

# 计算机网络实验报告

本文是计算机网络——socket 通信程序设计的实验报告。

## 1 实验目的

- 1、掌握 TCP 和 UDP 协议主要特点和工作原理
- 2、理解 socket 的基本概念和工作原理
- 3、编程实现 socket 网络通信（C++ & Python）

## 2 实验环境

CentOS 6.2 + GCC 4.4.6

59.77.8.124 服务器提供的 Unix 平台

Win10+Python3.7

## 3 实验内容

### 3.1 实现迭代式 echo 服务

#### 3.1.1 实验要求

1.1 客户端：读懂客户机范例代码 `client_example.c`

为所有 socket 函数调用添加错误处理代码；

范例中服务器地址和端口是固定值，请你将它们改成允许用户以参数形式从命令行输入；

范例中客户机发送的是固定文本”Hello Network!”，请改成允许用户输入字符串，按回车发送，获取服务器响应并显示；

继续修改，实现 的循环：“接受用户输入—> 按回车发送—> 获取服务器响应并显示”，输入“bye”或按键“ESC”退出

## 1.2 服务器端：根据服务器范例代码 `server_example.c` 1.1

为所有 `socket` 函数调用添加错误处理代码；

范例中服务器端口是固定值，请你将它们改成允许用户以参数形式从命令行输入；

范例中服务器只处理一条文本，请改成循环处理客户机发来的字符串逆序回送，收到“bye”或按键“ESC”退出；

继续修改，使服务器能够迭代处理客户机：一个客户机响应结束后，继续接受下一个客户机，如此无限循环；（提示，按 `Ctrl-C` 可以终止服务器程序）

### 3.1.2 基于 TCP 协议的客户机程序实现

本程序在命令行的命令格式为

```
./echo_client -a [address] -p [port]
```

或

```
./echo_client -p [port] -a [address]
```

其中 `./echo_client` 为可执行程序名，`address` 为要连接的服务器的 IP 地址，`port` 为该服务器的端口号。这里 IP 地址和端口号的输入顺序可以调换。

具体程序如下所示。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <unistd.h>
9
10 void Inverse(char str[], char ptr[])
11 {
12     int n = strlen(str);
13     int i;
14     for(i = 0; i <= n; ++i)
15     {
16         ptr[i] = str[n - i - 1];
17     }
18     ptr[i - 1] = '\0';
19 }
20
21 int main(int argc, char *argv[])
```

```

22 {
23     int client_sock;
24     int port;
25     char *ipAddress = "127.0.0.1"; // 默认的服务器ip;
26     struct sockaddr_in server_addr;
27     char send_msg[255] = "";
28     char recv_msg[255], recv_msg1[255];
29     short server_port = 12345; // 默认的服务器端口
30     // 如果用户通过命令行参数传入服务器ip和端口，则使用用户指定的服务器ip和端口
31     if (argc == 5)
32     {
33         if (strcmp(argv[1], "-a") == 0)
34         {
35             ipAddress = (char *)malloc(strlen(argv[2])*sizeof(char));
36             strcpy(ipAddress, argv[2]);
37         }
38         else if (strcmp(argv[3], "-a") == 0)
39         {
40             ipAddress = (char *)malloc(strlen(argv[4])*sizeof(char));
41             strcpy(ipAddress, argv[4]);
42         }
43         else
44         {
45             printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p  

46                 [port] -a [addr]\n");
47             exit(1);
48         }
49         if (strcmp(argv[3], "-p") == 0)
50         {
51             port = atoi(argv[4]);
52         }
53         else if (strcmp(argv[1], "-p") == 0)
54         {
55             port = atoi(argv[2]);
56         }
57         else
58         {
59             printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p  

60                 [port] -a [addr]\n");
61             exit(1);
62         }

```

```

62
63     }
64     else
65     {
66         printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p [
        port] -a [addr]\n");
67         exit(1);
68     }
69
70     /* 创建socket */
71     if ((client_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
72     {
73         printf("Fail to create socket.\n");
74         exit(1);
75     }
76
77     /* 指定服务器地址 */
78     server_addr.sin_family = AF_INET;
79     server_addr.sin_port = htons(port);
80     if (inet_aton(ipAddress, &server_addr.sin_addr) == 0)
81     {
82         printf("Fail to transform IP address to string.\n");
83         exit(1);
84     }
85     memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
86
87     /* 连接服务器 */
88     if (connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr
        )) < 0)
89     {
90         printf("Fail to connect.\n");
91         exit(1);
92     }
93
94     printf("%% ");
95     while (fgets(send_msg, 255, stdin) != NULL)
96     {
97         /* 发送消息 */
98         printf("Send:\n%s\r", send_msg);
99         if (send(client_sock, send_msg, sizeof(send_msg), 0) < 0)
100         {
101             printf("Fail to send.\n\n");

```

```

102         continue;
103     }
104
105     /* 接收并显示消息 */
106     if (recv(client_sock, recv_msg, sizeof(recv_msg), 0) < 0)
107     {
108         printf("Fail to recieve.\n\n");
109         continue;
110     }
111     Inverse(recv_msg, recv_msg1);
112     printf("Recv: %s\r", recv_msg1);
113
114     printf("\n");
115     if (strcmp(send_msg, "bye\n") == 0)
116         break;
117     printf("%%\n");
118 }
119
120 /* 关闭socket */
121 if (close(client_sock) < 0)
122 {
123     printf("Fail to close\n");
124     exit(1);
125 }
126
127 return 0;
128 }

```

### 3.1.3 基于 TCP 协议的服务器程序实现

本程序在命令行中的命令格式为

./echo\_server [port]

其中port 为服务器打开的端口号。

具体的程序如下所示。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>

```

```

7  #include <arpa/inet.h>
8  #include <unistd.h>
9
10 void Inverse(char str[], char ptr[])
11 {
12     int n = strlen(str);
13     int i;
14     for(i = 0; i <= n; ++i)
15     {
16         ptr[i]=str[n-i-1];
17     }
18     ptr[i-1] = '\0';
19 }
20
21 int main(int argc, char *argv[])
22 {
23     int server_sock_listen, server_sock_data;
24     struct sockaddr_in server_addr, client_addr;
25     char recv_msg[255], recv_msg1[255];
26     socklen_t addr_len;
27
28     if (argc != 2)
29     {
30         printf("Usage: ./echo_client [port]\n");
31         exit(1);
32     }
33
34     /* 创建socket */
35     if ((server_sock_listen = socket(AF_INET, SOCK_STREAM, 0)) < 0)
36     {
37         printf("Fail to create socket.\n");
38         exit(1);
39     }
40
41     /* 指定服务器地址 */
42     server_addr.sin_family = AF_INET;
43     server_addr.sin_port = htons(atoi(argv[1]));
44     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
45     memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
46
47     /* 绑定socket与地址 */
48     if (bind(server_sock_listen, (struct sockaddr *)&server_addr, sizeof(

```

```

server_addr)) < 0)
49     {
50         printf("Fail to bind.\n");
51         exit(1);
52     }
53
54     /* 监听socket */
55     if (listen(server_sock_listen, 0) < 0)
56     {
57         printf("Fail to listen.\n");
58         exit(1);
59     }
60
61     while (1) {
62         if ((server_sock_data = accept(server_sock_listen, (struct sockaddr *)&
        client_addr, &addr_len)) < 0)
63         {
64             printf("Fail to accept.\n");
65             exit(1);
66         }
67         printf("Connecting to %s:%d\n\n", inet_ntoa(client_addr.sin_addr), ntohs
        (client_addr.sin_port));
68
69         while (1)
70         {
71             /* 接收并显示消息 */
72             memset(recv_msg, 0, sizeof(recv_msg));
73             if (recv(server_sock_data, recv_msg, sizeof(recv_msg), 0) < 0)
74             {
75                 printf("Fail to receive.\n\n");
76                 continue;
77             }
78             Inverse(recv_msg, recv_msg1);
79             printf("Recv: %s\n", recv_msg1);
80
81             /* 发送消息 */
82             printf("Send:\n%s", recv_msg);
83             if (send(server_sock_data, recv_msg, sizeof(recv_msg), 0) < 0)
84             {
85                 printf("Fail to send.\n\n");
86                 continue;
87             }

```

```

88
89         printf("\n");
90         if (strcmp(recv_msg, "bye\n") == 0)
91         {
92             printf("Disconnect to %s:%d\n\n", inet_ntoa(client_addr.sin_addr
93                 ), ntohs(client_addr.sin_port));
94             break;
95         }
96     }
97     /* 关闭数据socket */
98     if (close(server_sock_data) < 0)
99     {
100         printf("Fail to close data socket.\n");
101         exit(1);
102     }
103 }
104
105 /* 关闭连接socket */
106 if (close(server_sock_listen) < 0)
107 {
108     printf("Fail to close listening socket.\n");
109     exit(1);
110 }
111
112 return 0;
113 }

```

### 3.1.4 运行结果

在 CentOS 虚拟机上运行服务器和客户端程序，其运行结果如图所示。



```
root@localhost:~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost 桌面]# ./echo_client1 -a 127.0.0.1 -p 2019
Fail to connect.
[root@localhost 桌面]# ./echo_client1 -a 127.0.0.1 -p 2019
% hello
Send:
hello
Recv:
olleh
% 123
Send:
123
Recv:
321
% aaabbb
Send:
aaabbb
Recv:
bbbaaa
% bye
Send:
bye
Recv:
eyb
[root@localhost 桌面]#
```

(a) 客户端

```
root@localhost:~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost 桌面]# ./echo_server1 2019
Fail to bind.
[root@localhost 桌面]# ./echo_server1 2019
Connecting to 127.0.0.1:46962

Recv:
olleh
Send:
hello

Recv:
321
Send:
123

Recv:
bbbaaa
Send:
aaabbb

Recv:
eyb
Send:
bye

Disconnect to 127.0.0.1:46962
```

(b) 服务器

图 1 迭代式 echo 服务的运行结果

### 3.1.5 服务器监听函数完成队列最大长度测试

先将服务器的listen() 中的backlog 参数设置为 1，如下图所示。

```
cs174299@mcore:~/network
45     memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
46
47     /* 绑定socket与地址 */
48     if (bind(server_sock_listen, (struct sockaddr *)&server_addr, sizeof
(server_addr)) < 0)
49     {
50         printf("Fail to bind.\n");
51         exit(1);
52     }
53
54     /* 监听socket */
55     if (listen(server_sock_listen, 1) < 0)
56     {
57         printf("Fail to listen.\n");
58         exit(1);
59     }
60
61     while (1) {
62         if ((server_sock_data = accept(server_sock_listen, (struct s
ockaddr *)&client_addr, &addr_len)) < 0)
63         {
64             printf("Fail to accept.\n");
65             exit(1);
66         }
67     }
68 }
```

图 2 修改服务器完成队列的最大长度为 1

接着，让 2 个客户端与服务器连接，并使用 netstat 查看服务器 socket 状态。但是显示 2 个连接全部建立成功。我又尝试连接第 3 个客户端，同样还是建立成功。结果如图 3 所示。

```
[cs174299@mcore network]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mcore network]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mcore network]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mcore network]$
```

图 3 使用 netstat 查看 socket 状态

事实上，backlog 表示的是监听队列的最大长度，而不是服务器可以同时链接的客服端的长度。当客户端发起connect 时候，这时占用的就是服务器的监听队列，所以backlog 限制的是客户端 connect 的数量，当服务器调用accept 时就会从监听队列里面提取出一个连接，这是监听队列里面的连接数量就会减少，所以只有服务器accept 处理请求较慢的时候，此时客户端 connect 较多，超过backlog 的数量之后，才会出现新来的请求被拒绝的现象。<sup>[1]</sup> 于是当backlog 设置很小，这时我们接进多少台机器都没问题是因为服务器机器处理速度很快队列来不及填满就处理完了，而且在同一个时刻到来的连接还是很少的，因为我开启客户端的速度不可能赶上机器处理的速度，即并没有真正实现同时连接服务器。<sup>[2]</sup> 因此，这里 socket 的状态不会出现原本我们想象的 SYN\_RCVD，客户端也不会得到一个 ECONNREFUSED 错误。

事实上，内核为任何一个给定的监听套接口维护两个队列。（1）未完成连接队列：每个这样的 SYN 分节对应其中一项，已由某个客户发出并到达服务器，而服务器正在等待完成相应的 TCP 三路握手过程。这些套接口处于SYN\_RCVD 状态；（2）已完成连接队列，每个已完成 TCP 三路握手过程的客户对应其中一项。这些套接口处于ESTABLISHED 状态。

<sup>1</sup> <http://blog.chinaunix.net/uid-24880153-id-3757266.html>

<sup>2</sup> <http://blog.chinaunix.net/uid-17102734-id-2830185.html>

```

[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:8888     127.0.0.1:12345    ESTABLISHED
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     ESTABLISHED
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        1      0 127.0.0.1:8888     127.0.0.1:12345    CLOSE_WAIT
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     FIN_WAIT2
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        1      0 127.0.0.1:8888     127.0.0.1:12345    CLOSE_WAIT
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:8888     127.0.0.1:12345    ESTABLISHED
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     ESTABLISHED

```

图 5 服务器先连接再挂断再连接的 socket 状态

当来自客户的 SYN 到达时，TCP 在未完成连接队列中创建一个新项，然后响应以三路握手的第二个分节——服务器的 SYN 响应，其中稍带对客户 SYN 的 ACK（即 SYN + ACK）。这一项一直保留在未完成连接队列中，直到三路握手的第三个分节（客户对服务器 SYN 的 ACK）到达或者该项超时为止（曾经源自 Berkeley 的实现为这些未完成连接的项设置的超时值为 75 秒）。如果三路握手正常完成，该项就从未完成连接队列移到已完成连接队列的队尾。当进程调用 accept 时，已完成连接队列中的队头项将返回给进程，或者如果该队列为空，那么进程将被投入睡眠，直到 TCP 在该队列中放入一项才唤醒它。<sup>[3]</sup>

### 3.1.6 客户机 bind 测试

首先修改程序，让客户端强行绑定本机 8888 端口。

下面，运行客户机连接服务器，由客户机主动 close，然后再次运行客户机。利用 netstat 进行分析，其显示结果如图所示。

```

[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:8888     127.0.0.1:12345    ESTABLISHED
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     ESTABLISHED
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     TIME_WAIT
[cs174299@mc00re ~]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:8888     127.0.0.1:12345    ESTABLISHED
tcp        0      0 127.0.0.1:12345    127.0.0.1:8888     ESTABLISHED
[cs174299@mc00re ~]$

```

图 4 客户机先连接再挂断再连接的 socket 状态

之后，运行客户机连接服务器，由服务器主动 close，再次运行服务器，由客户机发送信息。利用 netstat 进行分析，其显示结果如图 5 所示。

从上述两个过程可以看出，如果服务器正常运行，而客户端的挂断再连接不会影响客户端的使用；但是如果服务器出现挂断再连接，这时客户端会出现 CLOSE\_WAIT，服务器端

<sup>3</sup> <http://blog.csdn.net/shandongmachao/article/details/48372485>

出现 FIN\_WAIT\_2。<sup>[4]</sup> 这是因为服务器端强制断开 Socket 时向客户端发送了 FIN 请求，客户端已经没有能力继续回复 ACK，造成了服务器端大量的端口处在 FIN\_WAIT\_2 状态，不能释放。而客户端程序仍在执行，服务器端关闭了 socket 链接，但是我方忙于读或者写，没有关闭连接，造成 CLOSE\_WAIT。<sup>[5]</sup> 这时不关闭客户端程序，重启服务器，客户端再次发送信息，却在经历三次握手后退出，socket 也随之消失。

## 3.2 实现并发式 echo 服务

### 3.2.1 实验要求

在任务 1 的基础上修改服务器代码，以创建子进程的并发方式提供服务

实验报告要求：

给出客户机和服务器的源代码，并附上客户机和服务器运行时的截图，要求至少 2 个并发客户机

服务器 accept 之后会返回一个用于传输数据的 socket，调用 fork() 会使父子进程同时拥有此 socket 描述符，父进程分支中是否需要关闭该 socket？请分析原因（分别运行关和不关的代码，netstat 观察运行状态）

### 3.2.2 基于 TCP 协议的客户机程序实现

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <unistd.h>
9
10 void Inverse(char str[], char ptr[])
11 {
12     int n = strlen(str);
13     int i;
14     for(i = 0; i <= n; ++i)
15     {
16         ptr[i] = str[n - i - 1];
17     }
18     ptr[i - 1] = '\0';
```

---

<sup>4</sup> <http://blog.csdn.net/realmeh/article/details/17597527>

<sup>5</sup> <http://blog.csdn.net/xiaofei0859/article/details/6050806>

```

19 }
20
21 int main(int argc , char *argv[])
22 {
23     int client_sock;
24     int port;
25     char *ipAddress;
26     struct sockaddr_in server_addr;
27     char send_msg[255];
28     char recv_msg[255] , recv_msg1[255];
29
30     if (argc == 5)
31     {
32         if (strcmp(argv[1] , "-a") == 0)
33         {
34             ipAddress = (char *)malloc(strlen(argv[2])*sizeof(char));
35             strcpy(ipAddress , argv[2]);
36         }
37         else if (strcmp(argv[3] , "-a") == 0)
38         {
39             ipAddress = (char *)malloc(strlen(argv[4])*sizeof(char));
40             strcpy(ipAddress , argv[4]);
41         }
42         else
43         {
44             printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p
45                 [port] -a [addr]\n");
46             exit(1);
47         }
48         if (strcmp(argv[3] , "-p") == 0)
49         {
50             port = atoi(argv[4]);
51         }
52         else if (strcmp(argv[1] , "-p") == 0)
53         {
54             port = atoi(argv[2]);
55         }
56         else
57         {
58             printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p
59                 [port] -a [addr]\n");

```

```

59         exit(1);
60     }
61
62 }
63 else
64 {
65     printf("Usage: ./echo_client -a [addr] -p [port] OR ./echo_client -p [
        port] -a [addr]\n");
66     exit(1);
67 }
68
69 /* 创建socket */
70 if ((client_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
71 {
72     printf("Fail to create socket.\n");
73     exit(1);
74 }
75
76 /* 指定服务器地址 */
77 server_addr.sin_family = AF_INET;
78 server_addr.sin_port = htons(port);
79 if (inet_aton(ipAddress, &server_addr.sin_addr) == 0)
80 {
81     printf("Fail to transform IP address to string.\n");
82     exit(1);
83 }
84 memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
85
86 /* 连接服务器 */
87 if (connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr
    )) < 0)
88 {
89     printf("Fail to connect.\n");
90     exit(1);
91 }
92
93 printf("%%\n");
94 while (fgets(send_msg, 255, stdin) != NULL)
95 {
96     /* 发送消息 */
97     printf("Send:\n%s\r", send_msg);
98     if (send(client_sock, send_msg, sizeof(send_msg), 0) < 0)

```

```

99      {
100          printf("Fail to send.\n\n");
101          continue;
102      }
103
104      /* 接收并显示消息 */
105      if (recv(client_sock, recv_msg, sizeof(recv_msg), 0) < 0)
106      {
107          printf("Fail to recieve.\n\n");
108          continue;
109      }
110      Inverse(recv_msg, recv_msg1);
111      printf("Recv: %s\r", recv_msg1);
112
113      printf("\n");
114      if (strcmp(send_msg, "bye\n") == 0)
115          break;
116      printf("%%\n");
117  }
118
119      /* 关闭 socket */
120      if (close(client_sock) < 0)
121      {
122          printf("Fail to close\n");
123          exit(1);
124      }
125
126      return 0;
127  }

```

### 3.2.3 基于 TCP 协议的服务器程序实现

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <sys/wait.h>
7  #include <netinet/in.h>
8  #include <arpa/inet.h>
9  #include <unistd.h>
10

```

```

11 void Inverse(char str[], char ptr[])
12 {
13     int n = strlen(str);
14     int i;
15     for(i = 0; i <= n; ++i)
16     {
17         ptr[i]=str[n-i-1];
18     }
19     ptr[i-1] = '\0';
20 }
21
22 int main(int argc, char *argv[])
23 {
24     int server_sock_listen, server_sock_data;
25     struct sockaddr_in server_addr, client_addr;
26     char recv_msg[255], recv_msg1[255];
27     socklen_t addr_len;
28     pid_t pid;
29     int pid_status;
30
31     if (argc != 2)
32     {
33         printf("Usage: ./echo_client [port]\n");
34         exit(1);
35     }
36     /* 创建socket */
37     if ((server_sock_listen = socket(AF_INET, SOCK_STREAM, 0)) < 0)
38     {
39         printf("Fail to create socket.\n");
40         exit(1);
41     }
42     /* 指定服务器地址 */
43     server_addr.sin_family = AF_INET;
44     server_addr.sin_port = htons(atoi(argv[1]));
45     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
46     memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
47
48     /* 绑定socket与地址 */
49     if (bind(server_sock_listen, (struct sockaddr *)&server_addr, sizeof(
        server_addr)) < 0)
50     {
51         printf("Fail to bind.\n");

```



```

52     exit(1);
53 }
54
55 /* 监听socket */
56 if (listen(server_sock_listen, 20) < 0)
57 {
58     printf("Fail to listen.\n");
59     exit(1);
60 }
61
62 signal(SIGCHLD, SIG_IGN);
63 while (1)
64 {
65     if ((server_sock_data = accept(server_sock_listen, (struct sockaddr *)&
66         client_addr, &addr_len)) < 0)
67     {
68         printf("Fail to accept.\n");
69         exit(1);
70     }
71     printf("Connecting to %s:%d\n\n", inet_ntoa(client_addr.sin_addr), ntohs
72         (client_addr.sin_port));
73
74     if ((pid = fork()) < 0)
75     {
76         printf("Fork error.\n");
77         exit(1);
78     }
79     else if (pid == 0) //子进程
80     {
81         if (close(server_sock_listen) < 0) //关闭监听套接字
82         {
83             printf("Fail to close listening socket.\n");
84             exit(1);
85         }
86         while (1) //处理该客户端的请求
87         {
88             /* 接收并显示消息 */
89             memset(recv_msg, 0, sizeof(recv_msg));
90             if (recv(server_sock_data, recv_msg, sizeof(recv_msg), 0) < 0)
91             {
92                 printf("Fail to receive.\n\n");

```

```

92         continue;
93     }
94     printf("From %s:%d: \n", inet_ntoa(client_addr.sin_addr), ntohs(
        client_addr.sin_port));
95     Inverse(recv_msg, recv_msg1);
96     printf("Recv: %s\n", recv_msg1);
97
98     /* 发送消息 */
99     printf("Send:\n%s", recv_msg);
100    if (send(server_sock_data, recv_msg, sizeof(recv_msg), 0) < 0)
101    {
102        printf("Fail to send.\n\n");
103        continue;
104    }
105
106    printf("\n");
107    if (strcmp(recv_msg, "bye\n") == 0)
108    {
109        printf("Disconnect to %s:%d\n\n", inet_ntoa(client_addr.
            sin_addr), ntohs(client_addr.sin_port));
110        break;
111    }
112    }
113    exit(0);
114 }
115 if (close(server_sock_data) < 0) //父进程关闭连接套接字，继续等待其他连
    接的到来
116 {
117     printf("Fail to close data socket.\n");
118     exit(1);
119 }
120 }
121
122 /* 关闭连接socket */
123 if (close(server_sock_listen) < 0)
124 {
125     printf("Fail to close listening socket.\n");
126     exit(1);
127 }
128
129 return 0;
130 }

```

### 3.2.4 运行结果

同样在 Unix 平台上运行服务器和客户端程序，其运行结果如图所示。

```
From 80.159.4.8:33215:
Recv:
1mun ma i
Send:
i am num1

From 80.159.4.8:33215:
Recv:
eyb
Send:
bye

Disconnect to 80.159.4.8:33215
```

(a) 客户端 1

```
From 127.0.0.1:46968:
Recv:
olleh
Send:
hello

From 127.0.0.1:46968:
Recv:
eyb
Send:
bye

Disconnect to 127.0.0.1:46968
```

(b) 客户端 2

```
root@localhost:~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost 桌面]# ./echo_server2 2019
Connecting to 80.159.4.8:33215

From 80.159.4.8:33215:
Recv:
olleh
Send:
hello

From 80.159.4.8:33215:
Recv:
uoyerawoh
Send:
howareyou

Connecting to 127.0.0.1:46968

From 127.0.0.1:46968:
Recv:
cccbbbaaa
Send:
aaabbbccc

From 127.0.0.1:46968:
Recv:
2munm'i,olleh
Send:
hello,i'mnum2

From 80.159.4.8:33215:
Recv:
1mun ma i
Send:
```

(c) 服务器

图 6 并发式 echo 服务的运行结果

### 3.2.5 服务器监听函数完成队列最大长度测试

先将服务器的listen() 中的backlog 参数设置为 1，接着，让 3 个客户端与服务器连接，并使用netstat 查看服务器 socket 状态。但是显示 3 个连接全部建立成功。我又尝试连接第 4 个客户端，同样还是建立成功。结果如图 7所示。其原因和迭代式的应该是一致的。

```

[cs174299@mc core para]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mc core para]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:9999     127.0.0.1:50108    ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:50107    ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:50106    ESTABLISHED
tcp        0      0 127.0.0.1:50107    127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:50106    127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:50108    127.0.0.1:9999     ESTABLISHED
[cs174299@mc core para]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:9999     127.0.0.1:50109    ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:50108    ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:50107    ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:50106    ESTABLISHED
tcp        0      0 127.0.0.1:50107    127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:50106    127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:50109    127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:50108    127.0.0.1:9999     ESTABLISHED
[cs174299@mc core para]$

```

图 7 使用 netstat 查看 socket 状态

### 3.2.6 客户机 bind 测试

首先修改程序，让客户端强行绑定本机 8888 端口。

下面，先运行一个客户机连接服务器，然后再尝试让另一个客户机连接，结果第二个客户机无法绑定 8888 端口，如图所示。利用 netstat 进行分析，其显示结果如图 9 所示。

```

[cs174299@mc core para]$ ./echo_client -a 127.0.0.1 -p 9999
Fail to bind.
[cs174299@mc core para]$

```

图 8 客户机无法绑定 8888 端口

```

[cs174299@mc core para]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
[cs174299@mc core para]$ netstat -an | grep '127.0.0.1'
tcp        0      0 127.0.0.1:631      0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:25       0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.1:8888     127.0.0.1:9999     ESTABLISHED
tcp        0      0 127.0.0.1:9999     127.0.0.1:8888     ESTABLISHED

```

图 9 利用 netstat 分析 socket 状态

可见，客户端绑定端口后，会导致一台机器上只能运行一个客户端，还有可能与其他进程发生冲突，所以不建议客户端强制绑定端口。

## 3.3 Python 实现简单多人聊天室

### 3.3.1 实验要求

已知聊天室服务的 IP 和端口，用户可以随时加入聊天室

服务端在用户进入/聊天室时应显示系统消息

新用户需输入一个昵称，且不能与现存用户同名

用户以规定字符串（如“exit”）退出聊天室

支持异常捕捉

### 3.3.2 多人实时聊天室服务器端实现

```
1 import socket
2 import threading
3 import time
4 import logging
5
6
7 def main():
8     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9     addr = ('127.0.0.1', 9998)
10    s.bind(addr)
11    print('UDP Server on %s:%d...' % (addr[0], addr[1]))
12
13    user = {} # {addr:name}
14    while True:
15        try:
16            data, addr = s.recvfrom(1024)
17            #if addr in user:
18                #s.sendto('\n该用户已存在'.encode(), addr)
19                #continue
20            if not addr in user:
21                for address in user:
22                    s.sendto(data + ' 进入聊天室...' .encode(), address)
23                user[addr] = data.decode('utf-8')
24                continue
25            #else:
26                #s.sendto('\n该用户已存在'.encode(), addr)
27                #return
28
29            if 'EXIT' in data.decode('utf-8'):
30                name = user[addr]
31                user.pop(addr)
32                for address in user:
33                    s.sendto((name + ' 离开了聊天室...' ).encode(), address)
34            else:
35                print('"%s" from %s:%s' %
36                      (data.decode('utf-8'), addr[0], addr[1]))
37                for address in user:
38                    if address != addr:
39                        s.sendto(data, address)
40
41        except ConnectionResetError:
```

```

42         print('Someone left unexcept. ')
43
44
45 if __name__ == '__main__':
46     main()

```

### 3.3.3 多人实时聊天室客户端实现

```

1  import socket
2  import threading
3  import logging
4  import time
5
6  def recv(sock, addr):
7      '''
8      一个UDP连接在接收消息前必须要让系统知道所占端口
9      也就是需要send一次，否则win下会报错
10     “    data=sock.recv(1024)
11         OSError: [WinError 10022] 提供了一个无效的参数。    ”
12     '''
13     sock.sendto(name.encode('utf-8'), addr)
14     while True:
15         data = sock.recv(1024)
16         print(data.decode('utf-8'))
17
18
19 def send(sock, addr):
20     while True:
21         string = input()
22         message = name + ' : ' + string
23         data = message.encode('utf-8')
24         sock.sendto(data, addr)
25         if string == 'EXIT':
26             break
27
28 def main():
29     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
30     server = ('127.0.0.1', 9998)
31
32     tr = threading.Thread(target=recv, args=(s, server), daemon=True)
33     ts = threading.Thread(target=send, args=(s, server))
34     tr.start()

```

```

35     ts.start()
36     ts.join()
37     s.close()
38
39 if __name__ == '__main__':
40     print("----- 欢迎来到聊天室,退出聊天室请输入'EXIT'-----")
41     name = input('请输入你的名称:')
42     print('-----%s-----' % name)
43     main()

```

### 3.3.4 运行结果

在 Windows 的 shell 环境中运行程序，结果如下图所示。

```

Windows PowerShell
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1> python ./client.py
----- 欢迎来到聊天室,退出聊天室请输入'EXIT' -----
请输入你的名称:aaa
hello
bbb 进入聊天室...
bbb : hello,i am bbb
i am aaa
ccc 进入聊天室...
ccc : i am ccc
hahaha
bbb : bbbbbbbb
bbb : exit
bbb 离开了聊天室...

```

(a) 客户端 1

```

Windows PowerShell
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1> python ./client.py
----- 欢迎来到聊天室,退出聊天室请输入'EXIT' -----
请输入你的名称:bbb
hello,i am bbb
aaa : i am aaa
ccc 进入聊天室...
ccc : i am ccc
aaa : hahaha
bbbbbbbb
exit
EXIT
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1>

```

(b) 客户端 2

```

Windows PowerShell
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1> python ./client.py
----- 欢迎来到聊天室,退出聊天室请输入'EXIT' -----
请输入你的名称:ccc
i am ccc
aaa : hahaha
bbb : bbbbbbbb
bbb : exit
bbb 离开了聊天室...

```

(c) 客户端 3

```

Windows PowerShell
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1> python client.py
----- 欢迎来到聊天室,退出聊天室请输入'EXIT' -----
请输入你的名称: aaa
该用户已经存在!
请输入你的名称: ccc
您已经进入聊天室...
ccc

```

(d) 服务器

图 10 多人实时聊天室

```
Windows PowerShell
PS C:\Users\Apple\Desktop\大三上\计网\实验\PythonApplication1\PythonApplication1> python client.py
-----欢迎来到聊天室, 退出聊天室请输入'EXIT'-----
请输入你的名称: aaa
该用户已经存在!
请输入你的名称: ccc
您已经进入聊天室...
ccc -----
```

图 11 重名进入聊天室

## 4 小结

通过本次实验，我学会了如何在 Unix/Linux 系统上进行简单的 Socket 编程。在实验过程中遇到了一些问题：

- 1、Unix/Linux 系统中的回车和换行是分开的，也就是说每次 fgets 到的字符串最后两个字符是“\n”和“\r”，于是在进行输出时必须在最后人工添加回车符，否则就不能正常读写。
- 2、服务器与客户端三次握手后就闪退，这是因为程序中在如此过程后就关闭了套接字，于是又添加了一重死循环强制不断处理而直到发送 bye 后才关闭套接字。
- 3、由于 Unix 系统环境编程课程，对于父进程和子进程的概念还不清晰，在参考了教材后，模仿课本上的程序完成了并发服务器程序。

总之，经过本次实验，不仅加深了我对 socket 的理解，也增强了我 Unix 系统编程的能力。