

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： 夏豪诚

学 院： 计算机学院

系： 计算机系

专 业： 信息安全

学 号： 3170102492

指导教师： 邱劲松

2019 年 12 月 24 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: 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

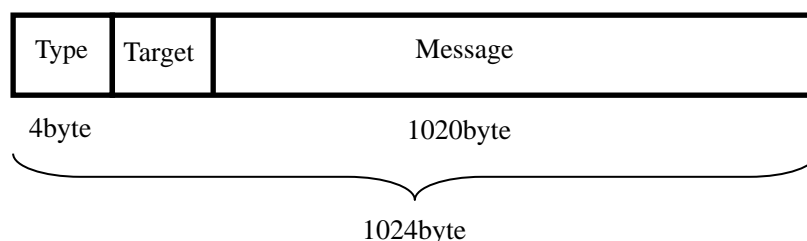
- 设计请求、指示（服务器主动发给客户端的）、响应数据包的格式，至少要考虑如下问题：
 - a) 定义两个数据包的边界如何识别
 - b) 定义数据包的请求、指示、响应类型字段
 - c) 定义数据包的长度字段或者结尾标记
 - d) 定义数据包内数据字段的格式（特别是考虑客户端列表数据如何表达）
- 小组分工：1 人负责编写服务端，1 人负责编写客户端
- 客户端编写步骤（需要采用多线程模式）
 - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
 - b) 编写一个菜单功能，列出 7 个选项
 - c) 等待用户选择
 - d) 根据用户选择，做出相应的动作（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



```
enum{
    TIME,
    NAME,
    LIST,
    SEND
};
```

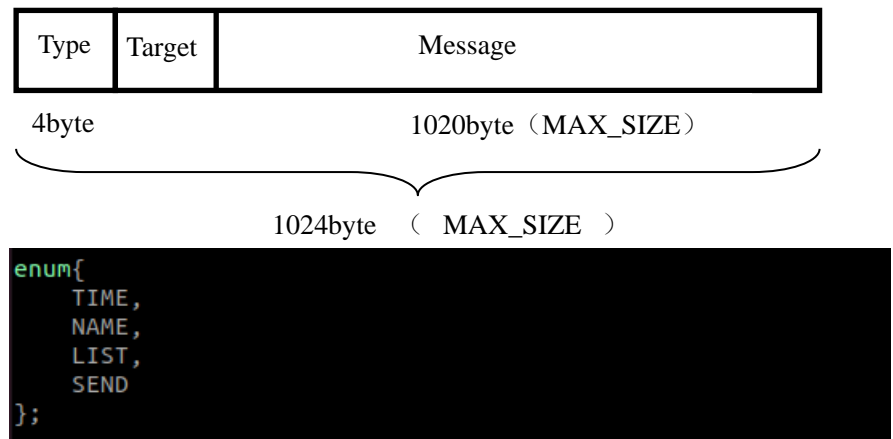
TIME 类型：client 向服务端请求时间

NAME 类型：client 向服务端请求名字信息

LIST 类型：client 向服务端请求客户端列表信息

SEND 类型： client 向服务端发送消息并指定目标服务端，由服务端发向指定的客户端。（注意，此时才存在 Target 段，并且 Target 段和 Message 段用一个':',即冒号字符分隔，其余情况 Message 段的大小均为 1020byte）

● 描述响应数据包的格式（画图说明），响应类型的定义



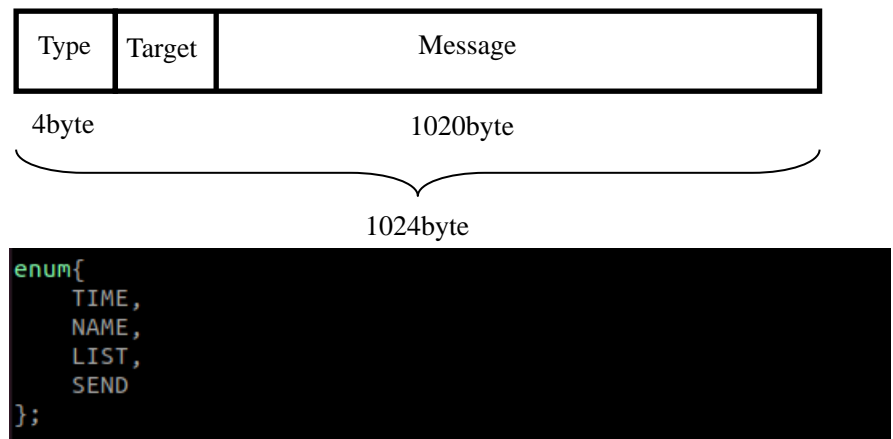
TIME 类型： 服务端返回时间

NAME 类型： 服务端返回名字信息

LIST 类型： 服务端返回客户端列表信息

SEND 类型： 服务端将客户端消息发往指定的客户端（注意，此时 Target 段已经消失）
（响应包的长度视实际有效内容大小而定）

● 描述指示数据包的格式（画图说明），指示类型的定义



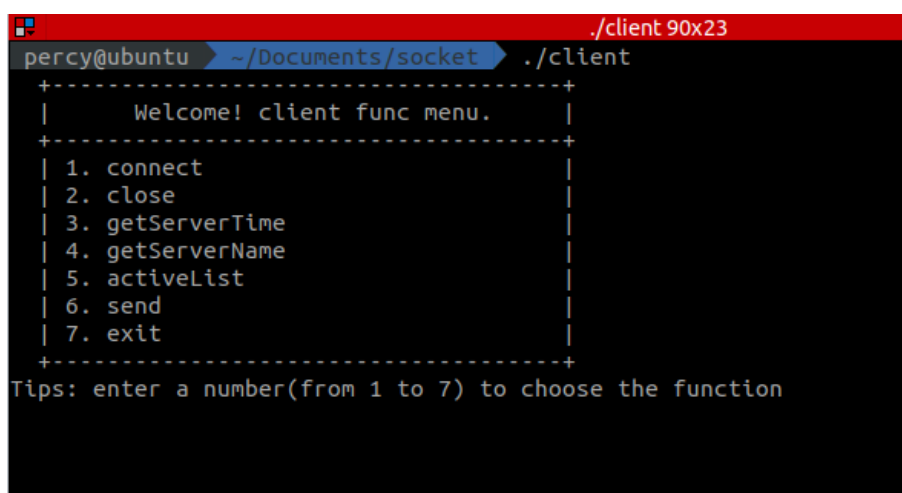
TIME 类型： client 向服务端请求时间

NAME 类型： client 向服务端请求名字信息

LIST 类型： client 向服务端请求客户端列表信息

SEND 类型: client 向服务端发送消息并指定目标服务端, 由服务端发向指定的客户端。(注意, 此时才存在 Target 段, 并且 Target 段和 Message 段用一个':', 即冒号字符分隔, 其余情况 Message 段的大小均为 1020byte)

- 客户端初始运行后显示的菜单选项

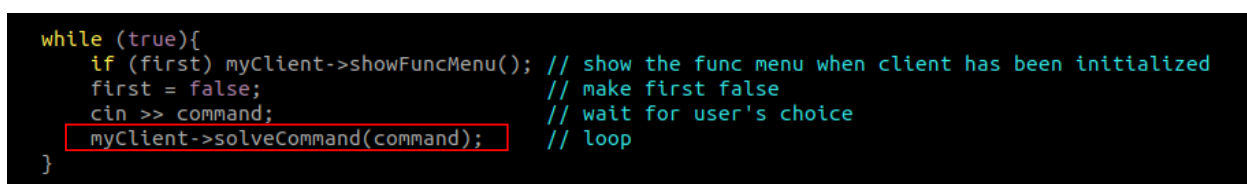


```
percy@ubuntu ~/Documents/socket ./client
+-----+
| Welcome! client func menu. |
+-----+
| 1. connect                  |
| 2. close                    |
| 3. getServerTime            |
| 4. getServerName            |
| 5. activeList                |
| 6. send                     |
| 7. exit                     |
+-----+
Tips: enter a number(from 1 to 7) to choose the function
```

图 4 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)

客户端的循环即为等待用户输出命令选择功能。关键代码如下:



```
while (true){
    if (first) myClient->showFuncMenu(); // show the func menu when client has been initialized
    first = false;                       // make first false
    cin >> command;                       // wait for user's choice
    myClient->solveCommand(command);       // loop
}
```

图 5 客户端主线程循环代码片段 1

而 solveCommand()函数的具体内容在下图展示, 即为根据输入的数据选择对应的功能函数的调用。

```

void client::solveCommand(int command){
    int recv;
    switch(command){
        case 1: cliConnect(); break;
        case 2: cliClose();
                cout<<"Connection has been closed"<<endl;
                break;
        case 3: recv = getServerTime();
                break;
        case 4: recv = getServerName();
                break;
        case 5: recv = activeList();
                break;
        case 6: recv = cliSend();
                break;
        case 7: if (sock == -1) exit(0);
                else
                {
                    cliClose();
                    sock = -1;
                    exit(0);
                }
                break;
        default: break;
    }
}
}

```

图 6 客户端主线程循环代码片段 2

- 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）

listenResponse()即为主线程通过 pthread_create()创建的线程的运行函数。可以看到我们使用 recv()函数实现数据的接收。

```

void *listenResponse(void *arg){
    int sock;
    char server_response[256];
    memset(server_response, 0, sizeof(server_response));
    sock = (int)*(int*)arg;

    // block
    while(true){
        int ret = recv(sock, &server_response, sizeof(server_response),0);
        if (ret > 0){
            cout<<"\n"
                <<"[Data received] \n"
                <<"The data is: \n"
                <<server_response
                <<endl;
            memset(server_response, 0, sizeof(server_response));
        }
        else if (ret < 0){
            // exception
            pthread_exit(NULL);
        }
        else if (ret == 0){
            cout<<"Error: Socket closed, exiting current receive thread"<<endl;
            pthread_exit(NULL);
        }
    }
    pthread_exit(NULL);
}

```

图 7 客户端接收数据子线程循环代码

- 服务器初始运行后显示的界面

由于监听端口已被设置为学号后四位，故不需要传入此参数。

```
#define SOCK_PORT 2492    // port number
```

```
percy@ubuntu ~/Documents/socket$ ./server
Server: serv_socket create success!
Server: bind success!
Server: listening...
```

图 8 服务器初始界面

- 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）

主线程在图中红色矩形框框起的代码处进行循环。

```
int main(){
    pthread_t tid;
    server *myServer = new server();
    myServer->init(tid);
    myServer->servBind();
    myServer->servListen();
    myServer->servCommunacation();
    return 0;
}
```

图 9 服务器的主线程循环关键代码片段 1

servCommunacation()函数中的循环部分如下，监听端口，当服务端监听端口出现新的客户端请求的时候，会根据收到的请求更新当前活动的客户端列表，并为不同的客户端创建线程进行服务（利用thread_id[]这一数组将客户端列表的id和服务端子线程的线程号对应）：


```

int server::servCommunication(){
    while(true){
        // accept

        clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,&clnt_addr_size);
        if (clnt_sock == -1){
            perror("Error: accept failed!\n");
            continue;
        }
        else{
            // connect
            cout<<"\na new connection request!"<<endl;
            cout<<"Request client is :"  

            <<inet_ntoa(clnt_addr.sin_addr)  

            <<":"  

            <<ntohs(clnt_addr.sin_port)  

            <<endl;

            // update the list
            alladdr[id] = (char*)malloc(sizeof(char) * ADDR_LENGTH);
            strcpy(alladdr[id], inet_ntoa(clnt_addr.sin_addr));
            allport[id] = ntohs(clnt_addr.sin_port);

            isConnected[id] = 1;

            // create new thread
            if (pthread_create(&thread_id[id], NULL, &communication, (void *)&clnt_sock) == -1){
                id --;
                perror("Error: create thread failed!\n");
                break;
            }
            id ++;
        }
    }
}

```

图 10 服务器的主线程循环关键代码片段 2

- 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）

```

// operations
while(1){
    // set rfd
    FD_ZERO(&rfd);           // initialize rfd
    FD_SET(0, &rfd);
    maxfd = 0;
    FD_SET(newfd, &rfd);     // clean the original info, in order to detect the situation that clients has closed
    if(maxfd < newfd)        // set current newfd as the maxfd
        maxfd = newfd;

    // timeout
    tv.tv_sec = 6; // second
    tv.tv_usec = 0;

    // wait
    retval = select(maxfd+1, &rfd, NULL, NULL, &tv);
    if(retval == -1){
        perror("select error and client quit!\n");
        break;
    }
    else if(retval == 0){
        continue;
    }
}

```

图 11 服务器的客户端处理子线程循环关键代码片段 1

```

else{
    // server send message
    if(FD_ISSET(0, &rfd)){
        bzero(data_send, BUFFER_LENGTH); // initialize
        fgets(data_send, BUFFER_LENGTH, stdin); // input
        // solve the exit signal
        if(!strncasecmp(data_send, "quit", 4) || !strncasecmp(data_send, "exit", 4)){
            printf("Log: server require to quit!\n");
            break;
        }
        sendlen = send(newfd, data_send, strlen(data_send), 0);
        if( sendlen > 0){
            printf("Log: data send success!\n");
        }
        else{
            perror("Log: server has NOT sent your message!\n");
            break;
        }
    }
}

```

图 12 服务器的客户端处理子线程循环关键代码片段 2

```

// receive message from client
if(FD_ISSET(newfd, &rfd)){
    bzero(data_rcv, BUFFER_LENGTH);
    recrlen = recv(newfd, data_rcv, BUFFER_LENGTH, 0);
    if( recrlen > 0){
        printf("data receive: %s", data_rcv);

        // time()
        if(!strncmp(data_rcv, "time", 4)){
            totalsec = (int)time(0);
            sec = totalsec % 60;
            min = totalsec % 3600 / 60;
            hour = (totalsec % (24 * 60) / 3600 + 8) % 24;
            sprintf(localtime, "TIME%02d:%02d:%02d\n", hour, min, sec);
            // send time to client
            if( send(newfd, localtime, strlen(localtime), 0) > 0){
                printf("data send success!\n");
            }
            else{
                perror("data send: Server has NOT sent your message!\n");
                exit(errno);
            }
        }

        // name()
        else if(!strncmp(data_rcv, "name", 4)){
            // get host name
            gethostname(hostname, sizeof(hostname));
            sprintf(hostname, "NAME%s\n", hostname);
            // send time to client
            if( send(newfd, hostname, strlen(hostname), 0) > 0){
                printf("data send success!\n");
            }
            else{
                perror("data send: Server has NOT sent your message!\n");
                exit(errno);
            }
        }
    }
}

```

图 13 服务器的客户端处理子线程循环关键代码片段 3

- 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：

```
+-----+
|      Welcome! client func menu.      |
+-----+
| 1. connect                            |
| 2. close                             |
| 3. getServerTime                     |
| 4. getServerName                     |
| 5. activeList                        |
| 6. send                              |
| 7. exit                              |
+-----+
Tips: enter a number(from 1 to 7) to choose the function
1
please enter ip:
127.0.0.1
please enter port:
2492

[Data received]
The data is
:
hello from server!
```

服务端：

```
socket create success!
bind success!
listen success!
a new connection request!
the client is :127.0.0.1:50718
```

Wireshark 抓取的数据包截图：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	50718 → 2492 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2408550129 TSecr=0 WS=128
2	0.000000758	127.0.0.1	127.0.0.1	TCP	74	2492 → 50718 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=2408550129 TSecr=2408...
3	0.000018289	127.0.0.1	127.0.0.1	TCP	66	50718 → 2492 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2408550129 TSecr=2408550129
4	0.000757671	127.0.0.1	127.0.0.1	TCP	85	2492 → 50718 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=19 TSval=2408550130 TSecr=2408550129
5	0.000776758	127.0.0.1	127.0.0.1	TCP	66	50718 → 2492 [ACK] Seq=1 Ack=20 Win=65536 Len=0 TSval=2408550130 TSecr=2408550130

- 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：

```
[Data received]
The data is
Time info
:
08:06:16
```

服务端:

```
socket create success!  
bind success!  
listen success!  
a new connection request!  
the client is :127.0.0.1:50718  
Log: data send success!  
data receive: timedata 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）:

请求数据包:

No.	Time	Source	Destination	Protocol	Length
1	0.000000000	127.0.0.1	127.0.0.1	TCP	100
2	0.000013432	127.0.0.1	127.0.0.1	TCP	100
3	0.000064034	127.0.0.1	127.0.0.1	TCP	100
4	0.000069303	127.0.0.1	127.0.0.1	TCP	100
5	10.656716230	127.0.0.1	127.0.0.53	DNS	100
6	10.656841800	127.0.0.53	127.0.0.1	DNS	100
7	10.656901273	127.0.0.1	127.0.0.53	DNS	100
8	10.663669716	127.0.0.53	127.0.0.1	DNS	100

▶ Frame 1: 1090 bytes on wire (8720 bits), 1090 bytes captured (8720 bits) on
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00:
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 50764, Dst Port: 2492, Seq: 1, Ac
▼ Data (1024 bytes)
Data: 74696d6500...
[Length: 1024]

0000 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00E.
0010 04 34 56 a4 40 00 40 06 e2 1d 7f 00 00 01 7f 00 .4V@@@.....
0020 00 01 c6 4c 09 bc 77 57 33 1a f1 aa 08 69 80 18 ...L..ww3...i..
0030 02 00 02 29 00 00 01 01 08 0a 8f 9a a3 67 8f 9a ...).....g..
0040 48 b9 74 69 6d 65 00 00 00 00 00 00 00 00 00 00 Htime.....
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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

Datatype Message

数据响应包:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	1090	50788 → 2
2	0.000012073	127.0.0.1	127.0.0.1	TCP	66	2492 → 50
3	0.000068154	127.0.0.1	127.0.0.1	TCP	79	2492 → 50
4	0.000089608	127.0.0.1	127.0.0.1	TCP	66	50788 → 2
▶ Frame 3: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on interface 0						
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)						
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1						
▶ Transmission Control Protocol, Src Port: 2492, Dst Port: 50788, Seq: 1, Ack: 1025, Len						
▶ Data (13 bytes)						
Data: 5449d4530383a32353a34320a						
[Length: 13]						
0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E.			
0010	00 41 ad 34 40 00 40 06	8f 80 7f 00 00 01 7f 00	.A.4@.@.			
0020	00 01 09 bc c6 64 3d f9	70 9a ae 9c a1 40 80 18d=. p...@..			
0030	02 00 fe 35 00 00 01 01	08 0a 8f a3 e9 f4 8f a35.....			
0040	e9 f4 54 49 4d 45 30 38	3a 32 35 3a 34 32 0a	...TIME38 :25:42.			

Datatype Message

- 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。

服务端:

```
data receive: name
data send success!
```

客户端:

```
[Data received]
The data is
Name info
:
ubuntu
```

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）:

请求包:

No.	Time	Source	Destination	Protocol	Length
1	0.0000000000	127.0.0.1	127.0.0.1	TCP	109
2	0.000013513	127.0.0.1	127.0.0.1	TCP	60
3	0.000060315	127.0.0.1	127.0.0.1	TCP	70
4	0.000068817	127.0.0.1	127.0.0.1	TCP	60

Frame 1: 1090 bytes on wire (8720 bits), 1090 bytes captured (8720 bits) on
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 50888, Dst Port: 2492, Seq: 1, Ack:
Data (1024 bytes)

Data: 6e616d6500...
[Length: 1024]

```

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00      .....E.
0010  04 34 15 49 40 00 40 06 23 79 7f 00 00 01 7f 00      .4.I@.@.#y.....
0020  00 01 c6 c8 09 bc c6 85 2d 83 c1 26 31 4d 80 18      .....&1M.....
0030  02 00 02 20 00 00 01 01 08 0a 8f b5 79 9b 8f b5      ).....y.....
0040  4f ed 6e 61 6d 65 00 00 00 0a 00 00 00 00 00 00      0name.....
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
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      .....

```

Datatype **Message**

响应包:

No.	Time	Source	Destination	Protocol	Length
1	0.000000000	127.0.0.1	127.0.0.1	TCP	16
2	0.000013513	127.0.0.1	127.0.0.1	TCP	
3	0.000060315	127.0.0.1	127.0.0.1	TCP	
4	0.000068817	127.0.0.1	127.0.0.1	TCP	

▶ Frame 3: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface
 ▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00:00
 ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 ▶ Transmission Control Protocol, Src Port: 2492, Dst Port: 50888, Seq: 1, Ack: 1
 ▶ Data (11 bytes)

Data: 4e41d457562756e74750a
[Length: 11]

Offset	Hex	ASCII
0000	00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00E..
0010	00 3f 0d ed 40 00 40 06 2e ca 7f 00 00 01 7f 00	...?..@..
0020	00 01 09 bc c6 c8 c1 26 31 4d c6 85 31 83 80 18& 1M..1..
0030	02 00 00 fe 32 00 00 01 01 08 0a 8f b5 79 9b 8f b52.....y..
0040	79 9b 4e 41 d4 d5 75 62 75 6e 74 75 0a	..NAMEsub untu..

Datatype **Message**

相关的服务器的处理代码片段:

```
// name()
else if(!strcmp(data_recv, "name", 4)){
    // get host name
    char tempName[32];
    gethostname(tempName, 32);
    sprintf(hostname, "NAME%s\n", tempName);
    // send time to client
    if( send(newfd, hostname, strlen(hostname), 0) > 0){
        printf("\ndata send success!\n");
    }
    else{
        perror("data send: Server has NOT sent your message!\n");
        exit(errno);
    }
}
```

- 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。

服务端：

```
data receive: name
data send success!
```

客户端:

```
[Data received]
The data is
List info
:
id: 1, addr: 127.0.0.1, port: 50888
```

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）：

[illegible]

Datatype

Message

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	1090	509
2	0.000066021	127.0.0.1	127.0.0.1	TCP	106	249
3	0.000071264	127.0.0.1	127.0.0.1	TCP	66	509

▶ Frame 2: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface
 ▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00
 ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 ▶ Transmission Control Protocol, Src Port: 2492, Dst Port: 50904, Seq: 1, Ack: 102
 ▼ Data (40 bytes)
 Data: 4c49535469643a20312c20616464723a203132372e302e30...
 [Length: 40]

0000	00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00E.
0010	00 5c 6c da 40 00 40 06 cf bf 7f 00 00 01 7f 00	..l.@@.
0020	00 01 09 bc c6 d8 dd 78 93 20 9a a6 e1 fe 80 18x
0030	02 00 fe 50 00 00 01 01 08 0a 8f bd b2 29 8f bd	..P.....)
0040	b2 29 4c 49 53 54 59 64 3a 20 31 2c 20 61 64 64)LISTid: 1, add
0050	72 3a 20 31 32 37 2e 30 2e 30 2e 31 2c 20 70 6f	r: 127.0 .0.1, po
0060	72 74 3a 20 35 30 39 30 34 0a	rt: 5090 4.

Datatype

Message

相关的服务器的处理代码片段：

```

// list()
if(!strcmp(data_recv, "list", 4)){
    // traverse the client list
    for(i = 0; i < id; i++){
        bzero(data_send, BUFFER_LENGTH); // initialize
        // test if is still connected
        if(isConnected[i] == 1){
            sprintf(data_send, "LISTid: %d, addr: %s, port: %d\n", i+1, alladdr[i], allport[i]);
            if( send(newfd, data_send, strlen(data_send), 0) > 0){
                printf("\ndata send success!\n");
            }
            else{
                perror("data send: Server has NOT sent your message!\n");
                exit(errno);
            }
        }
    }
}
}

```

- 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：

```

6
Please input the number of the client:
1
Please input the message you want to send:
sendtomyself

[Data received]
The data is
Send info
:
id: 1, addr: 127.0.0.1, port: 52612 send you a message:
sendtomyself

```

服务器:

```

data send success!
data receive: send:1:sendtomyselfdata send success!

```

接收消息的客户端:

```

6
Please input the number of the client:
1
Please input the message you want to send:
sendtomyself

[Data received]
The data is
Send info
:
id: 1, addr: 127.0.0.1, port: 52612 send you a message:
sendtomyself

```

Wireshark 抓取的数据包截图（发送和接收分别标记）:

发送:（红色矩形标出了发送标记）

4	5.003796313	127.0.0.1	127.0.0.53	DNS
5	9.375113063	127.0.0.1	127.0.0.1	TCP
6	9.375154385	127.0.0.1	127.0.0.1	TCP
7	9.375217842	127.0.0.1	127.0.0.1	TCP
8	9.375256640	127.0.0.1	127.0.0.1	TCP
9	10.009858015	127.0.0.1	127.0.0.53	DNS
▶ Frame 5: 1090 bytes on wire (8720 bits), 1090 bytes captured (8720 bits) on interface 0 ▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 ▶ Transmission Control Protocol, Src Port: 52612, Dst Port: 2492, Seq: 1, Ack: 1 ▶ Data (1024 bytes) Data: 73656e643a313a73656e64746f6d7973656c66000000000000... [Length: 1024]				
0040	53 a8	73 65 6e 64	3a 31 3a 73 65 6e 64 74 6f 6d	send:1 :sendtom
0050	79 73	65 6e 66 00	00 00 00 00 00 00 00 00	yself
0060	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

发送标记

接收: (红色矩形标出了接收标记)

5	9.375113063	127.0.0.1	127.0.0.1	TCP	1090 52612
6	9.375154385	127.0.0.1	127.0.0.1	TCP	66 2492
7	9.375217842	127.0.0.1	127.0.0.1	TCP	139 2492
8	9.375256640	127.0.0.1	127.0.0.1	TCP	66 52612
9	10.009858015	127.0.0.1	127.0.0.53	DNS	107 Stand

▶ Frame 7: 139 bytes on wire (1112 bits), 139 bytes captured (1112 bits) on interface
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 2492, Dst Port: 52612, Seq: 1, Ack: 1025,
▶ Data (73 bytes)
Data: 53454e4469643a20312c20616464723a203132372e302e30...
[Length: 73]

0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E..
0010	00 7d 4e c3 40 00 40 06	ed b5 7f 00 00 01 7f 00	.}N.@.@...
0020	00 01 09 bc cd 84 ce cd	fd 1c b7 72 7d f7 80 18r}...
0030	02 00 fe 71 00 00 01 01	08 0a 8f d3 ed 34 8f d34..
0040	ed 34 53 45 4e 44 69 64	3a 20 31 2c 20 61 64 64	SENDid 1, add
0050	72 3a 20 31 32 37 2e 30	2e 30 2e 31 2c 20 70 6f	r 127.0.0.1, po
0060	72 74 3a 20 35 32 36 31	32 20 73 65 6e 64 20 79	rt: 5261 2 send y
0070	6f 75 20 61 20 6d 65 73	73 61 67 65 3a 0a 73 65	ou a mes sage:.se
0080	6e 64 74 6f 6d 79 73 65	6c 66 0a	ndtomysel f.

接受标记

相关的服务器的处理代码片段:

```
if(!strcmp(data_recv, "send", 4)){
    // get id of the receiver client
    j = 0;
    bzero(data_temp, BUFFER_LENGTH);
    for(i = 5; i < BUFFER_LENGTH; i++){
        if(data_recv[i] != ':'){
            data_temp[j] = data_recv[i];
            j++;
        }
        else{
            break;
        }
    }
    toID = atoi(data_temp);
    // get message client want to send
    j = 0;
    bzero(toMessage, BUFFER_LENGTH);
    for(i++; i < BUFFER_LENGTH; i++){
        if(data_recv[i] != '\n'){
            toMessage[j] = data_recv[i];
            j++;
        }
        else{
            break;
        }
    }
    // information to send: client's information + message
    bzero(data_send, BUFFER_LENGTH); // initialize
    sprintf(data_send, "SENDid: %d, addr: %s, port: %d send you a message:\n%s\n", curID+1, alladdr[curID], allport[curID], toMessage);
    if(send(allfd[toID], data_send, strlen(data_send), 0) > 0){
        printf("data send success!\n");
    }
    else{
        perror("data send: Server has NOT sent your message!\n");
        exit(errno);
    }
}
```

相关的客户端（发送和接收消息）处理代码片段：

```
int ret = recv(sock, &server_response, sizeof(server_response), 0);
if (ret > 0){
    // judge the response type
    type = -1;
    string mark="";
    if (!strncmp(server_response, "TIME", 4)){
        // TIME type
        type = 0;
        mark = "Time info\n";
    }else if(!strncmp(server_response, "NAME", 4)){
        type = 1;
        mark = "Name info\n";
    }
    else if(!strncmp(server_response, "LIST", 4)){
        type = 2;
        mark = "List info\n";
    }
    else if (!strncmp(server_response, "SEND", 4)){
        type = 3;
        mark = "Send info\n";
    }

    cout<<"\n"
         <<"[Data received] \n"
         <<"The data is\n" << mark <<": \n"
         <<(type==-1?server_response:(server_response+4))
         <<endl;
    memset(server_response, 0, sizeof(server_response));
}
else if (ret < 0){
    // exception
    pthread_exit(NULL);
}
else if (ret == 0){
    cout<<"Error: Socket closed, exiting current receive thread"<<endl;
    pthread_exit(NULL);
}
```

- 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。

答：客户端 TCP 仍然保持连接，并没有发出 TCP 连接释放消息。


服务器端 TCP 连接在长时间后自动断开。

- 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？

答：发现连接不再存在，再次发送消息会出错。

- 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。

客户端



```
./client 112x43
count = 95
[Data received]
The data is
Time info
:
08:13:08

count = 96
[Data received]
The data is
Time info
:
08:13:08

count = 97
[Data received]
The data is
Time info
:
08:13:08

count = 98
[Data received]
The data is
Time info
:
08:13:08

count = 99
[Data received]
The data is
Time info
:
08:13:08

count = 100
[Data received]
The data is
Time info
:
08:13:08
```

Wireshark

No.	Time	Source	Destination	Protocol	Length	Info
1	0.00000000	127.0.0.1	127.0.0.1	TCP	1090	54938
2	0.000024404	127.0.0.1	127.0.0.1	TCP	66	2492
3	0.000129516	127.0.0.1	127.0.0.1	TCP	79	2492
4	0.000150430	127.0.0.1	127.0.0.1	TCP	66	54938

[illegible]

```
0000 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00  . . . . . E
0010 04 34 d2 dd 40 00 40 06 65 e4 7f 00 00 01 7f 00  .4.@.@.e.
0020 00 01 d6 9a 09 bc 30 7b dc d9 27 06 d1 4b 80 18  . . . 0{ ' . k
0030 02 00 02 29 00 00 01 01 08 0a 5d b7 58 bd 5d b7  . . . ) . . ] . x .
0040 3a e5 74 69 6d 65 00 00 00 00 00 00 00 00 00  .:time.
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . .
```

1597	84.946324576	127.0.0.1	127.0.0.1	TCP	66 5494
1598	84.946361093	127.0.0.1	127.0.0.1	TCP	79 2492
1599	84.946368431	127.0.0.1	127.0.0.1	TCP	66 5494

Data: 54494d4530e383a31373a33320a	
0000	00 00 00 00 00 00 00 00 00 00 00 08 00 45 00 E
0010	00 41 56 80 40 00 40 06 e6 34 7f 00 00 01 7f 00 . AV @ @ . 4
0020	00 01 09 bc d6 a2 a4 a7 36 31 3c 0f 8b 43 80 18 61 < . C
0030	02 00 fe 35 00 00 01 01 08 0a 5d b8 a4 90 5d b8 5
0040	a4 90 54 49 4d 45 30 38 3a 31 37 3a 33 32 0a TIME08 : 17:32 .

客户端正常收到 100 次响应。

Wireshark 实际抓包数大于 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

服务器：（交错执行）

```
data receive: timeTime data send to 5 success!  
data receive: timeTime data send to 4 success!  
data receive: timeTime data send to 5 success!  
data receive: timeTime data send to 4 success!  
data receive: timeTime data send to 5 success!  
data receive: timeTime data send to 4 success!  
data receive: timeTime data send to 5 success!  
data receive: timeTime data send to 4 success!  
data receive: timeTime data send to 5 success!  
data receive: timeTime data send to 4 success!
```

客户端（正常收到 100 次响应）

```
./client 30x31
08:41:21
count = 97
[Data received]
The data is
Time info
:
08:41:22
count = 98
[Data received]
The data is
Time info
:
08:41:23
count = 99
[Data received]
The data is
Time info
:
08:41:24
count = 100
[Data received]
The data is
Time info
:
08:41:25

08:41:22
count = 97
[Data received]
The data is
Time info
:
08:41:23
count = 98
[Data received]
The data is
Time info
:
08:41:24
count = 99
[Data received]
The data is
Time info
:
08:41:25
count = 100
[Data received]
The data is
Time info
:
08:41:26
```

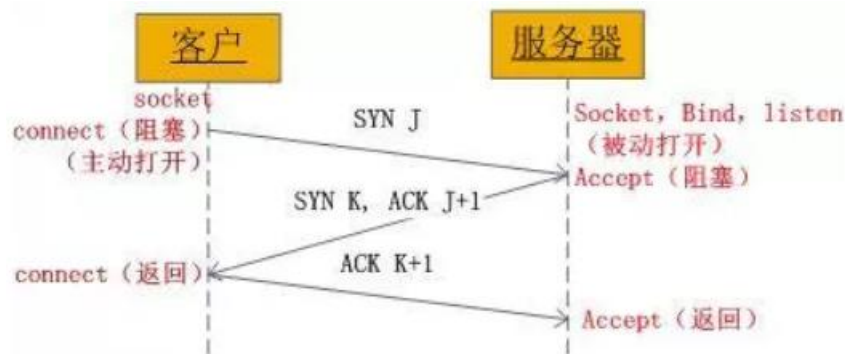
六、实验结果与分析

- 客户端是否需要调用 bind 操作？它的源端口是如何产生的？每一次调用 connect 时客户端的端口是否都保持不变？

答：客户端不需要调用 bind 操作，其端口是在连接服务器端时由操作系统随机分配，操作系统在分配的时候保证不会与现有的端口发生冲突。在每一次调用 connect 的时候客户端的源端口均会发生变化。

- 假设在服务端调用 listen 和调用 accept 之间设了一个调试断点，暂停在此断点时，此时客户端调用 connect 后是否马上能连接成功？

答：从运行结果中可以看出 connect 函数返回了 0（表示连接成功），但是由于服务端处于暂停状态，执行其他命令的时候是获取不到正确的回应的。在查询资料后了解到 TCP 的三次握手中各函数的对应关系如下所示：



我们可以看到，Listen 函数不是一个阻塞函数，其作用是将对应的 socket 变为背诵的连接监听 socket。所以在 listen 后，客户端可以通过 connect 函数与 socket 进行通信，并建立连接（三次握手后会被放入 ESTABLISHED 队列中，内核中队列的长度在调用 listen 函数的时候由 backlog 参数设置）。而 accept 作为一个阻塞函数的功能为从 established 队列中取出已经建立的连接，若没有连接则会阻塞。所以只执行 listen 但是不调用 accept 在 established 队列满员之前是不会影响客户端调用 connect 函数与服务端进行连接。

- 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？

答：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 的次数并不完全一致，连续快速发送包会导致多个包合并发送，实际数量少于 send 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？

答：服务器在同一个端口接收多个客户端的数据时区分不同客户端依靠的是客户端的 ip 或者 fd。

- 客户端主动断开连接后，当时的 TCP 连接状态是什么？这个状态保持了多久？（可以使用 netstat -an 查看）

答：保持 CLOSE_WAIT / FIN_WAIT_2 状态约一分钟。

- 客户端断网后异常退出，服务器的 TCP 连接状态有什么变化吗？服务器该如何检测连接是否继续有效？

答：无变化。服务器定时对客户端发送一个包检测连接状态，如果无回应则判断为服务端异常断开，中止连接。

七、 讨论、心得

本次实验进行的 socket 编程使用底层 API，总体实现还是具有一定的难度的，在本次实验中，我收获了基本 socket 编程以及服务器并发处理机制的经验，加深了对于多线程处理的理解。我们认为完成该实验需要预先熟悉实验一中 wireshark 的使用方法。实验的课后问题很有意义，我们尝试思考了发生如连接断开等异常之后服务器与客户端的后续工作过程，比较正常断开的区别，受益匪浅。