目录

第一章 网络安全技术的特点

1.1 网络安全与现代社会安全的关系

生活在现实世界的人类创造了网络虚拟社会的繁荣，同时也造成了网络虚拟社会的问题。现实世界中真善美的东西，网络的虚拟社会都有。同样，现实社会中丑陋的东西，网络的虚拟社会一般也会有，只是表现形式不一样。如果透过复杂的技术术语和计算机屏幕，人们会发现：计算机网络的虚拟社会和现实社会之间，在很多方面都存在着“对应”关系。现实社会中人与人在交往中形成了复杂的社会与经济关系，在网络社会中，这些社会与经济关系以数字化的形式延续着。

网络安全是现实社会安全的反映。网络安全问题实际上是个社会问题，光靠技术来解决这些问题是不可能的。网络安全是一个系统的社会工程，它涉及技术、政策、道德与法律法规等多方面。

1.2 网络安全与信息安全的关系

应用是网络存在和发展的理由。所有的信息系统与现代服务业都是建立在计算机网络与 Internet 环境之中的。正是由于这个原因，可以说网络应用系统的安全都是建立在计算机网络安全的基础之上的。

用户的各种信息被保存在不同类型的应用系统之中，这些应用系统都是建立在不同的计算机系统之中的。计算机系统包括硬件、操作系统、数据库系统等，它们是保证各类信息系统正常运行的基础。而运行信息系统的大型服务器或服务器集群及用户的个人计算机都是以固定或移动的方式接入到计算机网络与 Internet 中的。任何一种网络功能的服务实现都需要通过网络在不同的计算机系统之间多次进行数据与协议信息交换。

病毒、木马、蠕虫、脚本攻击代码等恶意代码利用 E-mail、FTP 与 Web 系统进行传播，网络攻击、网络诱骗、信息窃取也都是在网络环境中进行的。网络安全是信息系统安全的基础，不能保证网络的安全性，信息系统的安全性就无从谈起。因此，网络安全研究是信息安全研究的重要组成部分，也是信息安全研究的基础。

1.3 网络安全与网络新技术的关系

按照正常人的思维方式，一位技术人员在研究和开发一种基于网络的新的应用技术与系统时，只会想到这种应用可以给人们的生活和工作带来什么样的好处和乐趣，一般不会想到黑客或居心不良的人会利用这种技术做什么坏事。而黑客恰恰是一类逆向思维和不按正常规律办事的人，他们不遵守正常人所遵循的道德规范，“Everything over IP, IP over everything.”说明了计算机网络技术的成功，但是它所带来的问题也是网络技术人员始料未及的。P2P 是一种十分有价值的网络应用模式，但是 P2P 除了可以方面信息共享之外，同时也给恶意代码的传播提供了一种新的途径。手机病毒的出现与无线射频标识 RFID 芯片可能感染病毒的研究结果公布，表明移动设备将成为黑客和恶意软件编写者下一个主攻的目标。

网络技术不是在真空之中，计算机网络是要提供给全世界的用户使用的，网络技术人员在研究和开发一种新的基于网络的应用技术与系统时，必须面对这样一个复杂的局面，成功的网络应用技术与成功的应用系统的标志是功能性与安全性的统一。网络安全问题不应该简单地认为是从事网络安全技术工程师的事，也是每位信息技术领域的工程师与管理人员需要共同面对的问题。

1.4 网络安全与密码学的关系

密码学是信息安全研究的重要工具，密码学在网络安全中有很多重要的应用，但是网络安全涵盖的问题远远超出了密码学涉及的范围。人们对密码学与网络安全的关系的认识有一个过程，这个问题可以用美国著名的密码学专家 Bruce Schneier 在《Secrets and Lies:Digital Security in a Networked World》一书的前言中讲述的观点来说。Schneier 说过：我描述了一个数学的乌托邦：密码算法能将你最深的秘密保持数千年。但是，他现在认为：“事实并非如此，密码学并不能做那么多的事。”密码学并非存在于真空之中。密码学是数学的一个分支，它涉及数学、公式与逻辑。数学是完美的，而现实社会却无法用数学精确地描述。数学是精确的和遵循逻辑规律的，而计算机和网络安全涉及的是人所不知道的事，人与人之间的关系以及人与机器之间的关系。

密码学是研究网络安全所必需的一个重要的工具和方法，但是网络安全研究涉及的问题要广泛得多。

第二章 基于 DES 加密的 TCP 聊天程序

2.1 目的与需求

DES(Data Encryption Standard)算法是一种典型的对称分组加密算法，也是应用密码学中最基本的加密算法之一，目前广泛应用于网络通信加密、数据存储加密、口令与访问控制系统之中。

设计基于 DES 加密的 TCP 聊天程序的需求主要在于：在 Linux 环境下利用 socket 编写一个 TCP 聊天程序，网络传输中的数据通过 DES 算法进行加密。

2.2 DES 算法的基本内容

DES 算法包括初始置换 IP、逆初始置换IP^{-1}、16 圈迭代以及子密钥生成算法。

2.2.1 初始置换 IP

将 64bit 的明文重新排列，而后分成左右两块，每块 32bit，分别用L_0和R_0表示，IP 置换表如表 1 所示。通过对该表进行观察可以发现其中相邻两列的元素位置号数相差 8，前 32 个元素均为偶数号码，后 32 个均为奇数号码，这样的置换相当于将明文的各字节按列写出，各列经过偶采样置换后，再对其进行逆序排列，将阵中元素按行读出以便构成置换的输出。

表 1 IP 置换表

58	50	42	34	26	18	10	2

2.2.2 逆初始置换IP^{-1}

在 16 圈迭代之后，将左右两端合并为 64bit，进行逆初始置换IP^{-1}，得到输出的 64bit 密文，如表 2 所示。

表 2　逆初始置换表

输出的 64bit 为表中元素按行读出的结果。

IP 和IP^{-1}的输入与输出是已知的一一对应关系，它们的作用在于打乱原来输入的 ASCII 码顺序，并将原来明文的校验位p_8，p_{16}，……，p_{64}变为 IP 输出的一个字节。

2.2.3 16 圈迭代

16 圈迭代是 DES 算法的核心部分。将经过 IP 置换后的数据分成 32bit 的左右两段，进行 16 圈迭代，每轮迭代只对右边的 32bit 进行一系列的加密变换，在一轮加密变换结束时，将左边的 32bit 与右边进行异或后得到的 32bit，作为下一轮时右边的段，并将这轮迭代中的右边段未经任何加密变换时的初始值直接作为下一轮迭代时左边的段，这需要在每轮迭代开始时，先将右边段保存一个副本，以便在该轮迭代结束时，将该副本直接赋值给下一轮迭代的左边段。在每轮迭代时，右边的数据段要经过的加密运算包括选择扩展运算 E、密钥加运算、选择压缩运算 S，这些变换合称为 f 函数。

（1） 选择扩展运算

选择扩展运算（也称为 E 盒）的目的是将输入的右边 32bit 扩展成为 48bit 输出，其变换表如表 3 所示。置换结果按行输出的结果即为密钥加运算 48bit 的输入。

表 3

32					

（2） 密钥加运算

密钥加运算，是将选择扩展运算输出的 48bit 作为输入，与 48bit 的子密钥进行异或运算，异或的结果作为选择压缩运算（S 盒）的输入。

（3） 选择压缩运算

选择压缩运算（S 盒）是 DES 算法中唯一的非线性部分，它是一个查表运算，共有 8 张非线性的变换表，如表 4 至表 11，每张表的输入为 64bit，输出为 64bit。在查表之前，将密钥加运算的输出作为 48bit 的输入，将其分为 8 组，每组 6bit，分别进入 8 个 S 盒进行运算，得出 32bit 的输出

结果作为置换运算的输入。

表 4 选择压缩运算变换表 1

	1	2	3	4	5	6	7	8
1-8	0xe	0x0	0x4	0xf	0xd	0x7	0x1	0x4
9-16	0x2	0xe	0xf	0x2	0xb	0xd	0xb	0xe
17-24	0x3	0xa	0xa	0x6	0x6	0xc	0xc	0xb
25-32	0x5	0x9	0x9	0x5	0x0	0x3	0x7	0x8
33-40	0x4	0xf	0x1	0xc	0xe	0x8	0x8	0x2
41-48	0xd	0x4	0x6	0x9	0x2	0x1	0xb	0x7
49-56	0xf	0x5	0xc	0xb	0x9	0x3	0x7	0xe
57-64	0x3	0xa	0xa	0x0	0x5	0x6	0x0	0xd

表 5 选择压缩运算变换表 2

	1	2	3	4	5	6	7	8
1-8	0xf	0x	0x	0x	0x	0x	0x	0x
9-16	0x6	0x	0x	0x	0x	0x	0x	0x
17-24	0x9	0x	0x	0x	0x	0x	0x	0x
25-32	0xc	0x	0x	0x	0x	0x	0x	0x
33-40	0x0	0x	0x	0x	0x	0x	0x	0x
41-48	0xa	0x	0x	0x	0x	0x	0x	0x
49-56	0x5	0x	0x	0x	0x	0x	0x	0x
57-64	0x9	0x	0x	0x	0x	0x	0x	0x

表 6 选择压缩运算变换表 3

	1	2	3	4	5	6	7	8
1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x

33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x
57-64	0x	0x	0x	0x	0x	0x	0x	0x

表 7 选择压缩运算变换表 4

	1	2	3	4	5	6	7	8
1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x
33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x
57-64	0x	0x	0x	0x	0x	0x	0x	0x

表 8 选择压缩运算变换表 5

	1	2	3	4	5	6	7	8
1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x
33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x
57-64	0x	0x	0x	0x	0x	0x	0x	0x

表 9 选择压缩运算变换表 6

	1	2	3	4	5	6	7	8

1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x
33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x
57-64	0x	0x	0x	0x	0x	0x	0x	0x

表 10 选择压缩运算变换表 7

	1	2	3	4	5	6	7	8
1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x
33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x
57-64	0x	0x	0x	0x	0x	0x	0x	0x

表 11 选择压缩运算变换表 8

	1	2	3	4	5	6	7	8
1-8	0x	0x	0x	0x	0x	0x	0x	0x
9-16	0x	0x	0x	0x	0x	0x	0x	0x
17-24	0x	0x	0x	0x	0x	0x	0x	0x
25-32	0x	0x	0x	0x	0x	0x	0x	0x
33-40	0x	0x	0x	0x	0x	0x	0x	0x
41-48	0x	0x	0x	0x	0x	0x	0x	0x
49-56	0x	0x	0x	0x	0x	0x	0x	0x

57-64	0x	0x	0x	0x	0x	0x	0x	0x

S 盒算法流程如下。假设输入的 48bit 为$R_1R_2R_3\cdots R_{47}R_{48}$，需要将其转换为 32bit 值，先把输入值视为由 8 个 6bit 的二进制块组成，如下所示。

a = $a_1a_2a_3a_4a_5a_6$ = $R_1R_2R_3R_4R_5R_6$

b = $b_1b_2b_3b_4b_5b_6$ = $R_7R_8R_9R_{10}R_{11}R_{12}$

c = $c_1c_2c_3c_4c_5c_6$ = $R_{13}R_{14}R_{15}R_{16}R_{17}R_{18}$

d = $d_1d_2d_3d_4d_5d_6$ = $R_{19}R_{20}R_{21}R_{22}R_{23}R_{24}$

e = $e_1e_2e_3e_4e_5e_6$ = $R_{25}R_{26}R_{27}R_{28}R_{29}R_{30}$

f = $f_1f_2f_3f_4f_5f_6$ = $R_{31}R_{32}R_{33}R_{34}R_{35}R_{36}$

g = $g_1g_2g_3g_4g_5g_6$ = $R_{37}R_{38}R_{39}R_{40}R_{41}R_{42}$

h = $h_1h_2h_3h_4h_5h_6$ = $R_{43}R_{44}R_{45}R_{46}R_{47}R_{48}$

其中 a、b、•••、h 都是 6 位，故其十进制范围为 0~63，将转换后的十进制数值加一与对应表中的十六进制数值对应，查表得到 8 个 4bit 的结果，将其串在一起的 32bit 结果作为置换运算的输入。其中，a 对应表 4，b 对应表 5，以此类推。

（4） 置换运算

置换运算 P 是一个 32bit 的换位运算，对选择压缩运算输出的 32bit 数据按表 12 进行换位。

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

至此，最终获得的 32bit 数据，即为此轮迭代的输出。此输出与左边的 32bit 进行异或作为下一轮的右边段，进行加密运算前的原始的右边段作为下一轮的左边段。

2.2.4 子密钥生成

64bit 初始密钥经过置换选择 PC-1、循环左移运算 LS、置换选择 PC-2，产生 16 圈迭代所用到的子密钥k_i。初始密钥的第 8、16、24、32、40、48、56、64 位是奇偶校验位，其余 56 位为有效位。

2.3 TCP 协议

TCP 协议是一种面向连接、面向字节流的可靠传输层协议。两台采用 TCP 协议通信的计算机首先要建立 TCP 连接。TCP 协议以它自己的方式缓存数据，缓存过程对程序员和用户是透明的。TCP 协议采用 piggybacking ACK 的方法，允许双方同时发送数据。TCP 规定了报文段的最大报文段长度（MSS），默认的 MSS 值为 536 个字节。

2.4 套接字

套接字（socket）是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。它把复杂的 TCP/IP 协议族隐藏在 socket 接口的背后，通过调用简单的 socket 函数完成特定协议的数据传输任务。TCP/IP 提供了 3 中类型的套接字。

（1） 流式套接字（SOCK_STREAM）

流式套接字是面向连接的，可靠的数据传输服务，可保证无差错、无重复且按发送顺序提交给接收方。流式套接字在传输层使用 TCP 协议。

（2） 数据报套接字（SOCK_DGRAM）

数据报套接字提供无连接服务，数据以独立的数据报形式被传送，并且在传输过程中没有差错控制和流量控制，数据可能丢失、重复或者乱序。数据报套接字在传输层使用 UDP 协议。

（3） 原始套接字（SOCK_RAW）

原始套接字允许对较低层协议（如网络层的 IP、ICMP）直接进行访问。用于实现自己定制的协议或者对数据报作较低层的控制。

2.5 TCP 通信的相关函数

Linux 系统通过 socket 来进行网络编程。网络程序通过 socket 和其他几个函数的调用，会返回一个通信的文件描述符，程序员可以将这个描述符看成普通的文件描述符来操作，通过对描述符的读写操作可以实现网络中计算机之间的数据传输。这充分体现出 Linux 操作系统的设备无关性的优点。

2.5.1 socket 函数

```
int socket(int domain, int type, int protocol);
```

该函数用于创建通信的套接字，并返回该套接字的文件描述符。

2.5.2 bind 函数

```
int bind(int sockfd, const struct sockaddr* my_addr, socklen_t addrlen);
```

该函数用于将套接字与指定端口的相连。

sockaddr 结构体的定义如下：

```
struct sockaddr
{
    unsigned short sa_family;
    char           sa_data[4];
};
```

不过由于系统的兼容性，一般不用这个头文件，而使用另外一个结构（struct sockaddr_in）来代替。sockaddr_in 的定义如下：

```
struct sockaddr_in
{
    unsigned short      sin_family;
    unsigned short int sin_port;
    struct in_addr      sin_addr;
    unsigned char       sin_zero[8];
};
```

2.5.3 listen 函数

```
int listen(int sockfd, int backlog);
```

该函数用于实现服务器等待客户端请求的功能。

2.5.4 accept 函数

```
int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen);
```

该函数用于处于监听状态的服务器，在获得客户机连接请求后，会将其放置在等待队列中，当系统空闲时，服务器用该函数接受客户机的连接请求。

2.5.5 connect 函数

```
int connect(int sockfd, const struct sockaddr* serv_addr, socklen_t* addrlen);
```

该函数用于客户端向服务器发出连接请求。

2.5.6 write 函数

```
ssize_t write(int fd, const void* buf, size_t nbytes);
```

该函数用于服务器和客户端建立连接后，将 buf 中 nbytes 字节的内容写入文件描述符。

2.5.7 read 函数

```
ssize_t read(int fd, void* buf, size_t nbytes);
```

该函数用于从文件描述符 fd 中读取内容。

2.5.8 send 函数

```
ssize_t send(int s, const void* buf, size_t len, int flags);
```

该函数的作用基本同 write 函数相同，用于将信息发送到指定的套接字文件描述符中，其功能比 write 函数更为全面。

2.5.9 recv 函数

```
ssize_t recv(int s, void* buf, size_t len, int flags);
```

该函数的作用基本同 read 函数相同，用于从指定的套接字中获取信息。

2.5.10 close 函数

该函数用于关闭套接字，其调用形式为：close(sockfd)。

第三章 DES 加密解密设计

3.1 DES 类的定义

在 DES 部分的实现代码中，首先定义封装 DES 操作的类，CDesOperate，类的私有成员包括生成的 16 圈迭代密钥，初始密钥以及加密、解密流程中用到的四个函数。公有成员包括构造函数、析构函数以及根据上述 4 个函数封装的加密函数与解密函数，以方便调用。

```
typedef    int INT32;
```

```
class CDesOperate
{
private:
    ULONG32 m_arrOutKey[16][2];/*输出的 key*/
    ULONG32 m_arrBufKey[2];/*形成起始密钥*/

    INT32 MakeData(ULONG32    *left ,ULONG32    *right ,ULONG32 number);
    INT32 HandleData(ULONG32 *left , ULONG8 choice);
    INT32 MakeKey(    ULONG32 *keyleft,ULONG32 *keyright ,ULONG32 number);
    INT32 MakeFirstKey( ULONG32 *keyP );
public:
    CDesOperate();
    ~CDesOperate();
    INT32 Encry(char* pPlaintext,int nPlaintextLength,char *pCipherBuffer,int &nCipherBufferLength, char
*pKey,int nKeyLength);
    INT32 Decry(char* pCipher,int nCipherBufferLength,char *pPlaintextBuffer, int &nPlaintextBufferLength,
char *pKey,int nKeyLength);
};
```

其中 HandleData 用来执行一次完整的加密或解密操作，MakeData 用来实现 16 轮加密或解密迭代中的每一轮除去初始置换和逆初始置换的中间操作，MakeFirstKey 用来利用用户输入的初始密钥，来形成 16 个迭代用到的子密钥，MakeKey 用来形成 16 个密钥中的每一个子密钥，Encry 用于加密，Decry 用于解密。

3.2 DES 算法中用到的静态数组

初始置换 IP：

```
static const ULONG8    pc_first[64] = {/*初始置换 IP*/
                        58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
                        62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
                        57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
                        61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7 } ;
```

逆初始置换IP^{-1}：

```
static const ULONG8 pc_last[64] = { /*逆初始置换 IP-1*/
        40,8,48,16,56,24,64,32, 39,7,47,15,55,23,63,31,
        38,6,46,14,54,22,62,30, 37,5,45,13,53,21,61,29,
        36,4,44,12,52,20,60,28, 35,3,43,11,51,19,59,27,
        34,2,42,10,50,18,58,26, 33,1,41,9,49,17,57,25};
```

按位取值或赋值：

```
static const ULONG32    pc_by_bit[64] = { /*按位取值或赋值*/
                0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,
                0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,
```

```
            0x200000L, 0x100000L,    0x80000L, 0x40000L, 0x20000L,0x10000L,
            0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,
            0x100L, 0x80L,0x40L,0x20L, 0x10L, 0x8L, 0x4L, 0x2L, 0x1L,
            0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,
            0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,
            0x200000L, 0x100000L,    0x80000L, 0x40000L, 0x20000L, 0x10000L,
            0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,
            0x100L, 0x80L, 0x40L,0x20L, 0x10L, 0x8L,    0x4L, 0x2L, 0x1L,      };
```

置换运算 P：

```
static const ULONG8 des_P[32] = {/*置换运算 P*/
                                    16,7,20,21, 29,12,28,17, 1,15,23,26,
                                    5,18,31,10, 2,8,24,14, 32,27,3,9,
                                    19,13,30,6, 22,11,4,25};
```

选择扩展运算 E 盒：

```
static const ULONG8 des_E[48] = {/*数据扩展*/
                                        32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,
                                        12,13,14,15,16,17,16,17,18,19,20,21,
                                        20,21,22,23,24,25,24,25,26,27,28,29,
                                        28,29,30,31,32,1 };
```

选择压缩运算 S 盒：

```
static const ULONG8 des_S[8][64] = {/*数据压缩*/
        {
                0xe,0x0,0x4,0xf,0xd,0x7,0x1,0x4,0x2,0xe,0xf,0x2,0xb,
                0xd,0x8,0x1,0x3,0xa,0xa,0x6,0x6,0xc,0xc,0xb,0x5,0x9,
                0x9,0x5,0x0,0x3,0x7,0x8,0x4,0xf,0x1,0xc,0xe,0x8,0x8,
                0x2,0xd,0x4,0x6,0x9,0x2,0x1,0xb,0x7,0xf,0x5,0xc,0xb,
                0x9,0x3,0x7,0xe,0x3,0xa,0xa,0x0,0x5,0x6,0x0,0xd
        },

        {
                0xf,0x3,0x1,0xd,0x8,0x4,0xe,0x7,0x6,0xf,0xb,0x2,0x3,
                0x8,0x4,0xf,0x9,0xc,0x7,0x0,0x2,0x1,0xd,0xa,0xc,0x6,
                0x0,0x9,0x5,0xb,0xa,0x5,0x0,0xd,0xe,0x8,0x7,0xa,0xb,
                0x1,0xa,0x3,0x4,0xf,0xd,0x4,0x1,0x2,0x5,0xb,0x8,0x6,
                0xc,0x7,0x6,0xc,0x9,0x0,0x3,0x5,0x2,0xe,0xf,0x9
        },
        {
                0xa,0xd,0x0,0x7,0x9,0x0,0xe,0x9,0x6,0x3,0x3,0x4,0xf,
                0x6,0x5,0xa,0x1,0x2,0xd,0x8,0xc,0x5,0x7,0xe,0xb,0xc,
                0x4,0xb,0x2,0xf,0x8,0x1,0xd,0x1,0x6,0xa,0x4,0xd,0x9,
                0x0,0x8,0x6,0xf,0x9,0x3,0x8,0x0,0x7,0xb,0x4,0x1,0xf,
                0x2,0xe,0xc,0x3,0x5,0xb,0xa,0x5,0xe,0x2,0x7,0xc
        },
        {
```

```
                0x7,0xd,0xd,0x8,0xe,0xb,0x3,0x5,0x0,0x6,0x6,0xf,0x9,
                0x0,0xa,0x3,0x1,0x4,0x2,0x7,0x8,0x2,0x5,0xc,0xb,0x1,
                0xc,0xa,0x4,0xe,0xf,0x9,0xa,0x3,0x6,0xf,0x9,0x0,0x0,
                0x6,0xc,0xa,0xb,0xa,0x7,0xd,0xd,0x8,0xf,0x9,0x1,0x4,
                0x3,0x5,0xe,0xb,0x5,0xc,0x2,0x7,0x8,0x2,0x4,0xe
        },
        {
                0x2,0xe,0xc,0xb,0x4,0x2,0x1,0xc,0x7,0x4,0xa,0x7,0xb,
                0xd,0x6,0x1,0x8,0x5,0x5,0x0,0x3,0xf,0xf,0xa,0xd,0x3,
                0x0,0x9,0xe,0x8,0x9,0x6,0x4,0xb,0x2,0x8,0x1,0xc,0xb,
                0x7,0xa,0x1,0xd,0xe,0x7,0x2,0x8,0xd,0xf,0x6,0x9,0xf,
                0xc,0x0,0x5,0x9,0x6,0xa,0x3,0x4,0x0,0x5,0xe,0x3
        },
        {
                0xc,0xa,0x1,0xf,0xa,0x4,0xf,0x2,0x9,0x7,0x2,0xc,0x6,
                0x9,0x8,0x5,0x0,0x6,0xd,0x1,0x3,0xd,0x4,0xe,0xe,0x0,
                0x7,0xb,0x5,0x3,0xb,0x8,0x9,0x4,0xe,0x3,0xf,0x2,0x5,
                0xc,0x2,0x9,0x8,0x5,0xc,0xf,0x3,0xa,0x7,0xb,0x0,0xe,
                0x4,0x1,0xa,0x7,0x1,0x6,0xd,0x0,0xb,0x8,0x6,0xd
        },
        {
                0x4,0xd,0xb,0x0,0x2,0xb,0xe,0x7,0xf,0x4,0x0,0x9,0x8,
                0x1,0xd,0xa,0x3,0xe,0xc,0x3,0x9,0x5,0x7,0xc,0x5,0x2,
                0xa,0xf,0x6,0x8,0x1,0x6,0x1,0x6,0x4,0xb,0xb,0xd,0xd,
                0x8,0xc,0x1,0x3,0x4,0x7,0xa,0xe,0x7,0xa,0x9,0xf,0x5,
                0x6,0x0,0x8,0xf,0x0,0xe,0x5,0x2,0x9,0x3,0x2,0xc
        },
        {
                0xd,0x1,0x2,0xf,0x8,0xd,0x4,0x8,0x6,0xa,0xf,0x3,0xb,
                0x7,0x1,0x4,0xa,0xc,0x9,0x5,0x3,0x6,0xe,0xb,0x5,0x0,
                0x0,0xe,0xc,0x9,0x7,0x2,0x7,0x2,0xb,0x1,0x4,0xe,0x1,
                0x7,0x9,0x4,0xc,0xa,0xe,0x8,0x2,0xd,0x0,0xf,0x6,0xc,
                0xa,0x9,0xd,0x0,0xf,0x3,0x3,0x5,0x5,0x6,0x8,0xb
        } };
```

等分密钥，密钥循环左移及密钥选取：

```
static const ULONG8 keyleft[28] = {/*等分密钥*/
        57,49,41,33,25,17,9,1,58,50,42,34,26,18,
        10,2,59,51,43,35,27,19,11,3,60,52,44,36};

static const ULONG8 keyright[28] = {/*等分密钥*/
        63,55,47,39,31,23,15,7,62,54,46,38,30,22,
        14,6,61,53,45,37,29,21,13,5,28,20,12,4};

static const ULONG8 lefttable[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};/*密钥移位*/
static const ULONG8 keychoose[48] ={/*密钥选取*/
```

```
        14,17,11,24,1,5,3,28,15,6,21,10,
        23,19,12,4,26,8,16,7,27,20,13,2,
        41,52,31,37,47,55,30,40,51,45,33,48,
        44,49,39,56,34,53,46,42,50,36,29,32};
```

3.3 DES 密钥生成

DES 密钥是一个 64bit 的分组，但是其中 8bit 用于奇偶校验，所以密钥的有效位只有 56bit，由这 56bit 生成 16 轮子密钥。

首先将有效的 56bit 进行置换选择，将结果等分为 28bit 的两个部分，再根据所在的迭代轮数进行循环左移，左移后将两个部分合并为 56bit 的密钥，从中选取 48bit 作为此轮迭代的最终密钥，共生成 16 个 48bit 的密钥。每一个密钥，分为两个 24bit 的部分放在一个 ULONG32 的二维数组中保存。

每一轮密钥的生成，由 MakeKey 函数实现：

```
INT32 MakeKey(    ULONG32 *keyleft,ULONG32 *keyright ,ULONG32 number)
{
    ULONG32 tmpkey[2] ={0};
    ULONG32 *Ptmpkey = (ULONG32*)tmpkey;
    ULONG32 *Poutkey = (ULONG32*)&m_arrOutKey[number];
    INT32 j;
    memset((ULONG8*)tmpkey,0,sizeof(tmpkey));
    /*要最高的一位或两位*/
    *Ptmpkey = *keyleft&wz_leftandtab[lefttable[number]] ;
    Ptmpkey[1] = *keyright&wz_leftandtab[lefttable[number]] ;
    if ( lefttable[number] == 1)
    {
        *Ptmpkey >>= 27;
        Ptmpkey[1] >>= 27;
    }
    else
    {
        *Ptmpkey >>= 26;
        Ptmpkey[1] >>= 26;
    }
    Ptmpkey[0] &= 0xfffffff0;
    Ptmpkey[1] &= 0xfffffff0;
    /*得到高位的值*/
    *keyleft <<= lefttable[number] ;
    *keyright <<= lefttable[number] ;
    *keyleft |= Ptmpkey[0] ;
    *keyright |= Ptmpkey[1] ;
```

```
    Ptmpkey[0] = 0;
    Ptmpkey[1] = 0;

    /*从 56 位中选出 48 位,3 个 16 位*/
    for ( j = 0 ; j < 48 ; j++)
    {
        if ( j < 24 )
        {

            if ( *keyleft&pc_by_bit[keychoose[j]-1])
            {
                Poutkey[0] |= pc_by_bit[j] ;
            }
        }

        else /*j>=24*/
        {
            if ( *keyright&pc_by_bit[(keychoose[j]-28)])
            {
                Poutkey[1] |= pc_by_bit[j-24] ;
            }
        }
    }
    return SUCCESS;
}
```

3.4 DES 加密运算

DES 的加密运算也分为 16bit 圈迭代。

首先将明文分为 64bit 的数据块，不够 64bit 的用 0 补齐。在每一轮中，对每一个 64bit 的数据块，首先进行初始换位，并将数据块分为 32bit 的两部分：

```
INT32    number = 0 ,j = 0;
ULONG32 *right = &left[1] ;
ULONG32 tmp = 0;
ULONG32 tmpbuf[2] = { 0 };

/*第一次调整 pc_first[64]*/
for ( j = 0 ; j < 64 ; j++)
{
    if (j < 32 )
    {
        if ( pc_first[j] > 32)/*属于 right*/
        {
```

```
                if ( *right&pc_by_bit[pc_first[j]-1] )
                {
                    tmpbuf[0] |= pc_by_bit[j] ;
                }
            }
            else
            {
                if ( *left&pc_by_bit[pc_first[j]-1] )
                {
                    tmpbuf[0] |= pc_by_bit[j] ;
                }
            }
        }
        else
        {
            if ( pc_first[j] > 32)/*属于 right*/
            {
                if ( *right&pc_by_bit[pc_first[j]-1] )
                {
                    tmpbuf[1] |= pc_by_bit[j] ;
                }
            }
            else
            {
                if ( *left&pc_by_bit[pc_first[j]-1] )
                {
                    tmpbuf[1] |= pc_by_bit[j] ;
                }
            }
        }
    }
    *left    = tmpbuf[0];
    *right = tmpbuf[1];
```

经过初始置换并且分组之后，将进行 DES 加密算法的核心部分。

首先，保持左部不变，将右部由 32bit 扩展成为 48bit，分别存在两个 ULONG32 类型的变量里，每个占 24bit：

```
/*从 56 位中选出 48 位,3 个 16 位*/
for ( j = 0 ; j < 48 ; j++)
{
    if ( j < 24 )
    {

        if ( *keyleft&pc_by_bit[keychoose[j]-1])
        {
```

```
                Poutkey[0] |= pc_by_bit[j] ;
            }
        }

        else /*j>=24*/
        {
            if ( *keyright&pc_by_bit[(keychoose[j]-28)])
            {
                Poutkey[1] |= pc_by_bit[j-24] ;
            }
        }
}
```

在将右部扩展为 48bit 之后，与该轮的密钥进行异或操作，由于 48bit 分在一个 ULONG32 数组中的两个元素中，因此要进行两次异或操作：

```
for ( j = 0 ; j < 2 ; j++)
{
    exdes_P[j] ^= m_arrOutKey[number][j] ;
}
```

在异或操作完成之后，对新的 48bit 进行压缩操作，即 S 盒。

将其每取 6bit，进行一次操作：

```
/*由 48—>32*/
exdes_P[1] >>= 8 ;
rexpbuf[7] = (ULONG8) (exdes_P[1]&0x0000003fL) ;
exdes_P[1] >>= 6 ;
rexpbuf[6] = (ULONG8) (exdes_P[1]&0x0000003fL) ;
exdes_P[1] >>= 6 ;
rexpbuf[5] = (ULONG8) (exdes_P[1]&0x0000003fL) ;
exdes_P[1] >>= 6 ;
rexpbuf[4] = (ULONG8) (exdes_P[1]&0x0000003fL) ;
exdes_P[0]   >>=   8 ;
rexpbuf[3] = (ULONG8) (exdes_P[0]&0x0000003fL) ;
exdes_P[0] >>= 6 ;
rexpbuf[2] = (ULONG8) (exdes_P[0]&0x0000003fL) ;
exdes_P[0] >>= 6 ;
rexpbuf[1] = (ULONG8) (exdes_P[0]&0x0000003fL) ;
exdes_P[0] >>= 6 ;
rexpbuf[0] = (ULONG8) (exdes_P[0]&0x0000003fL) ;
exdes_P[0] = 0 ;
exdes_P[1] = 0 ;
```

8 个 6bit 的数据存在 ULONG rexpbuf[8]中，然后进行数据压缩操作，每一个 6bit 经过运算之后输出 4bit，因此最终输出的是压缩后的 32bit 数据：

```
/*由 48—> 32*/
```

```
*right = 0 ;
for ( j = 0 ; j < 7 ; j++)
{
    *right |= des_S[j][rexpbuf[j]] ;
    *right <<= 4 ;
}
*right |= des_S[j][rexpbuf[j]] ;
```

对新的 32bit 数据，进行一次置换操作：

```
/*又要换位了*/
datatmp = 0;
for ( j = 0 ; j < 32 ; j++)
{
    if ( *right&pc_by_bit[des_P[j]-1] )
    {
        datatmp |= pc_by_bit[j] ;
    }
}
*right = datatmp ;
```

再把左右部分进行异或作为右半部分，最原始的右边作为左半部分：

```
/*一轮结束收尾操作*/
*right ^= *left;
*left = oldright;
```

最后进行逆初始置换，完成一轮完整的加密操作：

```
/*最后一次调整 pc_last[64]*/
for ( j = 0 ; j < 64 ; j++)
{
    if (j < 32 )
    {
        if ( pc_last[j] > 32)/*属于 right*/
        {
            if ( *right&pc_by_bit[pc_last[j]-1] )
            {
                tmpbuf[0] |= pc_by_bit[j] ;
            }
        }
        else
        {
            if ( *left&pc_by_bit[pc_last[j]-1] )
            {
                tmpbuf[0] |= pc_by_bit[j] ;
            }
        }
    }
    else
```

```
    {
        if ( pc_last[j] > 32)/*属于 right*/
        {
            if ( *right&pc_by_bit[pc_last[j]-1] )
            {
                tmpbuf[1] |= pc_by_bit[j] ;
            }
        }
        else
        {
            if ( *left&pc_by_bit[pc_last[j]-1] )
            {
                tmpbuf[1] |= pc_by_bit[j] ;
            }
        }
    }
}

*left =    tmpbuf[0] ;
*right = tmpbuf[1];
```

3.5 封装 DES 加密函数

将上述运算整合在一起，可以封装成一个加密函数，以便于调用，其中 pPlaintext 为明文部分，nPlaintextLength 为明文长度，pCipherBuffer 为准备存放密文的缓冲区，nCipherBufferLength 为密文长度，pKey 为密钥，nKeyLength 为密钥长度：

```
INT32 Encry(char* pPlaintext,int nPlaintextLength,char *pCipherBuffer,int &nCipherBufferLength, char *pKey,int nKeyLength)
```

由于加密、解密均要以 32bit 为单位进行操作，故需要计算相关参数，以确定加密的循环次数以及密文缓冲区是否够用，确定后将需要加密的明文格式化到新分配的缓冲区内：

```
int nLenthofLong = ((nPlaintextLength+7)/8)*2;

if(nCipherBufferLength<nLenthofLong*4)
{//out put buffer is not enough
    nCipherBufferLength=nLenthofLong*4;
    return 0;
}
memset(pCipherBuffer,0,nCipherBufferLength);
ULONG32 *pOutPutSpace = (ULONG32 *)pCipherBuffer;

ULONG32 *pSource;
if(nPlaintextLength != sizeof(ULONG32)*nLenthofLong)
```

```
{
    pSource = new ULONG32[nLenthofLong];
    memset(pSource,0,sizeof(ULONG32)*nLenthofLong);
    memcpy(pSource,pPlaintext,nPlaintextLength);
}
else
{
    pSource = (ULONG32 *)pPlaintext;
}
```

开始对明文进行加密，加密后将之前分配的缓冲区从内存中删除：

```
ULONG32 gp_msg[2] = {0,0};
for (int i=0;i<(nLenthofLong/2);i++)
{
    gp_msg[0] = pSource[2*i];
    gp_msg[1] =pSource[2*i+1];
    HandleData(gp_msg,DESENCRY);
    pOutPutSpace[2*i] = gp_msg[0];
    pOutPutSpace[2*i+1] = gp_msg[1];
}
if(pPlaintext!=(char *)pSource)
{
    delete []pSource;
}
```

最后需要说明，上述函数为一次完整的加密流程，解密流程与加密流程基本一致，不同的的地方在于所生成的 16 个密钥的使用顺序，加密运算与解密运算的使用顺序正好想法。

第四章 基于 TCP 的聊天功能模块设计

4.1 建立连接

对于客户端，首先输入服务器 IP 地址，建立并初始化连接套接字和 sockaddr_in 结构体，向服务器请求连接，进行实时聊天，关闭套接字：

```
char strIpAddr[16];
printf("Please input the server address:\r\n");
cin>>strIpAddr;

int nConnectSocket, nLength;
struct sockaddr_in sDestAddr;
if ((nConnectSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket");
```

```
    exit(errno);
}

bzero(&sDestAddr, sizeof(sDestAddr));
sDestAddr.sin_family = AF_INET;
sDestAddr.sin_port = htons(SERVERPORT);
sDestAddr.sin_addr.s_addr = inet_addr(strIpAddr);
  /* 连接服务器 */
if (connect(nConnectSocket, (struct sockaddr *) &sDestAddr, sizeof(sDestAddr)) != 0)
{
    perror("Connect ");
    exit(errno);
}
else
{
    printf("Connect Success!    \nBegin to chat...\n");
    SecretChat(nConnectSocket,strIpAddr,"benbenmi");
}
close(nConnectSocket);
```

对于服务器端，建立并初始化本地 sockaddr_in 结构体，与本地套接字绑定并开始监听，建立远程 sockaddr_in 和套接字，在接受客户端连接请求后存储客户端的相关信息：

```
int nListenSocket, nAcceptSocket;
socklen_t nLength = 0;
struct sockaddr_in sLocalAddr, sRemoteAddr;
if ((nListenSocket = socket(PF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket");
    exit(1);
}

bzero(&sLocalAddr, sizeof(sLocalAddr));
sLocalAddr.sin_family = PF_INET;
sLocalAddr.sin_port = htons(SERVERPORT);
sLocalAddr.sin_addr.s_addr = INADDR_ANY;

if (bind(nListenSocket, (struct sockaddr *) &sLocalAddr, sizeof(struct sockaddr))== -1)
{
    perror("bind");
    exit(1);
}
if (listen(nListenSocket, 5) == -1)
{
    perror("listen");
    exit(1);
```

```
}
printf("Listening...\n");
nLength = sizeof(struct sockaddr);
if ((nAcceptSocket = accept(nListenSocket, (struct sockaddr *) &sRemoteAddr,&nLength)) == -1)
{
	perror("accept");
	exit(errno);
}
else
{
	close(nListenSocket);
	printf("server: got connection from %s, port %d,
socket %d\n",inet_ntoa(sRemoteAddr.sin_addr),ntohs(sRemoteAddr.sin_port), nAcceptSocket);
	SecretChat(nAcceptSocket,inet_ntoa(sRemoteAddr.sin_addr),"benbenmi");
	close(nAcceptSocket);

}
```

4.2 多进程全双工聊天程序分析

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，并发地运行许多进程。

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地运行。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。fork()函数向子进程返回 0，向父进程返回子进程的进程标识符，该标识符是一个非零的 int 值。

当 fork()函数被执行后，一个完全独立的子进程已经创建完毕并开始运行，在代码中可以利用该函数的返回值来区分父进程和子进程。另外，每个进程都有自己独立的堆栈段，所以两个进程的局部变量相互独立，在任意一个进程中都可以随便访问而不必考虑同步问题，但是如果进程使用了文件指针，则必须小心对待，因为两个进程的文件指针将会指向同一个低层文件，并行的读写操作可能会造成冲突。

另外，如果调用 fork()函数的次数过于频繁造成系统中进程总数过多，系统可能由于耗尽所有可用的资源而导致创建新进程失败。

该聊天功能在函数 SecretChat()中实现：

```
void SecretChat(int nSock,char *pRemoteName, char *pKey)
{
```

```
CDesOperate cDes;
if(strlen(pKey)!=8)
{
	printf("Key length error");
	return ;
}

pid_t nPid;
nPid = fork();
if(nPid != 0)
{
	while(1)
	{
		bzero(&strSocketBuffer, BUFFERSIZE);
		int nLength = 0;
		nLength = recv(nSock, strSocketBuffer,BUFFERSIZE,0);
		if(nLength !=BUFFERSIZE)
		{
			break;
		}
		else
		{
			int nLen = BUFFERSIZE;
			cDes.Decry(strSocketBuffer,BUFFERSIZE,strDecryBuffer,nLen,pKey,8);
			strDecryBuffer[BUFFERSIZE-1]=0;
			if(strDecryBuffer[0]!=0&&strDecryBuffer[0]!='\n')
			{
				printf("Receive message form <%s>: %s\n", pRemoteName,strDecryBuffer);
				if(0==memcmp("quit",strDecryBuffer,4))
				{
					printf("Quit!\n");
					break;
				}
			}
		}
	}
}
else
{
	while(1)
	{
		bzero(&strStdinBuffer, BUFFERSIZE);
		while(strStdinBuffer[0]==0)
		{
			if (fgets(strStdinBuffer, BUFFERSIZE, stdin) == NULL)
```

```
                {
                    continue;
                }
            }
            int nLen = BUFFERSIZE;
            cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pKey,8);
            if(send(nSock, strEncryBuffer, BUFFERSIZE,0)!=BUFFERSIZE)
            {
                perror("send");
            }
            else
            {
                if(0==memcmp("quit",strStdinBuffer,4))
                {
                    printf("Quit!\n");
                    break;
                }
            }
        }
    }

}
```