第3章　基于 DES 加密的 TCP 聊天程序

3.1 本章训练目的与要求

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，向应用层用户进程提供可靠的、全双工的数据流传输。

本章训练的目的如下。

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。

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。

3.2 相关背景知识

3.2.1 DES 算法的历史

上个世纪六七十年代，随着计算机在通信网络中的应用，对信息处理设备标准化的要求也越来越迫切，加密产品作为信息安全的核心，自然也有标准化需求。

1973 年，NBS 发布了公开征集标准密码算法的请求，并确定了一系列的设计准则如下。

(1) 算法应具有较高的安全性；

(2) 算法必须是完全确定，没有含糊之处；

(3) 算法的安全性必须完全依赖于密钥；

(4) 对于任何用户必须是不加区分的；

(5) 用于实现算法的电子器件必须经济实惠。

1974 年，IBM（美国商用电器公司）向 NBS 提交了由 Tuchman 博士领导的小组设计并经改造的 Lucifer 算法。NSA（美国国家安全局）组织专家对该算法进行了鉴定，使其成为 DES 的基础。

1975 年 NBS 公布了这个算法，并说明要以它作为联邦信息加密标准，征求各方意见。

1976 年，DES 被采纳作为联邦标准，并授权在非机密的政府通信中使用。DES 在银行、金融界崭露头角，随后得到广泛应用。

DES 系统于 1977 年被正式批准，并于同年 7 月 15 日宣布生效，长期来一直被美国政府、军队广泛使用。1988 年里根政府宣布 DES 算法服役期满（一般指十年）而转为民用，后来又被美国商界和世界其他国家采用。

1981 年 ANSI（美国国家标准技术研究所）将其作为数据加密标准，称之为 DEA（ANSI

X3.92）。1983 年 ISO（国际标准化组织）采用其作为标准，称为 DEA-1。

1984 年美国总统签署 145 号国家安全决策令（NSDD），命令 NSA 着手发展新的加密标准，用于政府系统非机密数据和私人企事业单位。NSA 宣布每隔 5 年对 DES 重新审议一次，以坚定其是否还适合继续作为联邦标准。1994 年 1 月，宣布要延续到 1998 年。

1990 年，在 Eurocrypt（欧洲密码年会）上，以色列密码学家 Shamir 提出了针对 DES 的“差分分析法”。1993 年，仍然是在 Eurocrypt 上，Matsui 提出“线性分析法”。

1997 年，ANSI 开始征集 AES（高级加密标准）。2000 年，选定比利时人 Joan Daemen 和 Vincent Rijmen 设计的 Rijndael 算法作为新的标准。

虽然 DES 已不再作为数据加密标准，但它仍然值得研究和学习。首先三重 DES 算法仍在 Internet 中广泛使用，如 PGP 和 S/MIME 中都使用了三重 DES 作为加密算法。其次 DES 是历史上最为成功的一种分组密码算法，它的使用时间之久，范围之大，是其它分组密码算法不能企及的，而 DES 的成功则归功于其精巧的设计和结构，学习 DES 算法的细节有助于读者深入了解分组密码的设计方法，理解如何通过算法实现混乱和扩散准则，从而快速的掌握分组密码的本质问题。

3.2.2 DES 算法细节

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。

1. 初始置换 IP

将 64 bit 的明文重新排列，而后分成左右两块，每块 32bit，用 L_0 和 R_0 表示。IP 置换表如表 2-1 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，各列经过偶采样和奇采样置换后，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 2-1 IP 置换表

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

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 2-2 所示。

表 2-2 逆初始置换表

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

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 码字划分，并将原来明文的校验位 $p_8,p_{16},\ldots,p_{64}$ 变为 IP 输出的一个字节。

3. 16 轮迭代

16 轮迭代是 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，作为下一轮迭代时右边的段，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 和置换 P，这些变换合称 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 2-3 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 2-3 选择扩展运算变换表

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

(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 2-4 至 2-11 所示，每张表的输入为 6bit，输出为 4bit。在查表之前，将密钥加运算的输出作为 48bit 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 2-4 选择压缩运算变换表

	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 2-5 选择压缩运算变换表

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

表 2-6 选择压缩运算变换表

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

57-64	0x5	0xb	0xa	0x5	0xe	0x2	0x7	0xc

表 2-7　选择压缩运算变换表

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

表 2-8　选择压缩运算变换表

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

表 2-9　选择压缩运算变换表

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

表 2-10　选择压缩运算变换表

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

33-40	0x1	0x6	0x4	0xb	0xb	0xd	0xd	0x8
41-48	0xc	0x1	0x3	0x4	0x7	0xa	0xe	0x7
49-56	0xa	0x9	0xf	0x5	0x6	0x0	0x8	0xf
57-64	0x0	0xe	0x5	0x2	0x9	0x3	0x2	0xc

表 2-11　选择压缩运算变换表

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

S 盒算法流程如下。假设输入的 48bit 为 $R_1R_2R_3\ldots R_{47}R_{48}$，需要将其转换为 32 位值，先把输入值视为由 8 个 6 位二进制块组成，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，故其十进制值范围为 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 2-4，b 对应表 2-5，c 对应表 2-6，d 对应表 2-7，e 对应表 2-8，f 对应表 2-9，g 对应表 2-10，h 对应表 2-11。

例如:

a.　a = 31，则到表 2-4 中找到 32 的位置，把对应的结果 0x8 赋给 a。

b.　d = 52，则到表 2-7 中找到 53 的位置, 把对应的结果 0x3 赋给 d。

c.　g = 15，则到表 2-10 中找到 16 的位置, 把对应的结果 0xa 赋给 g。

这里需要说明，有些资料在数据压缩这一步所采用的方法与本章不同，但殊途同归，最终的结果是相同的。

(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 2-12 进行换位。将数据 $R_1R_2R_3\ldots R_{31}R_{32}$ 转换成为 $R_{16}R_7R_{20}\ldots R_4R_{25}$。

表 2-12　置换运算变换表

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。

加密过程的流程如图 2-1 所示。

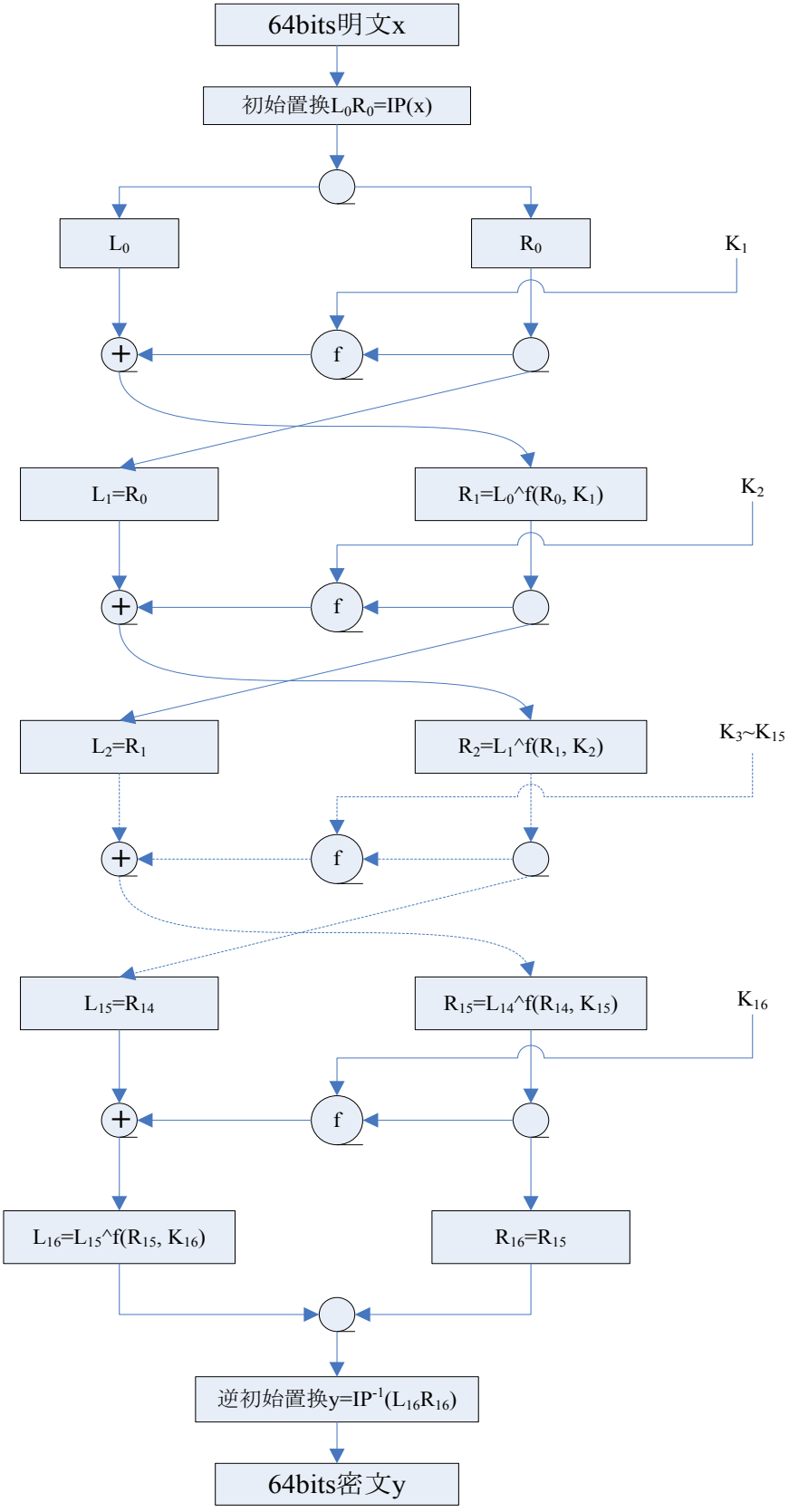

图 2-1　DES 加密流程图

其中，f(R,K)=P(S(E(R)^K))，R 为某一轮迭代的右边段，K 为该轮密钥，E 为选择扩展运算，S 为选择压缩运算，P 为置换运算。

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 轮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。下面以第 N 轮子密钥的产生为例进行说明。

(1) 置换选择 PC-1

置换选择 PC-1 只在第一轮子密钥的产生过程中需要使用，它的目的是从 64bit 初始密钥中选出 56bit 有效位。选择的过程是一个查表的过程，如表 2-13 与 2-14 所示，输出的 56bit 被分为两组，每组 28bit，分别进入 C 寄存器（查表 2-13 的结果）和 D 寄存器（查表 2-14 的结果）中，准备进行循环左移。

表 2-13 密钥置换选择 PC-1

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	50	44	36

表 2-14 密钥置换选择 PC-1

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

(2) 循环左移 LS：

此轮密钥的产生所需要循环左移的位数即为表 2-15 中的第 N 个元素（首元素为第一个）。C，D 寄存器中的 28bit 经过循环左移后，拼接为 56bit 作为此轮置换选择 PC-2 的输入，同时也作为第 N+1 轮子密钥循环左移的输入。

表 2-15 密钥循环左移位数表

1	1	2	2	2	2	2	2
1	2	2	2	2	2	2	1

(3) 置换选择 PC-2

置换选择 PC-2 将输入的 56bit 中的第 9、18、22、25、35、38、43、54 位删去，将其余位置按照表 2-16 置换位置，输出 48bit，作为第 N 轮的子密钥。

表 2-16 密钥置换选择 PC-2

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

生成 16 轮子密钥的流程如图 2-2 所示。

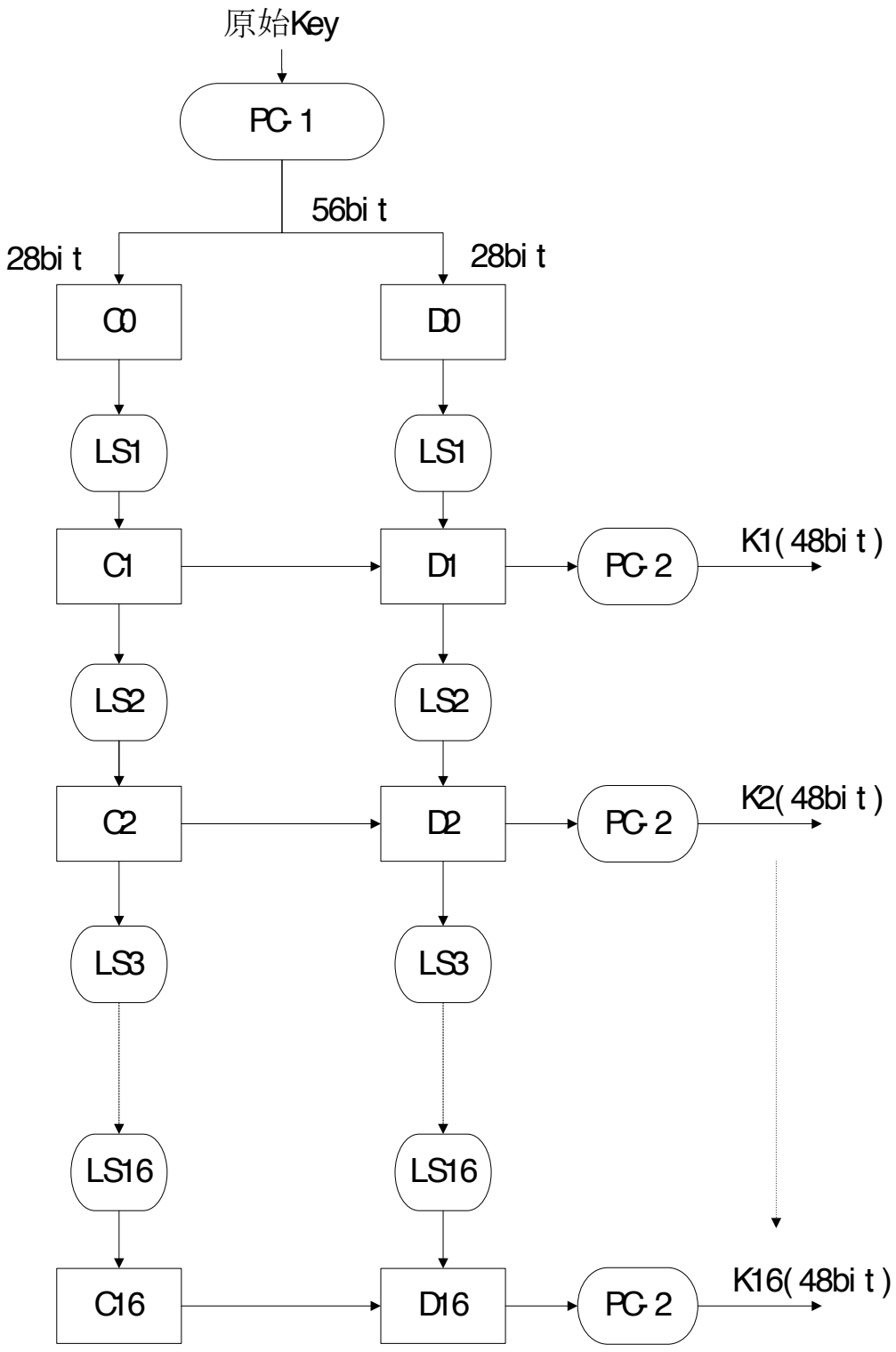

图 2-2　DES 子密钥生成流程图

3.2.3 TCP 协议

TCP 代表的含义是传输控制协议(Transmission Control Protocol)，它是一种端对端的协议。要用 TCP 协议与另一台计算机通信，两台机之间必须连接在一起，每一端都为通话做好准备。TCP 协议主要用来处理数据流，可以正确处理乱序的数据包。TCP 协议甚至还允许存

在丢失的或者损坏的数据包，最终它可以再次得到这些数据包。TCP 协议以它自己的方式缓存数据，不过，其缓存过程对程序员和用户是透明的。

TCP 协议每发送一个数据包将会收到一个确认信息。这种发送/应答模式可以使对方知道你是否收到了数据，从而保证了可靠性，当然，这也造成一些性能损失。为了改善系统效率不高的状况，TCP 引入了“捎带确认(piggybacking ACKs)”的方法。TCP 协议之所以是全双工的就是因为“捎带确认”信息，它允许双方同时发送数据，这是通过在当前的数据包中携带以前收到的数据的确认信息的方式实现的。从提高网络利用率的角度看，这比单纯发送一个通知对方“信息已收到”的数据包要好得多。同时，这种确认是批量确认的概念，即一次确认一个以上的数据包，表示“我收到了包括这个数据包在内的全部数据包”。

一个 TCP 段就是一个单个的 TCP 数据包。TCP 是一个数据流，因此，最关键的就是“连接”。最大报文段长度(MSS)是在连接的时候协商的，但是，它总是在不断地改变。默认的最大报文段长度是 536 字节，这是 576 字节(IP 协议保证的最小数据包长度)减去用于 IP 头的 20 个字节和用于 TCP 头的 20 个字节以后的长度。TCP 协议要设法避免在 IP 级别上的分段。因此，TCP 协议总是从 536 字节开始的。

一个 TCP 数据包的头是 20 个字节，如果使用一些选项，TCP 数据包头都可以放大。TCP 头不包含 IP 地址，它仅需要知道要连接哪一个端口。TCP 工作时要一直跟踪状态表中的端对端的连接，这个状态表包含 IP 地址和端口，也就是说， TCP 头不需要 IP 信息，因为它来自于 IP 头。

TCP 头部（如图 2-3 所示）各字段依次如下。

(1) 源端口，16 位:用于这次连接的本地 TCP 端口。

(2) 目的地端口，16 位:通讯目标机器的 TCP 端口。

(3) 序列号，32 位:用来跟踪数据包顺序的号码。

(4) 确认编号，32 位:确认的以前收到的序列号。

(5) 头长度，4 位:报头中的 32 位字(words)的数量。如果不使用选项，这个值设定为 5。

(6) 保留，6 位:为将来的使用保留的字节。

(7) 标记，一共 6 位: 每一个标记一个字节(开或者关)，其具体含义如下。

a. URG：紧急字段指针。如果为 1，表示数据包中包含紧急数据。此时 TCP 协议包头结构中的紧急指针有效。

b. ACK：确认标志位。如果为 1，表示包中的确认号是有效的。否则，包中的确认号是无效的。

c. PSH：推送功能。如果为 1，表示接收端应尽快将数据传送给应用层。

d. RST：重置位，用来复位一个连接。RST 标志置位的数据包称为复位包。一般情况下，如果 TCP 收到的一个分段明显不是属于该主机上的任何一个连接，则向远端发送一个复位包。

e. SYN: 用来建立连接，让连接双方同步序列号。如果 SYN=1 而 ACK=0，则表示数据包为连接请求；如果 SYN=1 而 ACK=1，则表示接受连接。

f. FIN：表示发送端已经没有数据要求传输了，请求释放连接。

(8) 窗口尺寸，16 位:从接收方将收到的确认字段开始。

(9) 校验和，16 位:TCP 头和数据的校验和。

(10) 应急指针，16 位:指向跟在 URG 数据后面的数据的序列号的偏移值。

(11) 选项：MSS、窗口比例等等。

(12) TCP 连接的两端使用两对 IP 地址和端口识别这个连接，并且向监听这个端口的应用程序发送数据。

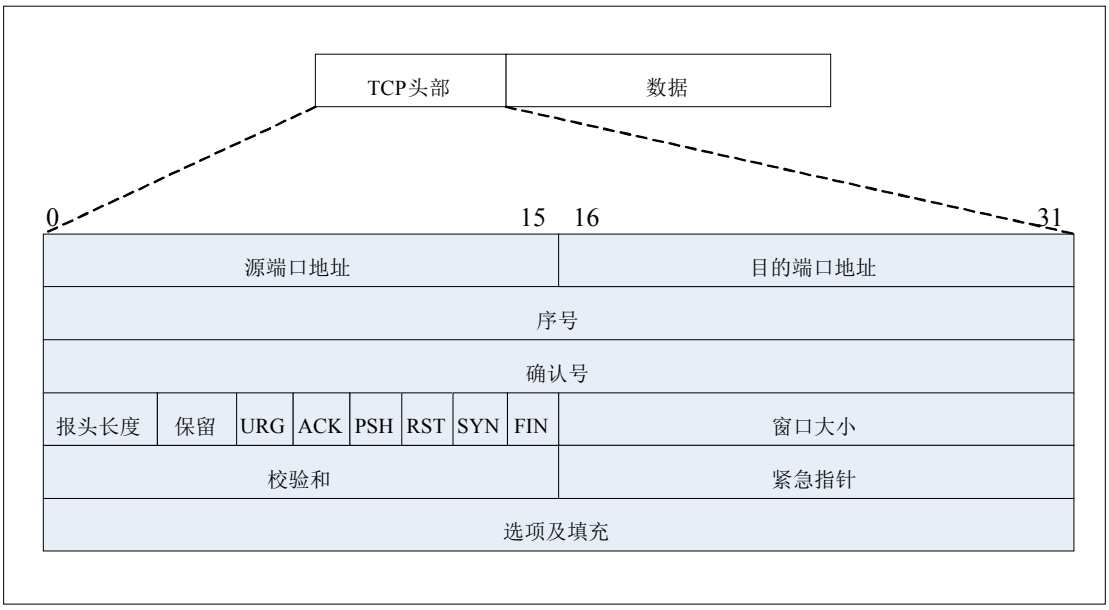

图 2-3　TCP 数据包格式

3.2.4 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 Socket 函数调用去传输数据以符合指定的协议。套接字存在于通信区域中，一个套接字只能与同一区域内的套接字交换数据。TCP/IP 提供了三种类型的套接字：

(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对数据报作较底层的控制。

在本章的练习中使用的就是流式套接字。

3.2.5 TCP 通信相关函数介绍

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。

下面介绍本章用到的几个 Linux 下的常用套接字函数。

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。

参数说明：

(1) Domain：说明网络程序所在的主机采用的通讯协族(AF_UNIX 和 AF_INET 等)。AF_UNIX 只能够用于单一的 Unix 系统进程间通信，而 AF_INET 是针对 Internet 的，因而可以允许在远程主机之间通信。

(2) Type：用于指定套接字的类型。根据 2.2.4 中给出的类型，本章程序将采用流式套接字。

(3) Protocol：用于指定套接字所使用的通信协议。正常情况下，对于给定的协议族，只有单一的协议支持特定的套接字类型，所以只要将 protocol 参数设置为 0 即可。

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相连。

参数说明：

(1) Sockfd：调用 socket 函数后返回的文件描述符。

(2) Addrlen：sockaddr 结构的长度。

(3) My_addr：一个指向 sockaddr 结构体的指针。

Sockaddr 结构体的定义如下。

```
struct sockaddr{
        unsigned short    sa_family;
        char              sa_data[14];
};
```

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];
}
```

这里主要使用 Internet 所以 sin_family 一般为 AF_INET，sin_addr 设置为 INADDR_ANY 表示可以和任何的主机通信，sin_port 是需要监听的端口号，sin_zero[8]是用来填充的。bind 将本地的端口同 socket 返回的文件描述符捆绑在一起。

此函数成功返回 0，失败返回-1，并设置 errno 变量。

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。

参数说明：

(1) Sockfd：经过 bind 操作的文件描述符。

(2) Backlog：设置请求队列的最大长度。

此函数成功返回 0，失败返回-1，并设置 errno 变量。

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连接请求。

参数说明：

(1) Sockfd：经过 listen 操作后的文件描述符。

(2) Addr：指向客户端结构体 sockaddr 的指针。

(3) Addrlen：addr 参数指向的内存空间的长度。

此函数调用时，服务器端的程序会一直阻塞到有一个客户程序发出了连接请求。

此函数成功时返回最后的服务器端的文件描述符，这个时候服务器端就可以向该描述符写信息了，失败时返回-1，并设置 errno 变量。

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。

参数说明：

(1) Sockfd：客户端 socket 返回的文件描述符。

(2) Serv_addr：存储服务器端的连接信息。

(3) Addrlen：serv_addr 的长度。

此函数成功返回 0，失败返回-1，并设置 errno 变量。

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 中的 nbytes 字节的内容写入文件描述符。

参数说明：

(1) Fd：socket 返回的文件描述符。

(2) Buf：指向要进行传输的内容的指针。

(3) Nbytes：即将传输的内容的大小。

此函数成功时返回写入的字节数，失败时返回-1，并设置 errno 变量。

7. read 函数

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

该函数用于从文件描述符 fd 中读取内容。

参数说明：同 write 函数。

此函数成功时返回读出的字节数，失败时返回-1，并设置 errno 变量。

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。

参数说明：

(1) S：要发送信息的文件描述符。

(2) Buf：指向要发送内容的指针。

(3) Len：为发送数据的长度。

(4) Flags：设为 0 时，功能与 write 函数相同。其他功能由于本章程序中不会用到，故在此不做详细介绍。

此函数成功时，返回时机发送的数据的字节数，失败时返回-1，并设置 errno 变量。

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。

参数说明：

(1) S：要读取内容的套接字文件描述符。

(2) Buf：指向要保存数据缓冲区的指针。

(3) Len：该缓存的最大长度。

(4) Flags：同 send 函数。

此函数成功时，返回 0，失败时返回-1，并设置 errno 变量。

10. close 函数

该函数用于关闭套接字，其调用形式为：close(sockfd);

3.3 实例编程练习

3.3.1 编程练习要求

在 Linux 平台下，实现基于 DES 加密的 TCP 通信，具体要求如下。

① 能够在了解 DES 算法原理的基础上，编程实现对字符串的 DES 加密解密操作。

② 能够在了解 TCP 和 Linux 平台下的 Socket 运行原理的基础上，编程实现简单的 TCP 通信，为简化编程细节，不要求实现一对多通讯。

③ 将上述两部分结合到一起，编程实现通信内容事先通过 DES 加密的 TCP 聊天程序，要求双方事先互通密钥，在发送方通过该密钥加密，然后由接收方解密，保证在网络上传输的信息的保密性。

下面给出示例程序的执行流程，仅供参考。

程序为命令行程序，命令行格式如下。

服务器：./chat　->输入 s

客户端：./chat　->输入 c

程序执行过程：（例如服务器 IP 地址为 192.168.1.91）

```
[root@localhost share]# ./chat
Client or Server?
s
Listening...
```

然后打开另一个控制台终端，运行客户端的可执行程序，客户端 IP 地址为 192.168.1.232

```
[root@localhost share]# ./chat
Client or Server?
c
Please input the server address:
192.168.1.91
Connect Success!
Begin to chat...
Hello~I'm client~
```

此时，客户端已经成功连接服务器，可以开始聊天，输入“Hello~I'm client~”，服务器端得出相应响应。

```
[root@localhost share]# ./chat
Client or Server?
s
Listening...
server: got connection from 192.168.1.232, port 53558, socket 4
Receive message form <192.168.1.232>: Hello~I'm client~
Hello~I'm server~
```

服务器端收到连接与信息之后，给客户端回复“Hello~I'm server~”，客户端也得到信息并响应。

```
[root@localhost share]# ./chat
Client or Server?
c
Please input the server address:
192.168.1.91
Connect Success!
Begin to chat...
Hello~I'm client~
Receive message form <192.168.1.91>: Hello~I'm server~
```

聊天过程中，任何一方输入“quit”，都可以关闭此连接，结束聊天。

3.3.2 编程训练设计与分析

程序分为两个部分，DES 算法部分和 TCP 通信部分。其中，DES 算法部分是核心部分，该部分用来实现对不大于一定长度要通过 TCP Socket 传输的信息进行加密和解密操作。

1. DES 加密解密设计实现分析

(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 轮迭代密钥，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，以及根据上述的四个函数封装的加密函数与解密函数，方便调用。

```
typedef int INT32;
class CDesOperate
{
private:
    ULONG32 m_arrOutKey[16][2];
    ULONG32 m_arrBufKey[2];
    INT32 HandleData(ULONG32 *left , ULONG8 choice);
    INT32 MakeData(ULONG32    *left ,ULONG32    *right ,ULONG32 number);
    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 用来对某一段字符加密，Decry 用来对某一段密文解密。

(2) DES 算法中用到的静态数组

初始置换 IP：

```
static ULONG8    pc_first[64] = {
                    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 ULONG8 pc_last[64] = {
        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 ULONG32    pc_by_bit[64] = {
            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 ULONG8 des_P[32] = {
                                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 ULONG8 des_E[48] = {
                                        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 ULONG8 des_S[8][64] =
{
        {
                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 ULONG8 keyleft[28] =
{
```

```
        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 ULONG8 keyright[28] =
{
        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 ULONG8 lefttable[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
static ULONG8 keychoose[48] ={
        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) DES 密钥生成

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

密钥生成，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 56 位的密钥，从中选取 48 位作为此轮迭代的最终密钥，共生成 16 个 48 位的密钥。每一个密钥，分为两个 24 位的部分放在一个 ULONG32 的二维数组中保存。

每一轮密钥生成，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*)&g_outkey[number];
    INT32 j;
    memset((ULONG8*)tmpkey,0,sizeof(tmpkey));
    *Ptmpkey = *keyleft&leftandtab[lefttable[number]] ;
    Ptmpkey[1] = *keyright&leftandtab[lefttable[number]] ;
    if ( lefttable[number] == 1)
    {
        *Ptmpkey >>= 27;
        Ptmpkey[1] >>= 27;
    }
    else
    {
        *Ptmpkey >>= 26;
        Ptmpkey[1] >>= 26;
    }
    Ptmpkey[0] &= 0xfffffff0;
    Ptmpkey[1] &= 0xfffffff0;
```

```
        *keyleft <<= lefttable[number] ;
        *keyright <<= lefttable[number] ;
        *keyleft |= Ptmpkey[0] ;
        *keyright |= Ptmpkey[1] ;
        Ptmpkey[0] = 0;
        Ptmpkey[1] = 0;
        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;
    }
```

(4) DES 加密运算

DES 的加密运算也分为 16 轮迭代。

首先将明文分为 64bit 的数据块，不够 64 位的用 0 补齐。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 };
        for ( j = 0 ; j < 64 ; j++)
        {
            if (j < 32 )
            {
                if ( pc_first[j] > 32)
                {
                    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)
                {
                    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 32 位扩展成为 48 位，分别存在两个 ULONG32 类型的变量里，每个占 24bit。

```
        for ( j = 0 ; j < 48 ; j++)
        {
            if ( j < 24 )
            {
                if ( *right&pc_by_bit[des_E[j]-1] )
                {
                    exdes_P[0] |= pc_by_bit[j] ;
                }
            }
            else
            {
                if ( *right&pc_by_bit[des_E[j]-1] )
```

```
                {
                    exdes_P[1] |= pc_by_bit[j-24] ;
                }
            }
        }
```

在将右部扩展成为 48 位之后，与该轮的密钥进行异或操作，由于 48 位分在一个 ULONG32 数组中的两个元素中，故要进行两次异或。

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

在异或操作完成之后，对新的 48 位进行压缩操作，即 S 盒。

将其每取 6 位，进行一次操作。

```
    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 6 位经过运算之后输出 4 位，故最终输出的是 32 位的压缩后的数据。

```
    *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。

```
    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，即将完成一轮完整的加密操作。

```
        *right ^= *left;
        *left = oldright;
```

最后进行逆初始置换，完成一轮完整的加密操作。

```
        for ( j = 0 ; j < 64 ; j++)
        {
            if (j < 32 )
            {
                if ( pc_last[j] > 32)
                {
                    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)
                {
                    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];
```

(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，代码如下。

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

首先检查初始密钥长度，若正确，则创建 16 轮迭代的密钥。

```
        if(nKeyLength != 8)
        {
            return 0;
        }
        MakeFirstKey((ULONG32 *)pKey);
```

由于加解密均要以 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;
        }

        return SUCCESS;
    }
```

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，仍为先进行初始置换，最后进行逆置换，中间 16 轮利用 16 个密钥的迭代加密，唯一不同的地方就是所生成的 16 个密钥的使用顺序，加密运算与解密运算的密钥使用顺序正好相反。

2. 基于 TCP 的聊天功能模块设计实现分析

TCP 通信流程如图 2-4 所示：

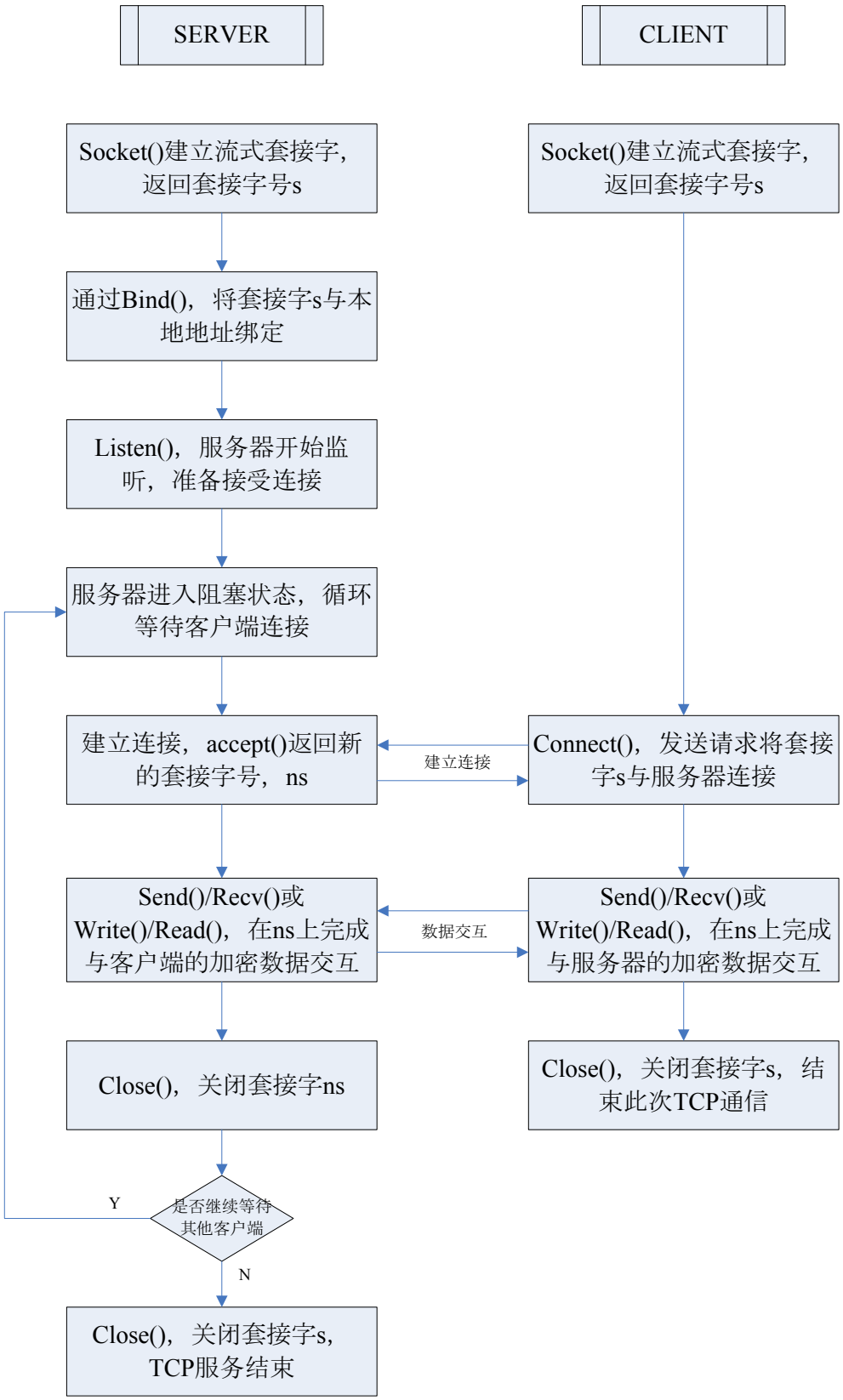

图 2-4 TCP 通信流程图

(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];
    cin>>strIpAddr;
    int nConnectSocket, nLength;
    struct sockaddr_in sDestAddr;
    if ((nConnectSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket");
        exit(errno);
    }
    sDestAddr.sin_family = AF_INET;
    sDestAddr.sin_port = htons(SERVERPORT);
    sDestAddr.sin_addr.s_addr = inet_addr(strIpAddr);
    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相关信息。

```
    int nListenSocket, nAcceptSocket;
    struct sockaddr_in sLocalAddr, sRemoteAddr;
    ……
    if (bind(nListenSocket, (struct sockaddr *) &sLocalAddr, sizeof(struct sockaddr))== -1)
    {
        perror("bind");
        exit(1);
    }
    if (listen(nListenSocket, 5) == -1)
    {
        perror("listen");
        exit(1);
    }
    nAcceptSocket = accept(nListenSocket,(struct sockaddr *) &sRemoteAddr,&nLength);
    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);
}
```

(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。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，每个进程都有自己的独立的堆栈段，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，就必须小心对待，因为两个进程的文件指针将会指向同一个底层文件，并行的读写操作可能造成冲突。

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，摘录代码如下。

```
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 = TotalRecv(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;
                    }
                }
            }
        }
```

该函数的参数有三个，其中 nSock 是 socket 句柄，要求其必须是一个已经建立连接的 socket；pRemoteName 指向一个字符串，代表远程主机的名字；pKey 指向另一个字符串，储存 DES 密钥。程序在连接建立完成后，直接调用该函数执行聊天功能，其中密钥为双方事先共享的字符串

该函数在完成必要的错误检查后，调用 fork()函数创建了一个子进程，如“if(nPid != 0)”满足，则代表当前进程为父进程，否则为子进程。父进程负责接收密文消息，解密并输出到屏幕；同时子进程负责从标准输入读取消息，加密并发送到指定套接字，两个进程完全并行，实现实时聊天的功能。由于父子进程的代码原理基本相同，故只对子进程代码进行分析。

如上文所示，聊天通讯双方传递固定大小的数据块，子进程通过调用 fgets()从标准输入读取数据，如果无法读到数据，则始终循环等待，直到读取到用户输入为止。由于本代码做教学只用，为了简化编写，未考虑用户输入超过缓冲区长度的现象。

在获得用户输入后，调用类 CDesOperate 中封装的加密函数 Encry()进行加密，然后调用 send()将加密后的数据块发送。

此外，需要解释的是，在父进程中从 socket 接收数据使用函数 TotalRecv()，这是因为 TCP 在某些特殊情况下（例如链路质量变化），recv()一次并不能返回对方发送的全部数据，需多次调用 recv()才能获得全部数据，使用该函数是为了确保将整个数据块可以被顺利接收。

该函数代码如下。

```
ssize_t TotalRecv(int s, void *buf, size_t len, int flags)
{
    size_t nCurSize = 0;
    while(nCurSize <len)
    {
        ssize_t nRes = recv(s,((char*)buf)+nCurSize,len-nCurSize,flags);
        if(nRes<0||nRes+nCurSize>len)
        {
            return -1;
        }
        nCurSize+=nRes;
    }
    return nCurSize;
}
```

3.4 扩展与提高

3.4.1 高级套接字函数

1. recv 函数和 send 函数

recv 函数和 send 函数提供了和 read 函数与 write 函数类似的功能，不同之处是前者提供了第四个参数来控制读写操作。两个函数的原型如下。

```
int recv(int sockfd,void *buf,int len,int flags)
int send(int sockfd,void *buf,int len,int flags)
```

该两函数的前三个参数和 read 函数,write 函数相同，第四个参数可以是 0 或者以下的组合：

(1) MSG_DONTROUTE: 不查找路由表；

(2) MSG_OOB：接受或者发送带外数据；

(3) MSG_PEEK：查看数据，并不从系统缓冲区移走数据；

(4) MSG_WAITALL：等待所有数据。

MSG_DONTROUTE： send 函数使用的标志。这个标志告诉 IP 协议，目的主机在本地网络上面，没有必要查找路由表。这个标志一般用在网络诊断和路由程序里面。

MSG_OOB：表示可以接收和发送带外数据。带外数据是相连的每一对流套接口间一个逻辑上独立的传输通道。带外数据是独立于普通数据传送给用户的，这一抽象要求带外数据设备必须支持每一时刻至少一个带外数据消息被可靠地传送。

MSG_PEEK：recv 函数的使用标志，表示只是从系统缓冲区中读取内容，而不清除系统缓冲区的内容。以便下次读的时候，仍然是相同的内容。一般在有多个进程读写数据时可以使用这个标志。

MSG_WAITALL：recv 函数的使用标志，表示等到所有的信息到达时才返回。使用这个标志的时候 recv 会一直阻塞，直到指定的条件满足，或者是发生了错误。

(1) 当读到了指定的字节时，函数正常返回，返回值等于 len ；

(2) 当读到了文件的结尾时，函数正常返回，返回值小于 len ；

(3) 当操作发生错误时，返回-1，且设置错误为相应的错误号(errno)。

如果 flags 为 0，则功能和 read 函数，write 函数完全相同。

2. shutdown 函数

shutdown 函数的原型如下。

```
int shutdown(int sockfd,int howto)
```

TCP 连接是双向的(是可读写的)，当使用 close 时，会把读写通道都关闭，有时侯希望只关闭一个方向，这个时候可以使用 shutdown 函数。针对不同的 howto 参数，系统会采取不同的关闭方式。

①howto=0 时，系统会关闭读通道，但是可以继续往套接字描述符写。

②howto=1 时，关闭写通道，和上面相反，这时候就只可以从套接字描述符里读了。

③howto=2 时，关闭读写通道，和 close 相同。在多进程程序里面，有几个子进程共享一个套接字时，如果使用 shutdown 函数，此时所有的子进程都不能够操作，这个时候只能使用 close 来关闭子进程的套接字描述符。

3.4.2 新一代对称加密协议 AES

1973年5月15日，美国国家标准局在联邦记录中公开征集密码体制，这一举措导致了数据加密标准(Data Encrypfion Standard，DES)算法的出现，并且一度成为世界上最广泛使用的密码体制。但后来由于DES算法及其变形的安全强度已经难以继续满足新的安全需要，难以对抗20世纪末出现的差分和线性密码分析，而且其实现速度，代码大小以及跨平台性均难以满足新的需求，于是美国政府于1997年开始公开征集新的数据加密标准(Advanced Encryption Standard，AES)算法，以取代DES。经过三轮筛选，最后选中比利时密码学家Joan Daemen和Vincent Rijmen提出的密码算法Rijndael作为AES正式取代DES。

1. AES 算法描述

Rijndael是具有可变分组长度和可变密钥长度的分组密码。其分组长度和密钥长度均可

以独立的设定为32bit的任意倍数，最小值为128bit，最大值为256bit，其输入输出均可看作一个一维的字符数组。假设输入明文为c_0，c_1，…，c_i…，其中0<i<4*N。将明文映射到一个字节矩阵上，称之为状态，在本例中N=4，密钥同理可以映射到密钥状态中，如表2-17所示。

表2-17　数据块长度为128bit的状态

c_0	c_4	c_8	c_{12}
c_1	c_5	c_9	c_{13}
c_2	c_6	c_{10}	c_{14}
c_3	c_7	c_{11}	c_{15}

Rijndael加密算法由3部分组成：一个初始轮密钥加法变换；Nr-1轮变换；最后一轮变换。下面给出实现该算法的伪代码：

```
Rijndael(State, CipherKey)
{
  //密钥扩展，即把输入的密钥扩展为加密用的密钥
  KeyExpansion(CipherKey, ExpandKey);
    //初始轮密钥加法变换
  AddRoundKey(State,ExpandKey);
    //N-1轮变换
  for (i=1;i<N;i++)
      Round(State, ExpandedKey[i])
      {
          //S-盒变换，非线性的砖匠变换
          ByteSub(State);
          //字节换位，将状态中的行按不同的偏移量进行循环移位
          ShiftRow(State);
          //作用在状态各列的砖匠变换
            MixColumn(State);
            //密钥加法
          AddRoundKey(State, ExpandedKey[i]);
      }
  //最后一轮变换
  FinalRound(State, ExpandedKey[N])
  {
      ByteSub(State);
      ShiftRow(State);
      AddRoundKey(State, ExpandedKey[i]);
  }
}
```

考虑到AddRoundKey()可以通过在每一列上执行一个额外的32位异或运算来实现，故也可以利用4KB的表通过查表操作实现，该实现方案对于每一轮的每一列仅仅需要4次查表和4次异或运算，实现这些操作的效率很高。同理，在解密算法中，也可以将轮变换的不同步骤合并为一组表的查询。

2. AES 算法与 DES 算法的比较

在一篇关于AES和DES两种算法比较的论文中，作者曾利用两种算法在相同环境下对同样长度的文件进行了加密，并且对其加密效率和时间进行了比较，程序的环境为Windows 2000，CPU 2.2GHz，内存256M，文件长度为3.77M（3960928字节），AES分组长度为128bit，密钥长度为128bit，DES分组长度为64bit，密钥长度为56bit，具体实验结果如下如表2-18与2-19所示。

表2-18 AES和DES加解密的时间比较

算法	加密(s)	解密(s)
DES	18	18
AES	6	10

表2-19 AES和DES加解密效率比较

算法	加密(Mb/s)	解密(Mb/s)
DES	1.676	1.676
AES	5.027	3.016

从上述实验数据可以看出，AES算法的分组长度与密钥长度均大于DES算法，而加解密效率也要高于DES算法，更为重要的是，要用穷举法破解AES在有限的时间内是不可能的。上述所有因素也促成了AES取代传统DES成为新一代加密体系。

3.4.3 DES 安全性分析

自 DES 算法 1977 年首次公诸于世以来，学术界对其进行了深入的研究，围绕它的安全性展开了激烈的争论。

1. DES 的安全性缺陷

在技术上对 DES 的批评主要集中在以下 3 个方面：

(1) 作为分组密码，DES 的加密单位仅有 64 位二进制，这对于数据传输来说太小，因为每个分组仅含有 8 个字符，而且其中某些位还要用于奇偶校验或其他通讯开销。

(2) DES 的密钥太短，有效密钥只有 56bit，而且各次迭代中使用的密钥是递推产生的，这种相关性必然降低密码体制的安全性，在现有技术下用穷举法寻找密钥已趋于可行。1999 年在电子前沿组织（Electronic Frontier Foundation，EFF）进行的一次测试中，只用了不到 3 天的时间就破解了一个 DES 加密系统。

(3) DES 不能对抗差分和线性密码分析。

2. 多重 DES 算法

针对 DES 算法上的缺陷，现在已发展出几十种改进的 DES，经过比较，大多学者认为多重 DES 具有较高的可行性。

为了增加密钥的长度，采用多重 DES 加密技术，将分组密码进行级联，在不同的密钥作用下，连续多次对一组明文进行加密。针对 DES 算法，最常用的是 3 重 DES 加密算法，它只用到两个 56 位 DES 密钥。假设这两个密钥为 K_1 与 K_2，该算法的步骤如下。

(1) 用密钥 K_1 进行 DES 加密;

(2) 用步骤 1 的结果使用密钥 K_2 进行 DES 解密;

(3) 用步骤 2 的结果使用密钥 K_1 进行 DES 加密。

三重 DES 可使加密密钥长度扩展到 128 位，其中有效 112 位。三重 DES 的 112 位密钥长度在可以预见的将来可认为是合适的、安全的，目前还没有找到针对此方案的攻击方法。

因为要破译它可能需要尝试 256 个不同的 56 位密钥直到找到正确的密钥。但是三重 DES 的时间是 DES 算法的 3 倍，时间开销较大。

3. 实现安全管理

现代密码学的特征是算法可以公开。保密的关键是如何保护好自己的密钥，而解密的关键则是如何能破解得到密钥。系统的安全管理者，要根据本系统实际所使用的密钥长度与其所保护的信息的敏感程度、重要程度以及系统实际所处安全环境的恶劣程度，在留有足够的安全系数的条件下来确定其密钥和证书更换周期的长短。同时，将已废弃的密钥和证书放入黑库归档，以备后用。密钥更换周期的正确安全策略是系统能够安全运行的保障，是系统的安全管理者最重要、最核心的日常工作任务。