

网络安全技术

实 验 报 告

学院：网安学院

年级：2020

班级：信安班

学号：2011395

姓名：魏伯繁

手机号：18611111192

2023 年 4 月 1 日

目录

1 实验目标	2
2 实验内容	2
2.1 DES 简介	2
2.2 Linux 下的 socket 编程	2
3 实验步骤	3
3.1 在 vmware 中配置 C/C++ 环境	3
3.2 DES 实现	3
3.2.1 常数录入	4
3.2.2 设计 DES 轮函数	4
3.2.3 异或运算实现	6
3.2.4 S 盒变换的实现	6
3.2.5 密钥置换	7
3.3 Socket 编程实现基于 TCP 通信	8
3.3.1 socket 创建	8
3.3.2 多线程的创建和编写	9
3.3.3 接口设计	12
3.3.4 对方 ip 地址和接口获取	14
3.4 实验结果展示	14
4 实验遇到的问题及其解决方法	15
4.1 win 程序迁移 linux 问题	15
4.2 DES 加密长度与输入长度的限制	16
4.2.1 段错误（核心已转储）	16
5 实验结论	16

1 实验目标

1. 理解 DES 加解密原理。
2. 理解 TCP 协议的工作原理。
3. 掌握 linux 下基于 socket 的编程方法

2 实验内容

本章训练的要求如下。

1. 利用 socket 编写一个 TCP 聊天程序。
2. 通信内容经过 DES 加密与解密。

本次实验要求使用 DES 对通信双方的通信内容进行解密后传输，接收方在接受到信息后同样使用 DES 进行解密，在本次实验中，我们默认通信双方已经提前交换了密钥并将密钥存储在了和可执行文件同级的 key.txt 中。

本次通信使用 S/C 模型，即客户-服务模型，由服务端首先启动并等待客户端接入，随后客户端输入服务端的 ip 地址以及端口号进行连接，连接成功后双方即可开始通信。

通信双方均可以使用 quit 命令退出通信。

2.1 DES 简介

DES (Data Encryption Standard) 算法是一种用 56 位有效密钥来加密 64 位数据的对称分组加密算法，该算法流程清晰，已经得到了广泛的应用，算是应用密码学中较为基础的加密算法。TCP (传输控制协议) 是一种面向链接的、可靠的传输层协议。TCP 协议在网络层 IP 协议的基础上，向应用层用户进程 供可靠的、全双工的数据流传输。

DES 明文分组长度为 64bit (不足 64 bit 的部分用 0 补齐)，密文分组长度也是 64bit。加密过程要经过 16 圈迭代。初始密钥长度为 64 bit，但其中有 8 bit 奇偶校验位，因此有效密钥长度是 56 bit，子密钥生成算法产生 16 个 48 bit 的子密钥，在 16 圈迭代中使用。解密与加密采用相同的算法，并且所使用的密钥也相同，只是各个子密钥的使用顺序不同。DES 算法的全部细节都是公开的，其安全性完全依赖于密钥的保密。算法包括初始置换 IP、逆初始置换 IP⁻¹、16 轮迭代以及子密钥生成算法

2.2 Linux 下的 socket 编程

Linux 是一种多用户、多进程的操作系统。每个进程都有一个唯一的进程标识符，操作系统通过对机器资源进行时间共享，并发的运行许多进程。在 Linux 中，程序员可以使用 fork() 函数创建新进程，它可以与父进程完全并发的运行，fork() 函数不接受任何参数，并返回一个 int 值。当它被调用时，

创建出的子进程除了拥有自己的进程标识符以外，其余特征，例如数据段，堆栈段，代码段等和其父进程完全相同。

Linux 系统是通过套接字 (socket) 来进行网络编程的。网络程序通过 socket 和其它几个函数的调用，会返回一个通讯的文件描述符，程序员可以将这个描述符看成普通的文件的描述符来操作，这就是 Linux 的设备无关性的好处。通过对描述符的读写操作可以实现网络之间的数据交流。

linux 中进行 socket 编程的流程如下图所示：

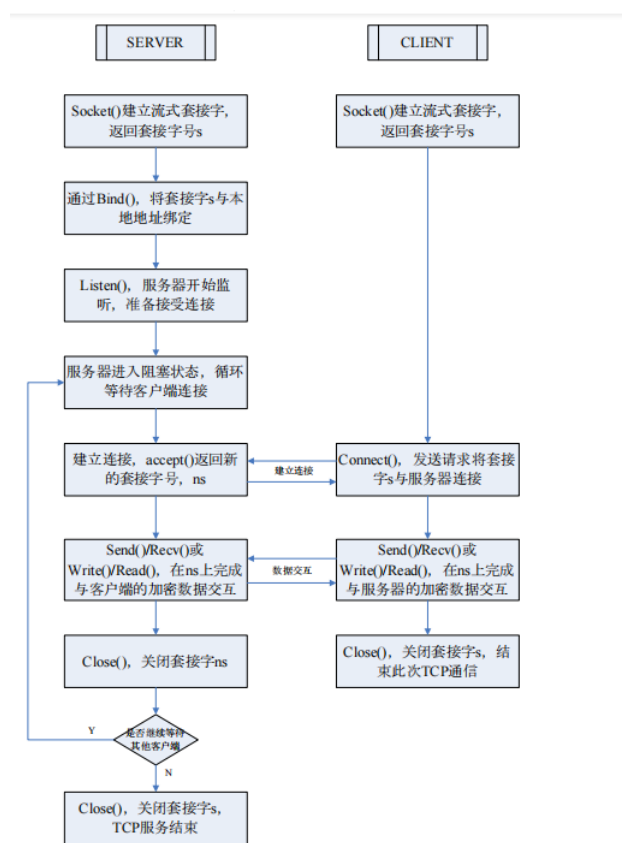


图 2-4 TCP 通信流程图

图 2.1: socket 编程流程图

3 实验步骤

3.1 在 vmware 中配置 C/C++ 环境

首先，我们需要在虚拟机中配置 C/C++ 环境，并进行测试代码的跑通，之前很多课程都已经用到了 VMware 配置虚拟环境，也在虚拟环境中使用了 C/C++ 编写代码，所以这一部分不做详细介绍。

3.2 DES 实现

第二步，我们需要实现一个完整的 DES 函数，这个函数的输入是 16 进制的字符串，输出的也是 16 进制的字符串，为了方便后续处理，我统一将该 16 进制字符串用二进制位来表示，存储在 char* 中。

3.2.1 常数录入

由于 DES 中需要使用非常多的常数, 为了提高程序性能, 将所需常数存储在常量中是一个常用的选择, 下面的代码展示了初始置换 IP 的存储, 将他存储在一个 int 型的数组中, 待需要使用时直接对数组的下标进行操作, 找到对应的置换单元即可

```

1      int IP[] = { 58, 50, 42, 34, 26, 18, 10, 2,
2                60, 52, 44, 36, 28, 20, 12, 4,
3                62, 54, 46, 38, 30, 22, 14, 6,
4                64, 56, 48, 40, 32, 24, 16, 8,
5                57, 49, 41, 33, 25, 17, 9,  1,
6                59, 51, 43, 35, 27, 19, 11, 3,
7                61, 53, 45, 37, 29, 21, 13, 5,
8                63, 55, 47, 39, 31, 23, 15, 7 }; //初始置换 IP
9  }

```

3.2.2 设计 DES 轮函数

DES 的总体加密流程如下图所示: 所以基本的设计思路为, 编写轮函数代码, 然后循环调用 16 次即可。

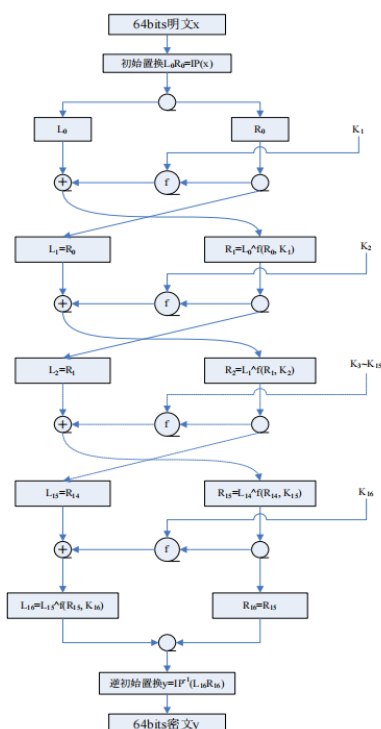


图 3.2: socket 编程流程图

下面的代码展示了加密轮函数的实现，首先完成左右交换后进行扩展置换，随后对密钥进行选取，选取密钥完成后依次完成一个轮函数中应该包含的四个操作：异或操作、S 盒置换、P 置换、异或运算

```

1      void EncryptDESround(int r) {
2          setOriginRight();
3          extendChange();//扩展置换
4          //printcstar64(Right, 1, 33);
5          //printcstar48(Right48, 1, 49);
6          shiftKey(shiftBits[r-1]);//移动密钥
7          getUsedKey();//完成密钥置换 2
8          //printcstar48(Key48, 1, 49);
9          calXOR();
10         //printcstar48(Right48, 1, 49);
11         SBoxChange();
12         //printcstar64(Right, 1, 33);
13         PChange();
14         calXOR2();
15         changeLR();
16     }

```

而对于上层的 DES 来说，其实质就是调用 16 次加密的轮函数，只不过在调用轮函数前需要对明文可密钥进行一些预处理，比如对 64 位密钥进行操作以取出真正有用的 56 位密钥，然后对明文进行初始置换并进行对称的左右分割。

```

1      std::string realencrypt(){
2          loadkey();//载入密钥
3          getRealKey();//产生真的密钥
4          initialChange();//初始置换
5          //现在进入这里就默认 message 中已经填充了 64 位的数据，Key 中填好了 56 位数据
6          setleft(); setright();//填充左右部分明文
7          setLeftKey(); setRightKey();//填充左右部分密文
8          for (int i = 1; i <= 16; i++) {
9              EncryptDESround(i);
10         }
11         finalChangeLR();

```

```
12         reverseInitialChange();
13         std::string s="";
14         for(int i=1;i<=64;i++){
15             s.push_back(message[i]);
16         }
17         return s;
18     }
```

3.2.3 异或运算实现

异或运算时 DES 加密中的一个重要运算，我实现异或运算的思路是：因为我的明文、密钥都是以二进制的形式保存的，中间的结果也是二进制，所以我的异或运算也是基于二进制实现的。

```
1         /计算异或运算
2         void calXOR() {
3             char temp[49];
4             for (int i = 1; i < 49; i++) {
5                 if (Key48[i] == Right48[i]) {
6                     temp[i] = '0';
7                     continue;
8                 }
9                 else {
10                     temp[i] = '1';
11                     continue;
12                 }
13             }
14             for (int i = 1; i < 49; i++) {
15                 Right48[i] = temp[i];
16             }
17         }
```

3.2.4 S 盒变换的实现

不管是 S 盒运算还是 P 置换，其实他们的本质都是一样的，就是查表然后做运算，所以在这里主要介绍 S 盒的实现，根据 S 盒的特点，我们将计算中的中间密文按照 8bits 分组，然后再对 8bits 进行拆分，最后按照其索引要求查找 S 盒结果，最后逐比特填入计算后的结果。

```

1      void SBoxChange() {
2          for (int i = 1; i <= 8; i++) {
3              char temp[7];
4              for (int j = 1; j <= 6; j++) {
5                  temp[j] = Right48[(i - 1) * 6 + j];
6              }
7              int r = getRow(temp[1], temp[6]);
8              int c = getColumn(temp[2], temp[3], temp[4], temp[5]);
9              int result = S_BOX[i - 1][r][c];
10             std::string s = int2string(result);
11             for (int k = 1; k <= 4; k++) {
12                 Right[(i - 1) * 4 + k] = s[k-1];
13             }
14         }
15     }

```

3.2.5 密钥置换

对 DES 加密方法来说，其密钥的轮转也有自己的逻辑，其核心就是密钥左移和密钥的置换，密钥左移的难点主要在于对边界以及数组下标的考虑，所以在这里还是主要查看密钥置换的操作，密钥置换也是将 56 位实际密钥分割成前后两半进行操作，并根据查表 PC 进行最后的结果输出。

```

1      //完成密钥置换 2
2      void getUsedKey() {
3          char temp[57]; //拿到现在的 56 比特
4          for (int i = 1; i < 57; i++) {
5              if (i < 29) {
6                  temp[i] = LeftKey[i];
7              }
8              else {
9                  temp[i] = RightKey[i - 28];
10             }
11         }
12         for (int i = 1; i < 49; i++) {
13             Key48[i] = temp[PC_2[i - 1]];

```



```
14         }  
15     }
```

3.3 Socket 编程实现基于 TCP 通信

之前在计算机网络课程中,我对 Win 上的 Socket 通信已经有了一个比较全面的了解,我发现在 Win 上可以跑通的代码在 linux 环境下并不能跑通,也就是 Linux 环境下的 Socket 编程和 Win 上的 Socket 编程是存在一定差别的。

同样,本次实验我使用了多线程进行收-发操作,linux 上的多线程操作和 windows 上多线程的使用以及创建方法也略有不同,这些都是需要我重新查找资料,重新进行代码重构的方面。

3.3.1 socket 创建

对于 socket 的创建过程,其实与 Win 中的流程是没有区别的,对于服务端来说,需要首先 createSocket,然后对 socket 进行绑定,随后进入 listen,如果 listen 成功监听到那么就执行 accept 函数,随后创建接收-发送线程,由于 TCP 连接会对每一个连接都返回一个 Socket,所以我们要保存这个 socket 的信息,以备后面接收发送信息时使用。

这里 linux 端和 win 端不同的一点就是 linux 端并不存在 socket 类这个东西,而是统一使用 int 代替,这是一个需要注意的地方。

```
1     int Servermain(std::string s,int port){  
2         serv_sock = CreateSock();  
3         struct sockaddr_in serv_addr = CreateSockAddrIn("127.0.0.1",port);  
4         std::cout<<"ip、端口绑定完成"<<std::endl;  
5         //绑定文件描述符和服务器的 ip 和端口号  
6         bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));  
7         //进入监听状态,等待用户发起请求  
8         listen(serv_sock, 5);  
9         //创建专门 accept 的线程  
10        pthread_t acceptPthread;  
11        pthread_create(&acceptPthread,NULL,pthread_accept,NULL);  
12        while(true){  
13            if(quit){  
14                close(serv_sock);  
15                break;  
16            }  
17        }
```

```
17     }
18     std::cout<<" 成功退出"<<std::endl;
19     return 0;
20 }
```

3.3.2 多线程的创建和编写

为了增强软件的可用性，实验中使用了多线程进行收发同时通信，在这里以 server 端为例进行代码编写，client 端与 server 除创建线程的时机不同外其他具体实现基本一致。对于 server 端来讲，创建线程的时机应该在 accept 函数调用成功后，拿到了本次通信的 socket 编码，随后将该 socket 的信息封装在一个 node 结构中，然后作为参数传递给线程。

```
1 void *pthread_accept(void *arg){
2     std::cout<<" 成功创建监听线程"<<std::endl;
3     struct sockaddr_in clientSock;
4     socklen_t clientLen = sizeof(clientSock);
5     //accept
6     int acceptedFd = accept(serv_sock, (sockaddr *)&clientSock, &clientLen);
7     std::cout<<" 成功连接客户端"<<std::endl;
8
9     struct sockaddr_in listendAddr;
10    int listendAddrLen;
11    listendAddrLen = sizeof(listendAddr);
12    if(getsockname(acceptedFd, (struct sockaddr *)&listendAddr, (socklen_t
13    *)&listendAddrLen) == -1){
14        printf("getsockname error\n");
15        exit(0);
16    }
17    printf("client`s address = %s:%d\n", inet_ntoa(listendAddr.sin_addr),
18    ntohs(listendAddr.sin_port));
19    oppo_ip = inet_ntoa(listendAddr.sin_addr);
20    oppo_port = ntohs(listendAddr.sin_port);
21
22
23    Node_server* node=new Node_server();
```

```
24     node->client = clientSock;
25     node->fd = acceptedFd;
26     pthread_t pthreadSend;
27     pthread_t pthreadRecv;
28     pthread_create(&pthreadSend,NULL,pthread_send_server,(void *)node);
29     pthread_create(&pthreadRecv,NULL,pthread_recv_server,(void *)node);
30
31     while(true){
32         if(quit){
33             close(acceptedFd);
34             break;
35         }
36     }
37     return nullptr;
38 }
```

其中，node 类的定义如下所示：其保存的内容即为 sock 的地址以及其 socket 编号

```
1     class Node_server{
2     public:
3         struct sockaddr_in client;
4         int fd;
5     };
```

收发线程的代码如下所示：对于发送线程而言，需要考虑的一点就是如果用户输入的太多，应该拒绝发送并将结果告知用户，随后将用户的内容处理加密后发送。如果发送的是 quit，则将一全局变量置为 true 后退出。

对于接收线程，将接受到的二进制串传入解密函数解密后打印输出在命令行上，如果收到的内容为 quit 则同样置全局变量为 true 随后退出。

```
1     void *pthread_send_server(void* arg){
2         std::cout<<" 成功创建发送线程"<<std::endl;
3         char buffer[7500];
4         Node_server*node = (Node_server*)arg;
5         while(true){
```

```
6         if(quit){
7             break;
8         }
9         std::cin.getline(buffer,900);
10        //getchar();
11        std::string s = buffer;
12        std::string fin = "quit";
13        if(s.find(fin)!=std::string::npos){
14            quit=true;
15        }
16        if(s.size()>900){
17            std::cout<<" 输入过长, 请重新输入"<<std::endl;
18            continue;
19        }
20        //现在 s 和 buffer 中存储了同样的内容, 要把这个内容传送给加密函数加密。
21        s = EncryptForTCP(s); //对输入的内容进行 DES 加密
22        memset(buffer,0,7500);
23        memcpy(buffer,s.c_str(),s.size());
24        int ret = send(node->fd,buffer,7500,0);
25        if(ret==-1){
26            std::cout<<" 发送失败"<<std::endl;
27            //std::cout<<errno<<std::endl;
28        }else{
29            //std::cout<<" 发送成功"<<std::endl;
30        }
31        memset(buffer,0,7500);
32    }
33    return nullptr;
34 }
35
36 void *pthread_recv_server(void* arg){
37     std::cout<<" 成功创建接收线程"<<std::endl;
38     char buffer[7500];
39     int sockfd = ((Node_server*)arg)->fd;
40     while(true){
```

```

41         int ret = recv(sockfd,buffer,7500,0);
42         if(ret>0){
43             std::string s = buffer;
44             //std::cout<<s<<std::endl;
45             printf("Recv a Message from %s:%d ", oppo_ip.c_str(), oppo_port);
46             s = DecryptForTCP(s);
47             std::string fin = "quit";
48             if(s.find(fin)!=std::string::npos){
49                 std::cout<<"recv quit"<<std::endl;
50                 quit=true;
51             }
52             std::cout<<s<<std::endl;
53         }
54         memset(buffer,0,7500);
55         if(quit){
56             break;
57         }
58     }
59     return nullptr;
60 }
61

```

3.3.3 接口设计

为了简化实验设计流程，在本次实验中采用接口的设计思想，即对于 socket 通信来说，他并不需要知道 DES 是怎么执行的，他只需要将收到的二进制串交给解密函数，解密函数会将二进制串进行分割（DES 对 64 位进行操作，如果输入大于 64 位需要分割，不足 64 位需要补足）、调整后调用 DES 的解密函数，并且将解密后的字符串返回给 socket 线程。同理，发送线程也不需要知道 DES 是怎么工作的，他只需要将用户的输入当做参数传入加密函数以获取对应的加密后的二进制字符串随后进行发送。

```

1     std::string EnryptForTCP(std::string s){
2         memset(cinedMessage,0,120);
3         memset(allMessage,0,960);
4         memset(message,0,65);

```

```
5         int i;
6         for(i=0;i<s.size();i++){
7             std::string temp = C2B(s[i]);
8             memcpy(allMessage+(8*i),temp.c_str(),8);
9         }
10        //printCstar(allMessage,i*8);
11        i=0;//赋予一个新值
12        int len=getAllMessageLen();
13        //std::cout<<"----比特数为: "<<len<<"----"<<std::endl;
14        std::string ss = "";
15        while (true)
16        {
17            memset(message,0,65);
18            if(i*64+64<len){
19                memcpy(message+1,allMessage+(i*64),64);
20                //完成加密动作
21                //std::cout<<"----第"<<i+1<<" 轮加密 ----"<<std::endl;
22                ss.append(realencrypt());
23                i++;
24                continue;
25            }else{
26                //每一次 fillAllMessage 函数都会把内存全都置零
27                memcpy(message+1,allMessage+(i*64),64);
28                for(int j=1;j<=64;j++){
29                    if(int(message[j])==0){
30                        message[j]='0';
31                    }
32                }
33                //完成加密动作
34                //std::cout<<"----第"<<i+1<<" 轮加密 ----"<<std::endl;
35                ss.append(realencrypt());
36                i++;
37                break;
38            }
39        }
```

```
40     return ss;
41 }
```

3.3.4 对方 ip 地址和接口获取

根据实验要求, 需要我们在聊天的过程中展示出对方的 ip 地址和端口号, 那么我们就需要用到 `getsockname` 函数来获取对方的 ip 地址和端口号, 获取成功后就可以将该值赋值给 `oppo_ip` 和 `oppo_port`, 在打印收到的消息时对这两个变量进行打印即可。具体代码如下图所示。

```
1     struct sockaddr_in listendAddr;
2     int listendAddrLen;
3     listendAddrLen = sizeof(listendAddr);
4     if(getsockname(acceptedFd, (struct sockaddr *)&listendAddr, (socklen_t
5     *)&listendAddrLen) == -1){
6         printf("getsockname error\n");
7         exit(0);
8     }
9     printf("client`s address = %s:%d\n", inet_ntoa(listendAddr.sin_addr),
10    ntohs(listendAddr.sin_port));
11    oppo_ip = inet_ntoa(listendAddr.sin_addr);
12    oppo_port = ntohs(listendAddr.sin_port);
```

3.4 实验结果展示

下图展示了将 `chat.cpp` 进行编译后执行的截图, 左侧是 server 段, 在创建时给出了其使用的 ip 地址以及端口后, 右侧是 client 端, client 端可以根据左侧服务端输出的内容进行连接。

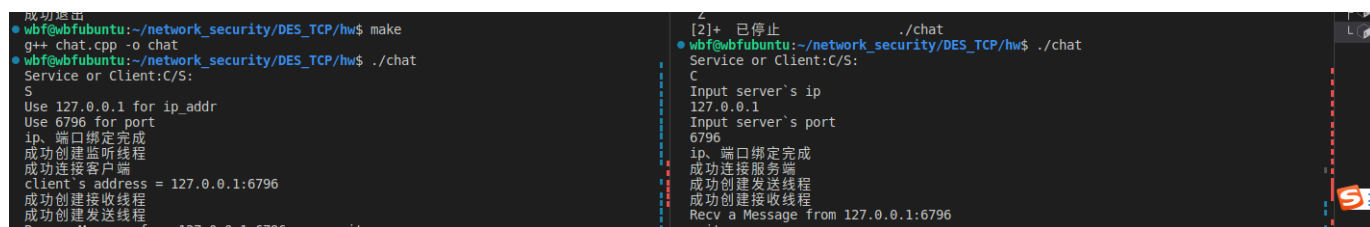


图 3.3: 初始化连接

下面这张图展示了两个用户的聊天情景, 并且在客户端主动输入 `quit` 的情况下双方均终止了聊天。

```
Recv a Message from 127.0.0.1:6796 hi Iam client
hello-I'm server
Recv a Message from 127.0.0.1:6796 nice to mmet you
me2
Recv a Message from 127.0.0.1:6796 recv quit
quit
成功退出
wbf@wbfubuntu:~/network_security/DES_TCP/hw$
```

图 3.4: 聊天内容展示

经过 vscode 上的测试,基本排除了程序的 bug,那么我们现在可以在虚拟机上运行程序看一下效果了:

```
wbf@wbfubuntu: ~/network_security/DES_TCP/hw
wbf@wbfubuntu:~/network_security/DES_TCP/hw$ ./chat
Service or Client:C/S:
S
Use 127.0.0.1 for ip_addr
Use 6796 for port
ip、端口绑定完成
成功创建监听线程
成功连接客户端
client's address = 127.0.0.1:6796
成功创建接收线程
成功创建发送线程
Recv a Message from 127.0.0.1:6796 hi,i am client!
hi,i am Server!
Recv a Message from 127.0.0.1:6796 nice 2 meet you!
me 2
Recv a Message from 127.0.0.1:6796 have a nice day bye~
bye~
Recv a Message from 127.0.0.1:6796 recv quit
quit
成功退出
wbf@wbfubuntu:~/network_security/DES_TCP/hw$
```

图 3.5: 实验效果图

4 实验遇到的问题及其解决方法

在本次实验中,我在编程过程中确实是遇到了一些问题,下面我列出了一些具有代表性的问题并给出了我的解决办法。

4.1 win 程序迁移 linux 问题

本次实验涉及到的一些内容其实在上学期计算机网络或者密码学课程中都有所涉及,我也编写了其中的一些部分代码,但是现在的问题是上学期的代码都是基于 windows 编写的,linux 在头文件、函数定义、变量类型等方面都和 win 有一定差别。

所以如果想要使用上学期的代码框架的话需要做代码的重构,我的解决办法是在网络上搜索对应的资料,比如搜索基于 linux 环境的 socket 编程,最终找到了一篇质量很高的博客作为参考,将不兼容的函数或者变量进行重写:

<https://blog.csdn.net/hguisu/article/details/7445768>

<https://blog.csdn.net/mybelief321/article/details/9377379>

4.2 DES 加密长度与输入长度的限制

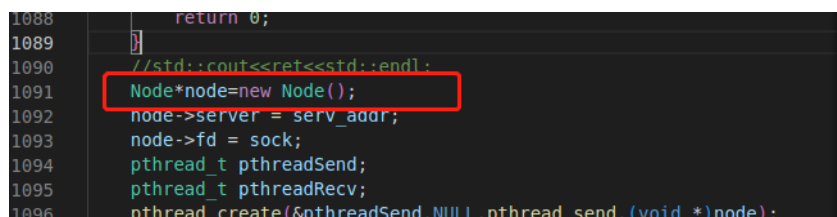
根据 DES 的特性我们知道，DES 每次只能够加密 64 比特量级的数据，而且是以比特位单位进行运算，所以我需要设计出一个良好的转换方式，我的转换方式是将英文单词转换成 asc 码的形式，然后将 asc 码对应的十进制转换成二进制进行加密，解密后对应的二进制数据也做相反的操作即可。

当输入不够时就对二进制数据补零即可。并且我也对输入的字符数做了限制，做多不能输入超过 7500bit 的数据，其中 7500bit 表示为将英文字符转换为二进制后的大小，多余的部分将被直接丢弃掉。

4.2.1 段错误（核心已转储）

这是在 linux 上开发经常遇到的一个问题，我在本次实验中也遇到了这个问题，由于未给出具体的错误行数，本次实验又是多线程，所以 debug 难度比较大，我的解决方法是使用命令行输出的方式逐步精准定位错误的行数，然后通过查阅资料得知，段错误的常见原因是访问了某些不应该被访问的内存空间，类似于数组越界等。

有了上面的两点支持，我首先根据命令行的输出将错误聚焦在了客户端创建两个线程之前发生的错误，然后我着重检查了一下各种数组的下标访问控制问题，检查后发现并没有问题，于是我再次阅读着 Socket 代码，发现原来是因为我只声明了一个指针对象，但没有 new 他，而直接为他赋值了，因为没有为他分配任何空间，所以会报错。



```
1088         return 0;
1089
1090         //std::cout<<ret<<std::endl;
1091         Node*node=new Node();
1092         node->server = serv_addr;
1093         node->fd = sock;
1094         pthread_t pthreadSend;
1095         pthread_t pthreadRecv;
1096         pthread_create(&pthreadSend,NULL,pthread_send,(void *)node);
```

图 4.6: 错误代码位置

5 实验结论

通过本次实验，我回顾了理论课上张老师讲授的 DES 密码基本流程，并且通过动手编写代码实现了 DES 加密函数，进一步理解了扩散和混淆在分组密码中的具体应用。同样，在 linux 上实现 socket 编程也让我进一步熟悉了 linux 系统的使用，明确了 linux 与 win 在编写代码上的区别，为日后实现功能更加强大、更加稳定的代码奠定了良好的基础。

通过学习老师在雨课堂是提供的第一次实验相关 pdf 参考知识，我也学习到了当前 DES 的一些缺点，比如差分攻击和线性攻击就可以以较高的概率攻破 DES，于是现在人们一般使用 3DES 或者 AES 进行加密。