Xi'an Jiaotong University

TCP/IP 网络编程

课程实验报告

计算机 03/杨南/10055076

计算机 03/徐彪根/10055075

2013/12/30

目录

一、功能演示

打开一个终端，输入以下命令进行编译：

```
root@ubuntu:/home/marinyoung/echo# make[1]
gcc –c server.c
gcc –c client.c
gcc –o server server.o
gcc –o client client.o
```

运行服务器：

```
root@ubuntu:/home/marinyoung/echo# ./server
Bind ok!
Listening…
```

新建一个终端，进入文件目录，运行客户端：

```
marinyoung@ubuntu:~/echo$ ./client 127.0.0.1
Server connected!
```

其中 127.0.0.1 是服务器地址，这里取本地的回环地址。

1、注册

正常情况下，客户端的信息如下所示：

REGIST

NAME: **marin**

PASS: **marinyoung**

Regist success!

如果用户已经注册（userData 里已经保存用户的信息）返回：

User exist!

2、登录

正常情况下，客户端的信息如下所示：

LOGIN

NAME: **marin**

PASS: **marinyoung**

Login success!

1）限定客户端登录后即不能执行登录命令，此时在键入 LOGIN 命令，服务器返回：

You're online, can't login!

2）如果输入的用户已经登录，输出：

You’re already login!

3）如果输入的用户没有注册（userData 里没有找到 marin 的信息），返回：

User not exist, Regist first!

[1] 用红色加粗字体表示用户输入的内容

4）如果密码错误，返回错误信息并提示重新输入密码：
Password error!
PASS:
5）如果密码连续错误输入三次，返回
Password error 3 times, exit!
此时需要重新执行 LOGIN 命令并输入用户名重新登录。

3、显示列表

正常情况下，返回已登录的用户（在线用户）列表
marin
admin
……
如果没有用户在线，返回： No online user!

4、发送聊天请求

正常情况：
CHAT
NAME: **marin**
Marin 为该客户端希望的聊天对象：
1）如果 marin 不在线，服务器返回：
User not found!
2）如果 marin 处于通信状态（status == TALKING || status == CALLING），返回：
Marin busy now!
3）如果用户在线，且没有和其他客户端通信，此时服务器会向 marin 客户端发送一个聊天请求：
Request to talk, accept?(y/n)
A）如果 marin 接受聊天请求（y || Y），发送聊天请求的客户端打印提示信息：
Accepted!
双方即可开始聊天。输入小写的 exit 即可结束聊天；
B）如果被拒绝（n || N），打印：
Rejected!
双方不能通信，需要重新选择聊天对象或者重新发送聊天请求。
C）其他字符，打印
Command error!

5、退出客户端

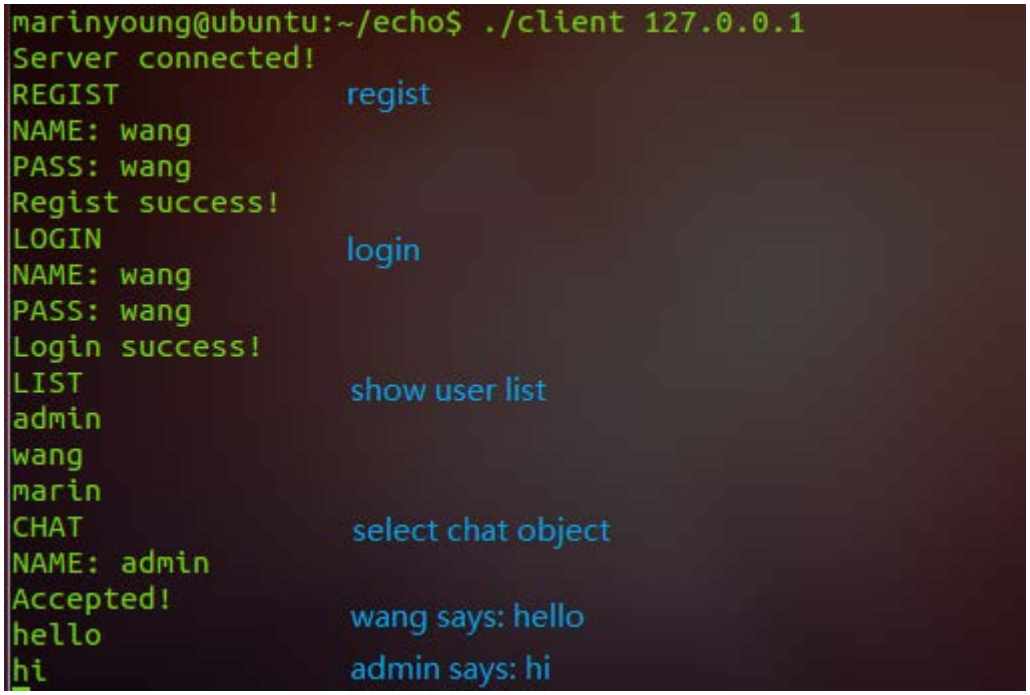
输入 EXIT 命令即可退出客户端。
EXIT
str_cli: server terminated prematurely: Success

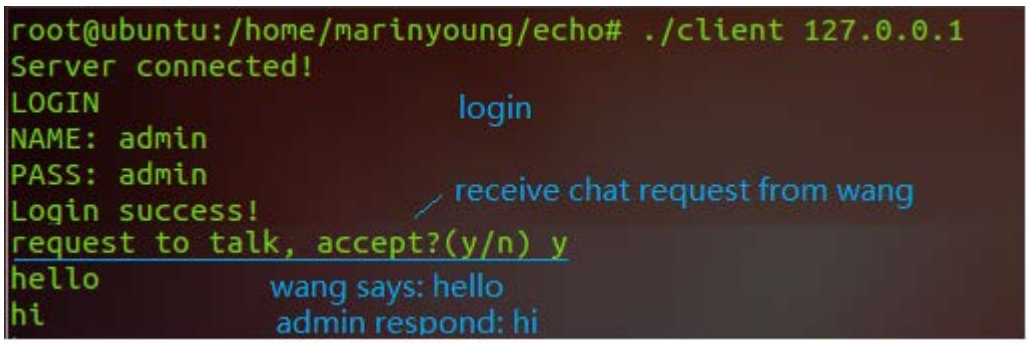
6、其他命令

输入错误的命令，服务器给出提示：
Cmd error!

7、实验截图

下面给出一个完整的操作过程的截图：





二、实现细节

1、unp.h

unp.h 里是各种头文件、宏定义以及函数声明。

```
#ifndef UNP_H
```

```
#define UNP_H

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>

#define BUFF_SIZE 50            // 服务器接收消息的缓冲区大小
#define MAX_LEN 15              // 用户名、密码的最大长度
#define SERV_PORT 8000          // 服务器的端口号
#define BACKLOG 20              // 定义服务器能建立的最大连接数
#define MAX_USER 20             // 最多同时在线用户数

#ifndef ERROR                   // 定义错误信息类型
#define ERR_USER_EXIST -1
#define ERR_ALREADY_LOGIN -1
#define ERR_PASSWORD -1
#define ERR_USER_NOT_EXIST -1
#define ERR_USER_NOT_FOUND -1
#define ERR_USER_BUSY -1
#define ERR_CMD -1
#endif

#ifndef BOOL                    // 自定义 bool 变量
#define TRUE 1
#define FALSE 0
#endif

enum USER_STATE                 // 定义用户的状态：OFFLINE、ONLINE、CALLING、TALKING
{
    OFFLINE = 0, ONLINE, CALLING, TALKING
};

struct USER                     // 结构体定义了用户名、状态、socket 连接描述符等信息
{
    char name[MAX_LEN];
    enum USER_STATE status;     /*status: 0:OFFLINE, 1:ONLINE, 2:CALLING, etc*/
```

```
    int sockfd;
};
// （省略）函数声明
// ……
#endif
```

2、sys.h

程序中有大量的系统调用，服务器端主要使用 send()和 recv()两个系统调用，客户端主要使用 read()和 write 两个系统调用。为代码书写方便，我们队这四个函数进行了重定义，（《Unix Network Programming》一书里称之为包裹函数）主要是对异常（错误）进行了中断处理，以 send()为例：

```
#ifndef SYS_H
#define SYS_H
#include "unp.h"
/*
 * NAME: Send
 * FUNCTION: 对 send 的异常情况进行处理
 * PARAMETER:    sockfd：一个用于标识已连接套接口的描述字
 *               buf：包含待发送数据的缓冲区
 *               len：缓冲区的长度
 *               flags：调用执行方式，一般为0
 * RETURN VALUE：nSend：实际发送的数据长度
*/
ssize_t Send(int sockfd, const void *buf, size_t len, int flags)
{
    int nSend;
    nSend = send(sockfd, buf, len, flags);
    if (nSend == -1)                // 发送异常
    {
        perror("send error!");      // 打印错误信息
        exit(1);                    // 异常退出
    }
    return nSend;
}

// （省略）recv()、read()、write()等系统调用的重定义
// ……
#endif
```

其余各系统调用的重定义方式与 send()相似。

3、server.c

基于 C/S 模型的客户端和服务器建立连接及通信的过程如下图所示：

1）建立连接

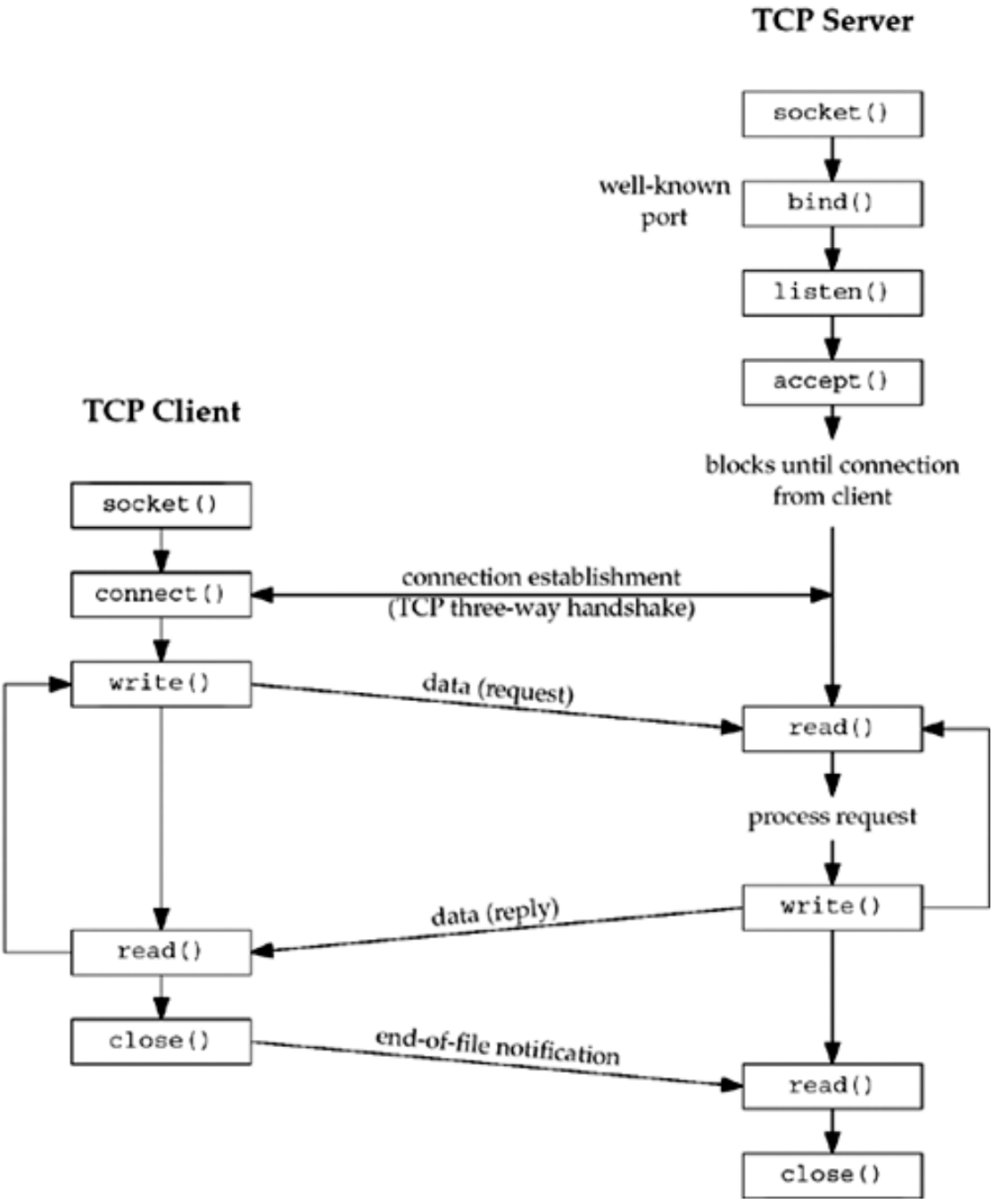


图 1 基本 TCP 客户/服务器程序的套接字函数

服务器建立一个新的 socket 之后，需要绑定到一个有名端口，然后监听该端口，一直租苏到新的客户连接到达。这一过程的代码如下：

```
// ……
/* create new socket */
listenfd = socket(AF_INET, SOCK_STREAM, 0);
// （略）异常检测和处理
/* set address */
memset(&serverAddr, 0, sizeof(serverAddr));          // 将服务器地址置 0
```

```
serverAddr.sin_family = AF_INET;                    // 协议簇为 AF_INET 即 IPv4 协议
serverAddr.sin_port = htons(SERV_PORT);             // 服务器端口
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);     // 服务器地址为本机 IP 地址
// 绑定到有名端口
errBind = bind(listenfd, (struct sockaddr*) &serverAddr, sizeof(serverAddr));
// （略）异常检测和处理
// ……
```

至此，服务器已经建立连接，并且监听端口，等待客户端连接的到来。

2）多路复用处理多客户端连接

Server.c 中，定义了一个结构体数组的全局变量

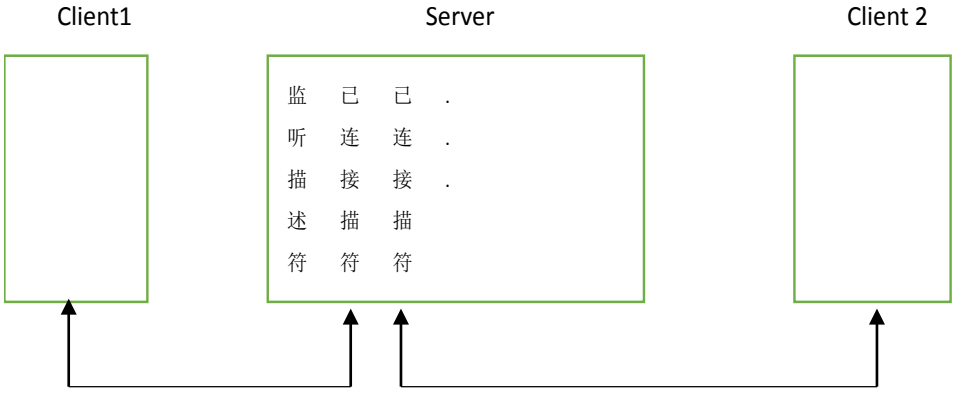
struct USER user[MAX_USER];

用于保存在线（已登录）的用户信息。同时定义一个全局变量 nUser 用于保存已登录的用户个数。如下图所示，假设有两个用户登录：

User[0]			User[1]					
Name	Status	Sockfd	Name	Status	Sockfd			

|←----------nUser----------→|

客户端与服务器建立连接后服务器的装填如下图所示。



客户端建立连接后的 TCP 服务器状态

具体的代码实现如下：

```
// ……
fd_set watchset, rset;                  //watchset 为监听的描述符集
int maxfd;                              //最大描述符序号
// ……
FD_ZERO(&watchset);                     /* clear all bits in watchset */
FD_SET(listenfd, &watchset);            /* turn on the bit for listenfd in watchset */

    for( ; ; )                          // 循环，响应多个客户端连接请求
    {
        rset = watchset;                // 保存原始的描述符集
        errSel = select(maxfd + 1, &rset, NULL, NULL, NULL); // 等待某个事件发生
        // （略）异常检测和处理
        /* if client connect request arrive (is the bit for listenfd on in rset?) */
```

```
        if (FD_ISSET(listenfd, &rset))
        {
            /* create a new socket */
            connfd = accept(listenfd, (struct sockaddr *) &clientAddr, &addrSize);
            // （略）异常检测和处理
            for (i = 0; i < MAX_USER; i ++)
            {
                if(user[i].sockfd == -1)
                {
                    user[i].sockfd = connfd;/* save descriptor */
                    break;
                }
            }
            if (i == MAX_USER)
            {
                printf("Too many users!");
                exit(1);
            }

            FD_SET(connfd, &watchset);              /* add new descriptor to set */
            maxfd = maxfd > connfd ? maxfd : connfd;    // 更新 maxfd
            maxOfI = i > maxOfI ? i : maxOfI;           /* maxOfI 为已经建立连接的客户
端的个数 */
        }

        for (i = 0; i <= maxOfI; i++)                   // 逐一扫描每一个客户端
        {
            if ( (sockfd = user[i].sockfd) == -1)
                continue;
            if (FD_ISSET(sockfd, &rset))                //sockfd 可读：收到了来自 sockfd 的
请求信息
            {
                // 处理
            }
```

	Fd0	Fd1	Fd2	Fd3	
Watchset：	listenfd	User[0].sockfd	User[1].sockfd	……	

Maxfd + 1

客户建立连接后的 TCP 服务器的数据结构

用 strace 工具追踪服务器运行可以看到建立连接过程中 watchset 的变化情况：

```
……
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(8000), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
……
```

```
listen(3, 20)                                   = 0
……
select(4, [3], NULL, NULL, NULL)                = 1 (in [3])
accept(3, {sa_family=AF_INET, sin_port=htons(51815), sin_addr=inet_addr("127.0.0.1")}, [16]) = 4
……
select(5, [3 4], NULL, NULL, NULL)              = 1 (in [4])
……
select(5, [3 4], NULL, NULL, NULL)              = 1 (in [3])
accept(3, {sa_family=AF_INET, sin_port=htons(51816), sin_addr=inet_addr("127.0.0.1")}, [16]) = 5
……
select(6, [3 4 5], NULL, NULL, NULL)            = 1 (in [5])
……
```

可以看到，每来一个新的 socket，就将其描述符 connfd 加入到 watchset 中，然后继续监听；当 watchset 中某一位可读，则执行响应的操作。

3）客户端命令处理

定义的命令主要有 REGIST、LOGIN、LIST、CHAT、EXIT 5 个，下面是其处理过程:

```
if (FD_ISSET(sockfd, &rset))
{
    memset(recvBuff, '\0', BUFF_SIZE);
    if (Recv(sockfd, recvBuff, BUFF_SIZE, 0) < 0)
    {
        close(sockfd);                          // 如果接收客户端信息错误
        FD_CLR(sockfd, &watchset);              // 清除描述符
        user[i].sockfd = -1;
    }
    printf("CMD: %s", recvBuff);
    if (strncmp(recvBuff, "REGIST", 6) == 0)
    {
        regist(sockfd);                         //注册用户
    }
    else if (strncmp(recvBuff, "LOGIN", 5) == 0)
    {
        // （省略）判断客户端是否已经登录
        if (login(sockfd) != TRUE)              // 登录
            continue;
        isClientLogin = TRUE;                   // 修改标志位，防止客户端再次登录
    }
    else if (strncmp(recvBuff, "LIST", 4) == 0)
    {
        sendUserList(sockfd);                   // 发送在线用户列表
    }
    else if (strncmp(recvBuff, "CHAT", 4) == 0)
```

```
    {
        Send(sockfd, "NAME: ", 6, 0);
        Recv(sockfd, name, MAX_LEN, 0);        //获取聊天对象的姓名
        if ((dstfd = selChatObj(sockfd, name)) < 0) // 根据姓名获取聊天对象的sockfd
            continue;
        if (sendChatReq(sockfd, dstfd) != TRUE) // 发送聊天请求
            continue;
        transMsg(sockfd, dstfd);
    }
    else if (strncmp(recvBuff, "EXIT", 4) == 0)     // 退出客户端
    {
        delUser(sockfd);                           // 删除用户
        FD_CLR(sockfd, &watchset);                 // 清除描述符
        close(sockfd);                             // 关闭socket
    }
    else
    {
        Send(sockfd, "Cmd error!\n", 11, 0);    // 错误命令：发送提示信息
    }
}
```

4）User Info Initialize

我们用 userInitial()函数来实现结构体数组的初始化。

Name	userInitial
Function	结构体数组初始化
Parameter	
Return Value	

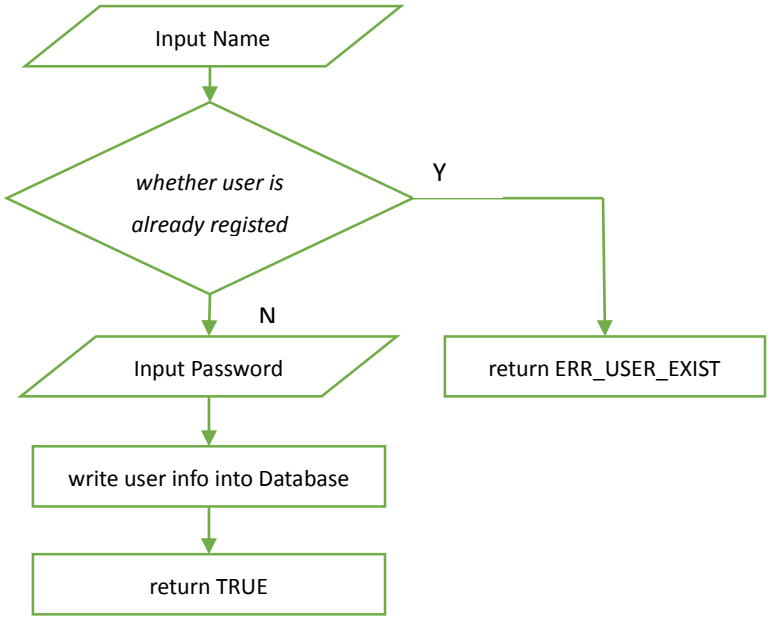
具体代码实现如下：

```
void userInitial()
{
    int i;
    for (i = 0; i < MAX_USER; i++)
    {
        memset(user[i].name, '\0', MAX_LEN);       // 用户名置空
        user[i].status = OFFLINE;                  // 状态为OFFLINE
        user[i].sockfd = -1;                       // 连接描述符不可用
    }
}
```

5）Regist

Name	Regist
Function	注册
Parameter	Sockfd：当前建立连接客户端的连接描述符
Return Value	ERR_USER_EXIST：注册用户已存在 TRUE：注册成功

注册过程的逻辑流程图如下：



根据流程图不难写出代码：

```
int regist(int sockfd)
{
	char recvBuff[BUFF_SIZE];
	char name[MAX_LEN], pwd[MAX_LEN], tmp[MAX_LEN];
	FILE *fp;
	int nRecv;

	Send(sockfd, "NAME: ", 6, 0);
	memset(name, '\0', MAX_LEN);
	nRecv = Recv(sockfd, name, MAX_LEN, 0);
	fp = fopen("userData", "r");                    // 只读方式打开文件，判断要注册用户是否已经被注册
	// （略）文件打开失败的异常检测和处理
	for ( ; fgets(recvBuff, sizeof(recvBuff), fp) != NULL; )
	{
		/* whether user already regist or not */
		if (strncmp(recvBuff, name, nRecv - 1) == 0)
		{
			Send(sockfd, "User exist!\n", 12, 0);
			fclose(fp);
```

```
                return ERR_USER_EXIST;
            }
        }
        fclose(fp);

        memset(tmp, '\0', sizeof(tmp));
        strncpy(tmp, name, nRecv - 1);          /* discard rest of recvBuff: length=MAX_LEN-
nRecv */

        Send(sockfd, "PASS: ", 6, 0);           // 用户名正确，输入密码
        memset(pwd, '\0', MAX_LEN);
        nRecv = Recv(sockfd, pwd, MAX_LEN, 0);

        fp = fopen("userData","a");             // 将注册的用户信息写在文件末尾
        // （略）文件打开失败的异常检测和处理
        memset(recvBuff, '\0', sizeof(recvBuff));
        strcpy(recvBuff, tmp);
        strcat(recvBuff, ":");
        strcat(recvBuff, pwd);
        fputs(recvBuff, fp);                    /* write user info into fp */
        fclose(fp);
        Send(sockfd, "Regist success!\n", 16, 0);   // 打印提示信息并返回
        return TRUE;
}
```

注册成功后，用户名和密码会追加到 userData 文件的末尾。例如，我们注册一个用户：

LOGIN

NAME:zheng

PASS:zheng

注册成功后，userData 文件的内容如下：

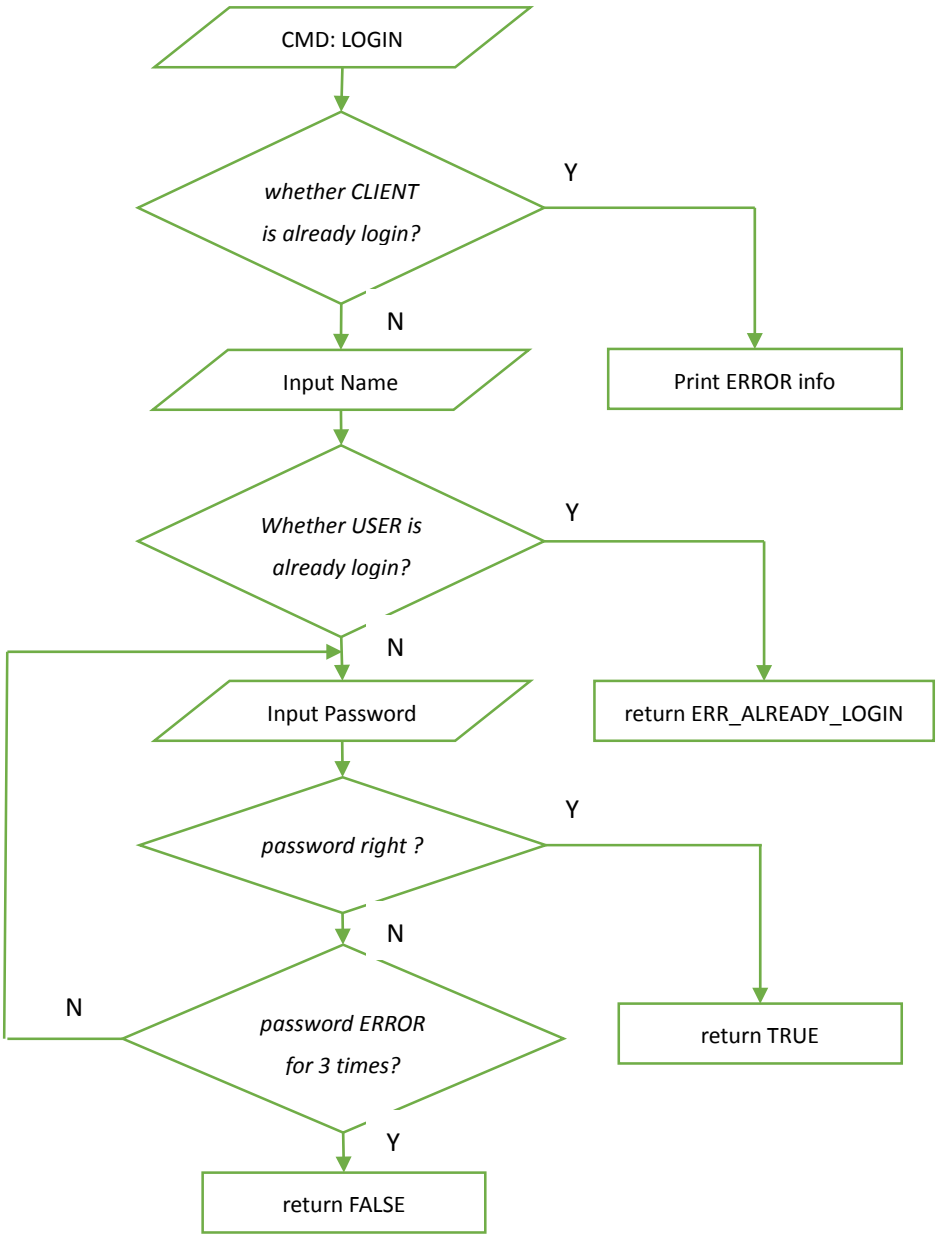
每一行的前半部分（“:”前的部分）为用户名，后半部分为对应的密码。

6）Login

Name	Login
Function	用户登录
Parameter	Sockfd：当前建立连接客户端的连接描述符
Return Value	ERR_USER_NOT_EXIST：用户不存在

	ERR_ALREADY_LOGIN：用户已登录 ERR_PASSWORD：密码错误 TRUE：登录成功

登录过程的逻辑流程图如下所示：




```
int login(int sockfd)
{
	char userName[MAX_LEN], password[MAX_LEN], buff[BUFF_SIZE];
	int nName, nPass, n;
	FILE *fp;
	char *read, *p, *pwd;
```

```
Send(sockfd, "NAME: ", 6, 0);
memset(userName, '\0', sizeof(userName));
nName = Recv(sockfd, userName, sizeof(userName), 0);
for (n = nUser; n != 0; n--)
{
	if (strncmp(user[n].name, userName, nName) == 0 )
	{
		Send(sockfd, "You're already login!\n", 22, 0);
		return ERR_ALREADY_LOGIN;
	}
}
fp = fopen("userData", "r");
if (fp == NULL)
{
	perror("Open file error!");
	exit(1);
}
while ((read = fgets(buff, sizeof(buff), fp)) != NULL) /* read a single line of fp to buff */
{
	p = strchr(buff, ':');
	pwd = p + 1;                                    // 找到密码的起始位置指针
	*p = '\0';
	if (strncmp(userName, buff, nName - 1) == 0)
	{
		p = strchr(pwd, '\n');                      // 扫描下一行
		*p = '\0';
		for (n = 0; n < 3; n++)
		{
			Send(sockfd, "PASS: ", 6, 0);
			memset(password, '\0', sizeof(password));
			nPass = Recv(sockfd, password, sizeof(password), 0);
			if (strncmp(password, pwd, nPass - 1) != 0)
			{
				Send(sockfd, "Password error!\n", 16, 0);
				continue;
			}
			Send(sockfd, "Login success!\n", 16, 0);
			addUser(sockfd, userName);
			fclose(fp);
			return TRUE;
		}
		Send(sockfd, "Password error 3 times, exit!\n", 30, 0);
		fclose(fp);
		return ERR_PASSWORD;
```

```
            }
        }
        if (read == NULL)
        {
            Send(sockfd, "User not exist, Regist first!\n", 30, 0);
            fclose(fp);
            return ERR_USER_NOT_EXIST;
        }
}
```

7）Add User

Name	addUser
Function	将登陆的用户添加进结构体数组中
Parameter	Sockfd：当前建立连接客户端的连接描述符 Name：已登录的客户姓名
Return Value	TRUE：添加成功

```
int addUser(int sockfd, const char *name)
{
    strncpy(user[nUser].name, name, strlen(name)); // 添加到用户数组的末尾
    user[nUser].status = ONLINE;
    user[nUser].sockfd = sockfd;
    nUser++;

    return TRUE;
}
```

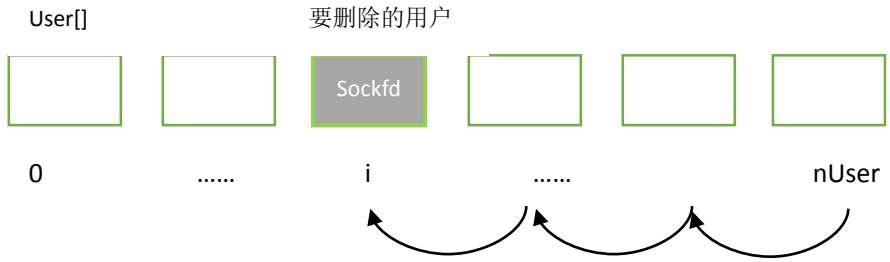
8）Delete User

Name	delUser
Function	从用户链表中删除连接描述符为 sockfd 的用户信息
Parameter	Sockfd：当前建立连接客户端的连接描述符
Return Value	

```
void delUser(int sockfd)
{
    int i;
    for (i = nUser; sockfd != user[i].sockfd; i--); // 找到要删除的用户在数组中的位置
    /* overwrite next user on current position */
    for (; user[i].sockfd != -1 && i < MAX_USER; i++)
    {
        strcpy(user[i].name, user[i + 1].name); // 每一个数组元素都向前移一位
```

```
            user[i].status = user[i + 1].status;
            user[i].sockfd = user[i + 1].sockfd;
        }
        nUser--;                                    // 在线用户数减 1
}
```

实际上，这是一种非常低效的方式。若删除的目标用户在结构体数组的末尾（user[nUser]），查找效率为 O(1)，此时直接将其删除即可；若删除的目标在结构体数组的首位（user[0]），查找时间复杂度为 O(N)，删除该数组元素后，需要将后面的每一个数组元素都向前移一位，时间复杂度为 O(N)。如下图所示：



目标用户的删除过程

后面的分析与思考中我们会给出基于链表的改进方案。

9）Send User List

Name	sendUserList
Function	向连接描述符为 sockfd 的客户端发送在线用户列表
Parameter	Sockfd：当前建立连接客户端的连接描述符
Return Value	ERR_USER_NOT_FOUND：无在线用户 TRUE：发送成功

```
int sendUserList(int sockfd)
{
    int i;
    if (nUser == 0)
    {
        Send(sockfd, "No online user!\n", 16, 0);
        return ERR_USER_NOT_FOUND;
    }
    for (i = 0; i < nUser; i++)                    // 逐一扫描用户数组并发送用户姓名
    {
        Send(sockfd, user[i].name, sizeof(user[i].name), 0);
    }
    return TRUE;
}
```

10）Get Socket Name

Name	getSockName
Function	获取连接描述符为 sockfd 的用户姓名
Parameter	Sockfd: 当前建立连接客户端的连接描述符
Return Value	User[i].name: 连接描述符为 sockfd 的用户姓名（char *）

```
char *getSockName(int sockfd)
{
	int i;
	for (i = nUser; i != 0; i--)			// 逐一顺序扫描用户数组
	{
		if (user[i].sockfd == sockfd)
			return user[i].name;
	}
}
```

11）Select Chat Object

Name	selChatObj
Function	获取 sockfd 的聊天对象 dstName 的连接描述符
Parameter	Sockfd: 当前建立连接客户端的连接描述符 dstNameL: 聊天对象的姓名
Return Value	ERR_USER_NOT_FOUND：无法找到聊天对象 ERR_USER_BUSY：聊天对象正忙 User[i].sockfd: 连接描述符为 sockfd 的用户的聊天对象的连接描述符

```
int selChatObj(int sockfd, const char *dstName)
{
	/* find sockfd of chat object */
	char *recvBuff;
	int i;
	for (i = 0; i < nUser; i++)
	{
		if (strncmp(user[i].name, dstName, sizeof(dstName)) == 0)
		{
			if (user[i].status == CALLING || user[i].status == TALKING)
			{
				sprintf(recvBuff, "%s busy now!", dstName);
				Send(sockfd, recvBuff, sizeof(recvBuff), 0);
				return ERR_USER_BUSY;
			}
			return user[i].sockfd;
```

```
        }
    }
    Send(sockfd, "User not found!\n", 16, 0);
    return ERR_USER_NOT_FOUND;
}
```

12）Send Chat Request

Name	sendChatReq
Function	连接描述符为 srcfd 的用户向连接描述符为 dstfd 的用户发送聊天请求
Parameter	Srcfd: 发起聊天请求的用户的连接描述符 Dstfd: 接受聊天请求的用户的连接描述符
Return Value	TRUE：接受聊天请求 FALSE：拒绝聊天请求

```
int sendChatReq(int srcfd, int dstfd)
{
    char recvBuff[2];
    int n, i, srcP, dstP;
    for (i = 0; i < nUser; i++)                    //扫描用户数组，获取 srcfd 和 dstfd 的索引位置
    {
        if (user[i].sockfd == srcfd)
        {
            srcP = i;
        }
        if (user[i].sockfd == dstfd)
            dstP = i;
    }
Again:
    Send(dstfd,"request to talk, accept?(y/n) ", 30, 0);
    memset(recvBuff, '\0', sizeof(recvBuff));
    n = Recv(dstfd, recvBuff, 2, 0);            // 接收 dstfd 的指令
    if (strncmp(recvBuff, "y\n", n) == 0 || strncmp(recvBuff, "Y\n", n) == 0)
    {
        Send(srcfd, "Accepted!\n", 10, 0); //dstfd 接受聊天请求
        user[srcP].status = user[dstP].status = TALKING;// 设置聊天双方的状态
        return TRUE;
    }
    else if (strncmp(recvBuff, "n\n", n) == 0 || strncmp(recvBuff, "N\n", n) == 0)
    {
        Send(srcfd, "Rejected!\n", 10, 0);  //dstfd 拒绝聊天请求
        return FALSE;
    }
    Else                                        //dstfd 输入指令错误，打印提示信息
```

```
    {
        Send(srcfd, "Command error!\n", 15, 0);
        goto Again;
    }
}
```

13）Transmit Message

Name	transMsg
Function	转发连接描述符为 srcfd 的用户和连接描述符为 dstfd 的用户的聊天消息
Parameter	Srcfd: 发起聊天请求的用户的连接描述符 Dstfd: 接受聊天请求的用户的连接描述符
Return Value	0: 结束聊天

```
int transMsg(int srcfd, int dstfd)
{
    char buff[BUFF_SIZE];                    // buff 存储服务器接收到的聊天消息内容
    fd_set rset;
    int maxfd, errSel;

    FD_ZERO(&rset);
    for ( ; ; )
    {
        memset(buff, '\0', sizeof(buff));    // 将缓冲区置 0
        FD_SET(srcfd, &rset);
        FD_SET(dstfd, &rset);
        maxfd = srcfd > dstfd ? srcfd : dstfd;

        errSel = select(maxfd + 1, &rset, NULL, NULL, NULL); // 等待某个事件发生
        if (errSel < 0)
        {
            perror("sel error!");
            exit(1);
        }
        if (FD_ISSET(srcfd, &rset))          // srcfd 可读：收到来自 srcfd 的聊天消息
        {
            Recv(srcfd, buff, sizeof(buff), 0);  // 将接收的消息保存到 buff 缓冲区
            if (strncmp(buff, "exit", 4) == 0)   // 如果收到的消息是 exit，结束聊天过程
            {
                FD_CLR(srcfd, &rset);
                FD_CLR(dstfd, &rset);
                return 0;
            }
```

```
            Send(dstfd, buff, sizeof(buff), 0);   // 将缓冲区内容发给 dstfd
        }
        if (FD_ISSET(dstfd, &rset))
        {
            Recv(dstfd, buff, sizeof(buff), 0);
            if (strncmp(buff, "exit", 4) == 0)
            {
                FD_CLR(srcfd, &rset);
                FD_CLR(dstfd, &rset);
                return 0;
            }
            Send(srcfd, buff, sizeof(buff), 0);
        }
    }
}
```

4、client.c

我们采用“瘦客户端”的结构，客户端的功能仅仅是接收服务器的消息并打印到终端里和接收终端的内容并发送给服务器。

1）建立连接

建立连接的过程如图 1。代码实现如下：

```
int main(int argc, char *argv[])                    // 运行时需要输入参数（服务器 IP 地址）
{
    // ……
    // （省略）参数错误处理
    /* create a new Socket */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    // （略）异常检测和处理
    /* set address */
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;                //IPv4 protocol
    serverAddr.sin_port = htons(SERV_PORT);   // 设置端口号
    if (inet_pton(AF_INET, argv[1], &serverAddr.sin_addr) <= 0)
    // （略）异常检测和处理
    /* connect */
    int errConnect = connect(sockfd, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
    // （略）异常检测和处理
    /* if object accepted */
    str_cli(stdin, sockfd);                         // 调用函数处理收发问题
    close(sockfd);                                  /* close socket */
    return 0;
```

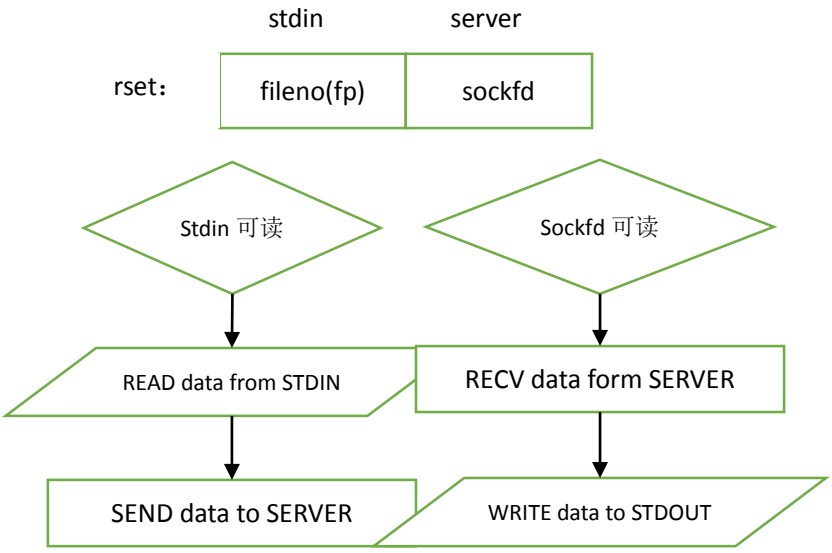
```
}
```

2）String Client

该程序参考《Unix Network Programming (Volume 1: The Sockets Networking API, Third Edition)》6.7 节（p137）的程序修改而来。

```
void str_cli(FILE *fp, int sockfd)
{
    char sendBuff[BUFF_SIZE], recvBuff[BUFF_SIZE];
    int maxfd, errSel, stdinEOF;
    fd_set rset;
    int n;

    stdinEOF = 0;
    FD_ZERO(&rset);
    for ( ; ; )
    {
        memset(sendBuff, 0, sizeof(sendBuff)); // 清空发送缓冲区
        memset(recvBuff, 0, sizeof(recvBuff));

        if (stdinEOF == 0)
            FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfd = fileno(fp) > sockfd ? fileno(fp) : sockfd;
        errSel = select(maxfd + 1, &rset, NULL, NULL, NULL);
        // （略）异常检测和处理
        if (FD_ISSET(sockfd, &rset))            /* socket is readable */
        {
            n = Read(sockfd, recvBuff, BUFF_SIZE);// 从服务器接收消息，写入 recvBuff
            if (n == 0)                         // 套接字读取到 EOF
            {
                if (stdinEOF == 1)              // 如果已在标准输入 stdin 上遇到 EOF
                    return;                     // 正常终止
                else                            // 否则，表示服务器过早终止
                {                               // 打印错误信息 并返回
                    printf("str_cli: server terminated prematurely");
                    return;
                }
            }
            Write(fileno(stdout), recvBuff, n);  // 将 recvBuff 内容输出到 stdout
        }
        if (FD_ISSET(fileno(fp), &rset))        /* input is readable */
        {
            n = Read(fileno(fp), sendBuff, BUFF_SIZE);// 从 IO 缓冲区读取消息到 sendBuff
```

```
                if (n == 0)                         // 标准输入碰到 EOF
                {
                    stdinEOF = 1;                   // 修改标志位
                    shutdown(sockfd, SHUT_WR);// 调用 SHUT_WR 以发送 FIN
                    FD_CLR(fileno(fp), &rset);      // 在 rset 中清除标准输入 fileno(fp)
                    continue;
                }
                Write(sockfd, sendBuff, n) == -1;   // 将 sendBuff 内容发送给服务器
            }
        }
}
```



Str_cli 函数的逻辑结构

5、Makefile

为方便编译，我们编写了一个简单的 Makefile 文件用于文件的编译。

```
# Makefile
# by MarinYoung
# 2013/12/15
all: server.o client.o                          # 生成目标文件
    gcc -o server server.o
    gcc -o client client.o

server.o: server.c unp.h sys.h                  # 编译源码
    gcc -c server.c
client.o: client.c unp.h sys.h
    gcc -c client.c

clean:                                          # 清除目标文件，可执行文件
```

```
    rm -rf server client
    rm -rf *.o
```

编译时执行 make 或者 make all 命令（需要 root 权限）即可生成可执行文件。

Makefile 多用于大型工程，防止重编译。这里只是一个简单的例子，没有定义变量，当文件比较多时，显然是不可取的。事实上，我们可以编写一个简单的 bash 文件来生成可执行文件：

```
# make.sh
#!/bin/bash
gcc -o server server.c
gcc -o client client.c
```

编译时输入命令 bash make.sh 即可。显然，这种方法比 Makefile 简单的多。

三、思考与改进

1、user 存储结构

我们定义了一个结构体

```
struct USER{                                    //: unp.h
    char name[MAX_LEN];
    enum USER_STATRE status;
    int sockfd;
}
```

用以存储在线用户的信息。在 server.c 中定义一个用户数组的全局变量：

```
struct USER user[MAX_USER];                     //: server.c
```

这样做的一个好处是代码实现较为简单，直观明了。但缺点也是显而易见的：

1） 用户数量受限于数组的长度：最多只能存储 MAX_USER 个在线用户；

2） 删除用户时间复杂度太大：这一点在 Delete User 函数的讨论里已经提到，当 MAX_USER 较大时，O(N)的时间复杂度会带来严重的效率问题。

3） 程序运行一开始就为用户分配了 MAX_SUER*sizeof(struct USER)的内存空间，当 nUser<<MAX_USER 时，带来了较大的空间浪费问题。

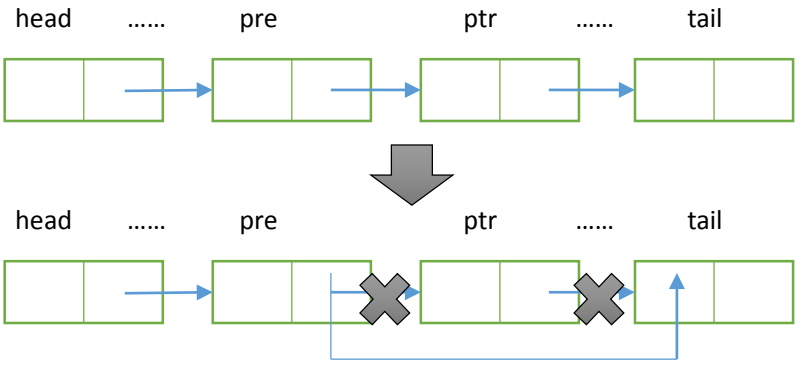
因此，可以考虑使用链表结构存储用户信息（这里使用带头结点的单链表）：

```
struct USER{                                    //: unp.h
    char name[MAX_LEN];
    enum USER_STATE status;
    int sockfd;
    struct USER *next;
}
```

同时，在 server.c 以全局变量的形式中定义链表的头结点：

```
struct USER *head;                              //: server.c
```

这样，删除某个用户节点的逻辑结构图如下所示：



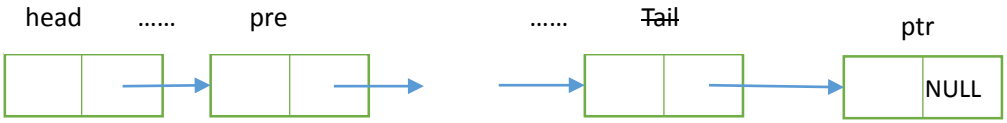
代码实现如下：

```
void delUser(int sockfd)
{
	struct USER *ptr, *pre;
	char *buff;

	for (ptr = pre = head; ptr != NULL; ptr = ptr->next)
	{
		if (ptr->sockfd == sockfd)
		{
			pre->next = ptr->next;			// 图
			ptr->next = NULL;			// 指针指向置为NULL
			free(ptr);				// 释放 ptr 节点
			break;
		}
		pre = ptr;
	}
}
```

采用这种方法，查找时间复杂度不变（依然为 O(N)），删除时间复杂度却降低为 O(1)。添加用户时，每登录一个用户，为其新分配一块内存空间，登出后将其释放掉，提高了空间利用率；

```
int addUser(int sockfd, const char *name)
{
	struct USER *ptr;

	for (ptr = head; ptr != NULL; ptr = ptr->next);		// 指针指向链表末尾
	ptr = (struct USER *)malloc(sizeof(struct USER));	// 分配新的用户空间
	strcpy(ptr->name, name);				// 设置用户信息
	ptr->status = ONLINE;
	ptr->sockfd = sockfd;
	ptr->next = NULL;
}
```

添加新的用户节点后链表结构如下图所示：



其他函数内需要查找目标节点直接从头结点（或者从尾节点）开始从头到尾遍历查找即可，这里不一一列举。

2、文件读取

在登录与注册函数中，我们使用一个 userData 的文件来存储用户信息（主要是用户名和密码）。登录时，以只读方式打开文件读取用户名密码信息；注册成功后，将用户名和密码追加到 userData 的末尾。当两个用户同时注册或登录，即需要同时读取或写入文件时，就会带来冲突问题。解决方法是使用互斥锁或者是信号量机制，当一个用户读取文件时，给文件加锁，不允许其他用户读取，读取完毕后释放锁，即可解决冲突问题。

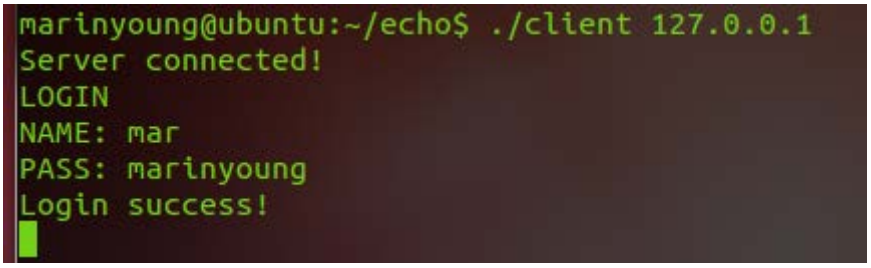
3、用户查找索引

结构体 USER 中定义 name 的最大长度为 MAX_LEN = 15Byte；regist 函数中接收的密码的最大长度为 MAX_LEN=15 个字节。因此在实际存储中无论用户名(name)和密码(pwd)的实际长度是多少，它们在内存中均占用 15 个字节。而我们在查找用户时大多数情况下使用姓名作为索引进行查找：多调用 strcmp()数将目的字符串与当前字符串相比较，返回值为 0 即表示两字符串相等。然而，这里存在一个问题：

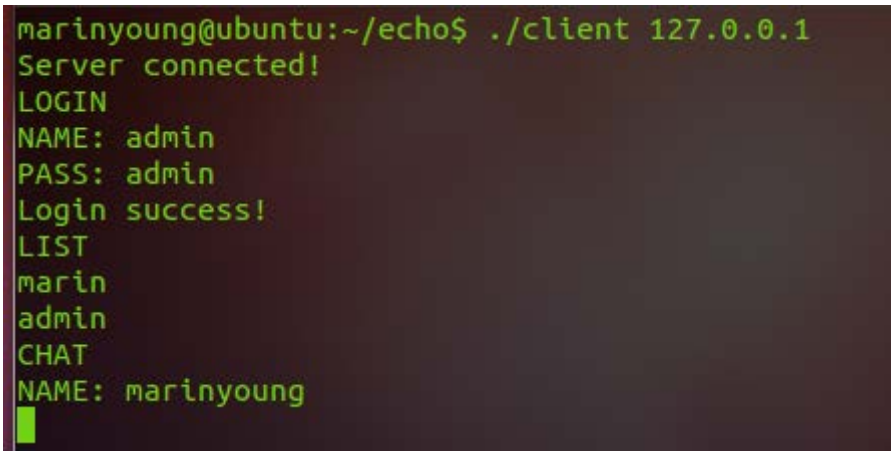
假设 len = sizeof(src) > sizeof(dst) ? sizeof(src) : sizeof(dst);

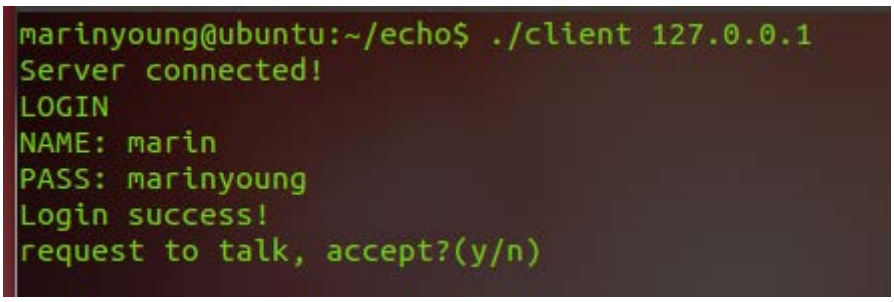
strcmp(const char *src, const char *dst);

比较的是二者的前 len 个字符是否相等。假设用户信息 userData 或者用户数组 user[MAX_USER]中存储的用户姓名为“marin”，这时如果输入的用户姓名字符串是”mar”，strcmp()的返回值依然是 0，即代码中无法区分”marin”与“mar”。例如：（1）userData 中存储的用户名是”marin“，我们输入”mar”依然能够登录成功；（2）LIST 显示的用户有”marin“和”admin”，我们向”marinyoung”发送了聊天请求，收到请求的却是”marin“！



错误演示（1）





错误演示（2）

改进方法是：为每一个用户添加一个唯一的 ID；以 ID 为索引进行查找。具体方法是：每添加一个用户，为其生成一个随机数的 ID，并添加到用户信息表中。

```
#include <time.h>                                   // 使用当前时钟作为随机数种子
#include <stdlib.h>
// ……
srand((unsigned) time(NULL));                       // 初始化随机数
ptr->ID = rand();
// ……
```

另一种改进方法是，把用户姓名的实际长度也保存下来（nLen = recv(sockfd, name, MAX_LEN, 0) ）。比较时，先比较二者长度是否相等（strlen(src) == strlen(dst) ?），若相等，再执行 strcmp(src, dst)。

4、查找效率

这里我们均使用从头到尾遍历查找，时间复杂度为 O(N)。可以考虑使用高级查找算法，如二分查找（时间复杂度 $O(\log_2 N)$）、DFS（时间复杂度 O(N)，需要对目标建树）、建索引散列查找等方法提高查找效率。

5、瘦客户端

我们采取瘦客户端方式进行通信，类似于 telnet 模式，客户端仅负责信息的收发，所有交互过程的处理都是在服务器端进行。当登录的用户较少时，问题不大明显，但当用户较多时，会严重占用服务器资源。因此，这绝对不是一种实用的解决方案。

四、总结

由于时间关系，程序写的比较粗糙，还有很多需要完善的地方。通过实验，我学会了很多东西：

1、 学会 linux 下 socket 程序的编写方法；
2、 利用 strace 跟踪进程执行时的系统调用和所接受的信号，已进行代码的调试；
3、 利用 gdb 命令调试程序；
4、 简单的 Makefile 文件的编写；
5、 熟悉 vim 编辑器的配置及使用。

程序仅仅完成了所要求的功能，还有很多地方还需要优化，同时离实用性还有很大的距离。这里我们只针对已完成部分进行优化，不考虑 GUI、数据库等。今后还应在 GUI 编程方面勤加练习。总的来说，这门课程实验让我学习到了许多全新的知识，并巩固了网络原理的内容，为今后的学习研究奠定了良好的基础。

参考资料

【1】[USA] W.Richard Stevens、Bill Fenner、Andrew M. Rudoff <<Unix Network Programming (Volume 1: The Sockets Networking API, Third Edition)>> Beijing: Post & Telecom Press 2010

【2】[USA] Stephen G. Kochan <<A Complete Introduction to the C Programming Language, Third Edition>> Beijing: Publishing House of Electronics Industry 2006

【3】[USA] Brian W. Kernighan、Dennis M. Ritchie <<The C Programming Language, Second Edition>> Prentice Hall Software Series