet.c_str(), packet.size());
31.     ret = send(connect_socket, send_buf, (int)strlen(send_buf), 0);
32.     if (ret == SOCKET_ERROR)
33.     {
34.         cout << "send failed with error: " << to_string(WSAGetLastError()) << endl;
35.         closesocket(connect_socket);
36.         WSACleanup();
37.         break;
38.     }
39.     unique_lock<mutex> lck(mtx_message);
40.     message.wait(lck);
41.     continue;
42. }

```

b. 接收消息

```

1. //接收消息
2.
3. case 'M':
4.     pos1 = recv_packet.find("-") + 1;
5.     char type = recv_packet[pos1];
6.     switch (type)
7.     {
8.     case 'Y':
9.         //接收信息的一方的编号
10.        int request_ID;
11.        pos1 = recv_packet.find("*") + 1;
12.        pos2 = recv_packet.find(" ", pos1);
13.        request_ID = stoi(recv_packet.substr(pos1, pos2 - pos1)); //将字符串转化为数字
14.        response_content = "the message has been sent to ";
15.        response_content += to_string(request_ID);
16.        Output(response_content);
17.        message.notify_one();
18.        break;
19.     case 'N':
20.        Output(response_content);
21.        message.notify_one();
22.        break;
23.     case 'M':
24.         //发送信息的客户端 ID
25.        int from_ID;

```

```

26.         pos1 = recv_packet.find("*") + 1;
27.         pos2 = recv_packet.find(" ", pos1);
28.         from_ID = stoi(recv_packet.substr(pos1, pos2 - pos1));
29.
30.         //发送信息的客户端 IP
31.         string from_IP;
32.         pos1 = pos2 + 1;
33.         pos2 = recv_packet.find("\n");
34.         from_IP = recv_packet.substr(pos1, pos2 - pos1);
35.
36.         //收到的信息
37.         string receive_message;
38.         pos1 = pos2 + 1;
39.         pos2 = recv_packet.find("#", pos1);
40.         receive_message = recv_packet.substr(pos1, pos2 - pos1);
41.
42.         mtx_output.lock();
43.         cout << "Message from " << to_string(from_ID) << " " << from_IP << endl;
44.
45.         cout << receive_message << endl;
46.         mtx_output.unlock();
47.         break;
48.     }

```

- 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。

客户端并未发出 TCP 连接释放的消息，服务端的 TCP 连接状态没有改变。服务端始终维持这一无用的子线程。这是因为客户端在退出前没有发送断开连接的请求。

- 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？
1. 查看之前异常退出的连接，发现依然存在

```

5
0 10.180.183.180
1 10.180.188.1
2 10.180.183.180

```

2. 选择给这个之前异常退出的客户端连接发送消息，服务端仍然会发回成功的响应数据包。但事实上消息并不能成功送达。

```
6
please input the destination id:
0
please input the message:
Are you ok?
the message has been sent to 0
```

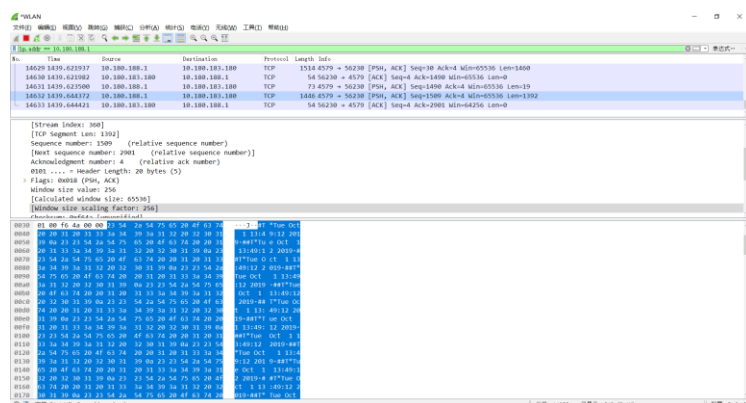
- 修改获取时间功能，改为用户选择 1 次，程序内自动发送 100 次请求。服务器是否正常处理了 100 次请求，截取客户端收到的响应（通过程序计数一下是否有 100 个响应回来），并使用 Wireshark 抓取数据包，观察实际发出的数据包个数。

修改了服务器代码，使得服务器收到一次要求时间的请求数据包，就自动发送 100 次包含时间的响应数据包。然而在实际运行中，我们发现客户端只收到了三个包。

```
N: 2
Tue Oct 1 13:49:12 2019

N: 3
Tue Oct 1 13:49:12 2019
```

使用 Wireshark 抓取数据包，发现服务端只发出了三个包。其中两个包数据量很大，里面包含多个数据包的内容，可以看到当中有很多个数据包边界标识符。



我们猜测是服务器一次性发送了 100 个响应数据包，但 send 函数的处理并没有这么快，以至于多个包含时间的响应数据包被堆积在一起打包发出。因此，我们在服务端的时间处理函数中加入了 Sleep(500) 函数语句，使服务端每发出一个时间数据包，就等待 500 毫秒再执行，给予了充分的响应时间后，客户端可以收到全部 100 个数据包。

1. 客户端收到 100 个数据包的截图

```
N: 91
Tue Oct 1 11:02:25 2019

N: 92
Tue Oct 1 11:02:26 2019

N: 93
Tue Oct 1 11:02:26 2019

N: 94
Tue Oct 1 11:02:27 2019

N: 95
Tue Oct 1 11:02:27 2019

N: 96
Tue Oct 1 11:02:28 2019

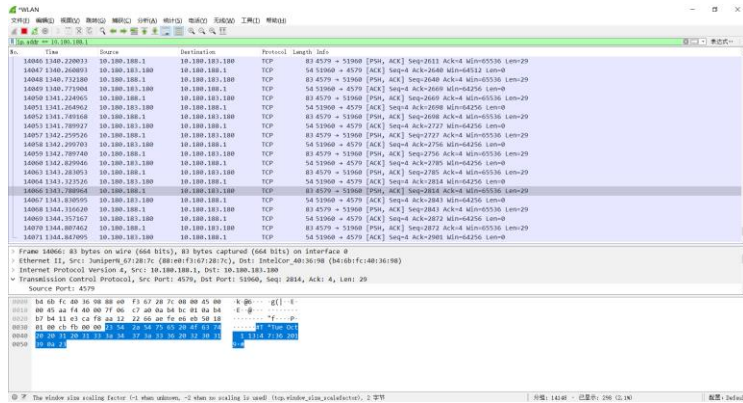
N: 97
Tue Oct 1 11:02:28 2019

N: 98
Tue Oct 1 11:02:29 2019

N: 99
Tue Oct 1 11:02:29 2019

N: 100
Tue Oct 1 11:02:30 2019
```

2. 使用 Wireshark 抓取数据包，观察实际发出的数据包个数



- 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

1. 两个客户端的 ID 和 IP

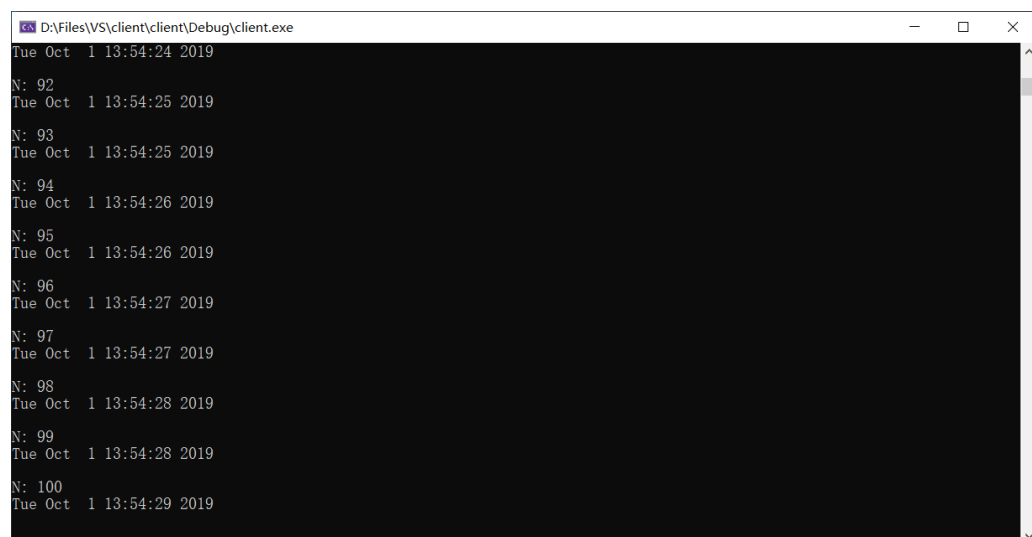
```
5
0 10.180.183.180
1 10.180.188.1
```

2. 服务器

```
191
Time send.
Packet received! Processing...
192
Time send.
Packet received! Processing...
193
Time send.
Packet received! Processing...
194
Time send.
Packet received! Processing...
195
Time send.
Packet received! Processing...
196
Time send.
Packet received! Processing...
197
Time send.
Packet received! Processing...
198
Time send.
Packet received! Processing...
199
Time send.
Packet received! Processing...
200
Time send.
```

3. 两个客户端

a. 客户端 1 (ID 为 0)



The screenshot shows a Windows command prompt window titled "D:\Files\VS\client\client\Debug\client.exe". The window displays a series of network communication logs. Each log entry consists of a line starting with "N:" followed by a number, and a second line showing a timestamp "Tue Oct 1 13:54:24 2019" (or similar). The numbers in the "N:" field range from 92 to 100. The timestamps are in the format "Tue Oct 1 13:54:24 2019".

```
D:\Files\VS\client\client\Debug\client.exe
Tue Oct 1 13:54:24 2019
N: 92
Tue Oct 1 13:54:25 2019
N: 93
Tue Oct 1 13:54:25 2019
N: 94
Tue Oct 1 13:54:26 2019
N: 95
Tue Oct 1 13:54:26 2019
N: 96
Tue Oct 1 13:54:27 2019
N: 97
Tue Oct 1 13:54:27 2019
N: 98
Tue Oct 1 13:54:28 2019
N: 99
Tue Oct 1 13:54:28 2019
N: 100
Tue Oct 1 13:54:29 2019
```

b. 客户端 2 (ID 为 1)



```
C:\Users\zht\Desktop\client (1).exe
Tue Oct 1 13:54:24 2019
N: 92
Tue Oct 1 13:54:24 2019
N: 93
Tue Oct 1 13:54:25 2019
N: 94
Tue Oct 1 13:54:25 2019
N: 95
Tue Oct 1 13:54:26 2019
N: 96
Tue Oct 1 13:54:26 2019
N: 97
Tue Oct 1 13:54:27 2019
N: 98
Tue Oct 1 13:54:27 2019
N: 99
Tue Oct 1 13:54:28 2019
N: 100
Tue Oct 1 13:54:28 2019
```

两个客户端都能收到 100 个时间响应包。

五、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。在实验调试中我们发现每一次客户端连接到服务端之后，都会被自动分配到一个随机端口，即使是一个客户端多次连接到同一服务端，每次连接时的端口也都会有变化。因为每次调用 connect 时，系统内核都会为客户端分配一个空闲端口。

- 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。

listen() 函数的主要作用就是将套接字变成被动的连接监听套接字（被动等待客户端的连接）。listen() 函数不会阻塞，它主要做的事情为，将该套接字和套接字对应的连接队列长度告诉内核，然后，listen() 函数就结束。

这样的话，当有一个客户端主动连接（connect()），内核就自动完成 TCP 三次握手，将建立好的链接自动存储到队列中，如此重复。

- 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，出现了丢包现象。

- 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？

因为一个 socket 结构体中不仅仅记录了本地的 IP 和端口号，还有目的地的 IP 和端口号。在 send 函数中传入了 socket 标识符，因此可以确定连接双方的具体信息。

- 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)

我们在进行本项测试时时与前面的测试不在同一天进行，两台电脑的 ip 地址发生了一些变化，客户端为 10.180.176.105，服务端地址为 10.180.176.119

1. 客户端正常连接时：

活动连接				
协议	本地地址	外部地址	状态	
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5700	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:9012	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:11200	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:16422	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:16423	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:23938	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49669	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:50001	0.0.0.0:0	LISTENING	
TCP	10.180.176.105:139	0.0.0.0:0	LISTENING	
TCP	10.180.176.105:52615	52.139.250.253:443	ESTABLISHED	
TCP	10.180.176.105:52619	39.107.190.67:443	ESTABLISHED	
TCP	10.180.176.105:52647	35.174.127.31:443	ESTABLISHED	
TCP	10.180.176.105:52716	47.95.42.129:443	ESTABLISHED	
TCP	10.180.176.105:53848	218.206.90.173:80	CLOSE_WAIT	
TCP	10.180.176.105:53849	218.206.90.173:80	CLOSE_WAIT	
TCP	10.180.176.105:53850	218.206.90.173:80	CLOSE_WAIT	
TCP	10.180.176.105:53852	52.139.250.253:443	ESTABLISHED	
TCP	10.180.176.105:61471	218.206.90.180:443	CLOSE_WAIT	
TCP	10.180.176.105:61629	151.101.40.133:443	CLOSE_WAIT	
TCP	10.180.176.105:61630	151.101.40.133:443	CLOSE_WAIT	
TCP	10.180.176.105:61757	112.13.64.14:80	CLOSE_WAIT	
TCP	10.180.176.105:61761	3.223.95.60:443	ESTABLISHED	
TCP	10.180.176.105:61763	39.156.41.9:80	CLOSE_WAIT	
TCP	10.180.176.105:61764	112.13.64.14:80	CLOSE_WAIT	
TCP	10.180.176.105:61765	112.34.111.26:443	CLOSE_WAIT	
TCP	10.180.176.105:61766	10.180.176.119:4579	ESTABLISHED	
TCP	10.180.176.105:61769	13.107.18.11:443	ESTABLISHED	
TCP	10.180.176.105:61770	202.89.233.101:443	ESTABLISHED	
TCP	10.180.176.105:61772	13.107.3.254:443	ESTABLISHED	
TCP	10.180.176.105:61773	204.79.197.254:443	ESTABLISHED	
TCP	10.180.176.105:61774	13.107.136.254:443	ESTABLISHED	

2. 客户端正常退出后：

活动连接

协议	本地地址	外部地址	状态
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5700	0.0.0.0:0	LISTENING
TCP	0.0.0.0:9012	0.0.0.0:0	LISTENING
TCP	0.0.0.0:11200	0.0.0.0:0	LISTENING
TCP	0.0.0.0:16422	0.0.0.0:0	LISTENING
TCP	0.0.0.0:16423	0.0.0.0:0	LISTENING
TCP	0.0.0.0:23938	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49669	0.0.0.0:0	LISTENING
TCP	0.0.0.0:50001	0.0.0.0:0	LISTENING
TCP	10.180.176.105:139	0.0.0.0:0	LISTENING
TCP	10.180.176.105:52615	52.139.250.253:443	ESTABLISHED
TCP	10.180.176.105:52619	39.107.190.67:443	ESTABLISHED
TCP	10.180.176.105:52647	35.174.127.31:443	ESTABLISHED
TCP	10.180.176.105:52716	47.95.42.129:443	ESTABLISHED
TCP	10.180.176.105:53848	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53849	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53850	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53852	52.139.250.253:443	ESTABLISHED
TCP	10.180.176.105:61471	218.206.90.180:443	CLOSE_WAIT
TCP	10.180.176.105:61761	3.223.95.60:443	ESTABLISHED
TCP	10.180.176.105:61766	10.180.176.119:4579	TIME_WAIT
TCP	10.180.176.105:61776	40.90.185.223:443	ESTABLISHED
TCP	10.180.176.105:61794	39.156.41.9:80	CLOSE_WAIT
TCP	10.180.176.105:61796	112.13.64.14:80	CLOSE_WAIT
TCP	10.180.176.105:61797	112.34.111.26:443	CLOSE_WAIT
TCP	10.180.176.105:61798	185.246.154.33:443	TIME_WAIT
TCP	10.180.176.105:61799	112.13.64.14:80	CLOSE_WAIT
TCP	10.180.176.105:61800	183.232.89.115:80	CLOSE_WAIT

TIME_WAIT 这个状态保持了不到 1 分钟就不复存在了。

3. 过了一分钟后的客户端：

活动连接

协议	本地地址	外部地址	状态
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5700	0.0.0.0:0	LISTENING
TCP	0.0.0.0:9012	0.0.0.0:0	LISTENING
TCP	0.0.0.0:11200	0.0.0.0:0	LISTENING
TCP	0.0.0.0:16422	0.0.0.0:0	LISTENING
TCP	0.0.0.0:16423	0.0.0.0:0	LISTENING
TCP	0.0.0.0:23938	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49669	0.0.0.0:0	LISTENING
TCP	0.0.0.0:50001	0.0.0.0:0	LISTENING
TCP	10.180.176.105:139	0.0.0.0:0	LISTENING
TCP	10.180.176.105:52615	52.139.250.253:443	ESTABLISHED
TCP	10.180.176.105:52619	39.107.190.67:443	ESTABLISHED
TCP	10.180.176.105:52647	35.174.127.31:443	ESTABLISHED
TCP	10.180.176.105:52716	47.95.42.129:443	ESTABLISHED
TCP	10.180.176.105:53848	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53849	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53850	218.206.90.173:80	CLOSE_WAIT
TCP	10.180.176.105:53852	52.139.250.253:443	ESTABLISHED
TCP	10.180.176.105:61471	218.206.90.180:443	CLOSE_WAIT
TCP	10.180.176.105:61761	3.223.95.60:443	ESTABLISHED
TCP	10.180.176.105:61776	40.90.185.223:443	ESTABLISHED
TCP	10.180.176.105:61799	112.13.64.14:80	CLOSE_WAIT
TCP	10.180.176.105:61804	39.156.41.9:80	CLOSE_WAIT
TCP	10.180.176.105:61805	112.13.64.14:80	CLOSE_WAIT
TCP	10.180.176.105:61810	203.208.39.238:443	ESTABLISHED
TCP	10.180.176.105:61811	112.34.111.26:443	CLOSE_WAIT
TCP	10.180.176.105:61817	183.232.93.214:80	ESTABLISHED
TCP	127.0.0.1:4300	0.0.0.0:0	LISTENING
TCP	127.0.0.1:4301	0.0.0.0:0	LISTENING
TCP	127.0.0.1:8088	0.0.0.0:0	LISTENING
TCP	127.0.0.1:8884	0.0.0.0:0	LISTENING
TCP	127.0.0.1:9901	0.0.0.0:0	LISTENING

可以看到，原有的连接状态已经不复存在。

- 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. 正常连接下的服务端：

活动连接				
协议	本地地址	外部地址	状态	
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:4579	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:7680	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49670	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:50187	0.0.0.0:0	LISTENING	
TCP	10.180.176.119:139	0.0.0.0:0	LISTENING	
TCP	10.180.176.119:4579	10.180.176.105:62867	ESTABLISHED	
TCP	10.180.176.119:58081	120.241.186.161:80	CLOSE_WAIT	
TCP	10.180.176.119:58082	120.241.190.227:80	CLOSE_WAIT	
TCP	10.180.176.119:58083	120.241.190.104:80	CLOSE_WAIT	
TCP	10.180.176.119:58086	40.90.185.223:443	ESTABLISHED	
TCP	10.180.176.119:58087	47.95.42.129:443	ESTABLISHED	
TCP	10.180.176.119:58089	112.13.87.219:443	TIME_WAIT	
TCP	10.180.176.119:58092	52.109.12.18:443	ESTABLISHED	
TCP	10.180.176.119:64467	39.107.190.67:443	ESTABLISHED	
TCP	10.180.176.119:64468	40.90.189.152:443	ESTABLISHED	
TCP	10.180.176.119:64475	223.252.199.69:6003	ESTABLISHED	
TCP	10.180.176.119:64482	117.184.242.106:443	ESTABLISHED	
TCP	10.180.176.119:64488	40.90.189.152:443	ESTABLISHED	
TCP	10.180.176.119:64491	108.177.97.188:5228	ESTABLISHED	
TCP	10.180.176.119:64509	35.174.127.31:443	ESTABLISHED	
TCP	10.180.176.119:64625	104.74.21.227:443	CLOSE_WAIT	

2. 客户端异常退出后的服务端：

活动连接				
协议	本地地址	外部地址	状态	
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:4579	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:7680	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:49670	0.0.0.0:0	LISTENING	
TCP	0.0.0.0:50187	0.0.0.0:0	LISTENING	
TCP	10.180.176.119:139	0.0.0.0:0	LISTENING	
TCP	10.180.176.119:4579	10.180.176.105:62867	ESTABLISHED	
TCP	10.180.176.119:56943	10.10.0.21:53	TIME_WAIT	
TCP	10.180.176.119:56950	172.217.24.14:443	SYN_SENT	
TCP	10.180.176.119:56951	172.217.24.14:443	SYN_SENT	
TCP	10.180.176.119:58081	120.241.186.161:80	CLOSE_WAIT	
TCP	10.180.176.119:58082	120.241.190.227:80	CLOSE_WAIT	
TCP	10.180.176.119:58083	120.241.190.104:80	CLOSE_WAIT	
TCP	10.180.176.119:58086	40.90.185.223:443	ESTABLISHED	
TCP	10.180.176.119:58087	47.95.42.129:443	ESTABLISHED	

我们发现，客户端断网后异常退出后，服务器的 TCP 连接状态并没有发生什么变化。

3. 服务器该如何检测连接是否继续有效

为了避免网络突然中断，用户没来得及发送断开连接的请求，因此服务器始终维持线程、占用资源的状况发生，对用户端与服务器端的代码应进行优化改进。可以通过心跳包机制进行判断：客户端每隔一段时间（如 2 或 3s）向用户器发送一次请求，这个线程将在连接被中断的时候也会被一同杀死。同样的，服务器端也有对应的函数，不断监听来自客户端的包，如果超过一定时间（如 5s 后）没有收到包的话，则会断开连接与此客户端的连接。

五、 讨论、心得

在本此实验过程中，我们对于 socket 编程有了更深入的了解和认识，在编写这次程序之前，我们对 socket 通信几乎一无所知，也根本不了解服务器和客户端如何进行通信，在查阅资料学习的过程中，我们逐渐了解了多线程编程模式，了解了服务器和客户端之间如何进行通信、如何接受和发送数据包以及如何处理数据包。

在实验过程中，进行到修改获取时间功能即改户选择 1 次，程序内自动发送 100 次请求这一个步骤时，我们遇到了一些困难，我们使用 Wireshark 抓取数据包，发现服务端只发出了三个包。其中两个包数据量很大，里面包含多个数据包的内容，其中有很多个数据包边界标识符。我们猜测是服务器一次性发送了 100 个响应数据包，但 send 函数的处理并没有这么快，以至于多个包含时间的响应数据包被堆积在一起打包发出。因此，我们在服务端的时间处理函数中加入了 Sleep(500) 函数语句，使服务端每发出一个时间数据包，就等待 500 毫秒再执行，给予了充分的响应时间后，客户端可以收到全部 100 个数据包。

另外，在我们发现，客户端断网后异常退出后，服务器的 TCP 连接状态并没有发生什么变化，在服务器该如何检测连接是否继续有效上，我们想出了一个解决方案，即针对网络突然中断，用户没来得及发送断开连接的请求而导致服务器始终维持线程、占用资源的状况，可以对用户端与服务器端的代码应进行优化改进。可以通过心跳包机制进行判断：客户端每隔一段时间（如 2 或 3s）向用户器发送一次请求，这个线程将在连接被中断的时候也会被一同杀死。同样的，服务器端也有对应的函数，不断监听来自客户端的包，如果超过一定时间（如 5s 后）没有收到包的话，则会断开连接与此客户端的连接。

在编程的过程中，我们还遇到一个困难：在 Windows 平台下的 socket 编程和在 Linux 平台下的 socket 编程有些许出入，但我们能搜集到的资料和信息大多都是关于在 Linux 平台下的 socket 编程。同时，Linux 平台下的线程操作函数也和 Windows 平台下不太一样，以至于我们最初总是无法调用正确的函数。在通过大量查询资料和学习之后，我们克服了这一困难。

经过这次实验，我们有效地学习了 socket 编程，我们认为本次实验具有很好的实践意义，我们受益匪浅。