第四章　基于 RSA 算法自动分配密钥的加密聊天程序

4.1　编程训练目的与要求

在讨论了传统的对称加密算法 DES 原理与实现技术的基础上，本章将以典型的非对称密码体系中 RSA 算法为例，以基于 TCP 协议的聊天程序加密为任务，系统地进行非对称密码体系 RSA 算法原理与应用编程技术的讨论和训练。通过练习达到以下的训练目的：

① 加深对 RSA 算法基本工作原理的理解。

② 掌握基于 RSA 算法的保密通信系统的基本设计方法。

③ 掌握在 Linux 操作系统实现 RSA 算法的基本编程方法。

④ 了解 Linux 操作系统异步 IO 接口的基本工作原理。

本章编程训练的要求如下。

① 要求在 Linux 操作系统中完成基于 RSA 算法的自动分配密钥加密聊天程序的编写。

② 应用程序保持第三章“基于 DES 加密的 TCP 通信”中示例程序的全部功能，并在此基础上进行扩展，实现密钥自动生成，并基于 RSA 算法进行密钥共享。

③ 要求程序实现全双工通信，并且加密过程对用户完全透明。

4.2　相关背景知识

4.2.1　公钥密码体系与 RSA 加密算法

1.　公钥密码体系的基本概念

传统对称密码体制要求通信双方使用相同的密钥，因此应用系统的安全性完全依赖于密钥的保密。针对对称密码体系的缺陷，Differ 和 Hellman 提出了新的密码体系—公钥密码体系，也称为非对称密码体系。在公钥加密系统中，加密和解密使用两把不同的密钥。加密的密钥（公钥）可以向公众公开，但是解密的密钥(私钥)必须是保密的，只有解密方知道。公钥密码体系要求算法要能够保证：任何企图获取私钥的人都无法从公钥中推算出来。

公钥密码体制中最著名算法是 RSA，以及背包密码、McEliece 密码、Diffe_Hellman、Rabin、零知识证明、椭圆曲线、EIGamal 算法等。

2.　公钥密码体系的特点

公钥密码体制如下部分组成：

(1) 明文：作为算法的输入的消息或者数据。

(2) 加密算法：加密算法对明文进行各种代换和变换。

(3) 密文：作为算法的输出，看起来完全随机而杂乱的数据，依赖明文和密钥。对于给定的消息，不同的密钥将产生不同的密文，密文是随机的数据流，并且其意义是无法理解的。

(4) 公钥和私钥：公钥和私钥成对出现，一个用来加密，另一个用来解密。

(5) 解密算法：该算法用来接收密文，解密还原出明文。

(6) 公钥密码体系的基本结构如图 4-1 所示。

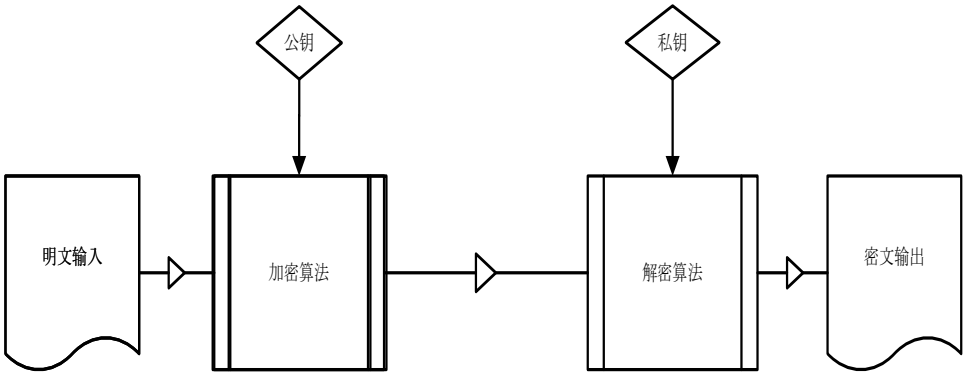

图 4-1 公钥密码体系原理示意图

3. RSA 加密算法的基本工作原理

RSA 加密算法是一种典型的公钥加密算法。RSA 算法的可靠性建立在分解大整数的困难性上。假如找到一种快速分解大整数算法的话，那么用 RSA 算法的安全性会极度下降。但是存在此类算法的可能性很小。目前只有使用短密钥进行加密的 RSA 加密结果才可能被穷举解破。只要其钥匙的长度足够长，用 RSA 加密的信息的安全性就可以保证。

RSA 密码体系使用了乘方运算。明文以分组为单位进行加密，每个分组的二进制值均小于 n，也就是说分组的大小必须小于或者等于 $Log_2 n$ ，在实际应用中，分组的大小是 k 位，则 $2^k<n<2^{k+1}$。

对于明文分组 M 和密文分组 C，加密和解密的过程如下。

（1） $\mathrm{C} = M^e \% \mathrm{n}$

（2） $M = C^d \% n = (M^e)^d \% n = M^{d \times e} \% n = \mathrm{M}$

其中 n、d、e 为三个整数，且 $\mathrm{d} \times \mathrm{e} \equiv 1 \% \Phi(\mathrm{n})$。收发双方共享 n，接受一方已知 d，发送一方已知 e，则此算法的公钥为{e，n}，私钥是{d，n}。

理解 RSA 基本工作原理需要数论的基础知识：

（1） 同余：两个整数 a，b，若它们除以整数 m 所得的余数相等，则称 a，b 对于模 m 同余，记作 $\mathrm{a} \equiv \mathrm{b} \% \mathrm{m}$。

（2） Euler 函数：Φ(n)是指所有小于 n 的正整数里，和 n 互质的整数的个数。其中 n 是一个正整数。假设整数 n 可以按照质因数分解写成如下形式：$n = P1^{a1} \times P2^{a2} \times \ldots \times Pm^{am}$；其中，$\mathrm{P1, P2 \ldots Pn}$为质数。则 $\Phi(\mathrm{n}) = n \times \left(1 - \frac{1}{P1}\right) \times \left(1 - \frac{1}{P2}\right) \times \ldots \times \left(1 - \frac{1}{Pm}\right)$ 。

4. RSA 密码体系公钥私钥生成方式

RSA 密码体系公钥私钥生成方式如下。任意选取两个质数，p 和 q，然后，设$\mathrm{n} = \mathrm{p} \times \mathrm{q}$；函数$\Phi(\mathrm{n})$为 Euler 函数，返回小于 n 且与 n 互质的正整数个数；选择一个任意正整数 e，使其与$\Phi(\mathrm{n})$互质且小于 Φ(n)，公钥{e，n}已经确定；最后确定 d，使得 $\mathrm{d} \times \mathrm{e} \equiv 1 \% \Phi(\mathrm{n})$，即$(\mathrm{d} \times \mathrm{e} - 1) \% \Phi(\mathrm{n}) = 0$，至此，私钥{d，n}也被确定。

4.3 实例编程练习

4.3.1 编程训练要求

本章训练要求读者在第三章“基于 DES 加密的 TCP 通信”的基础上进行二次开发，使原有的程序可以实现全自动生成 DES 密钥以及基于 RSA 算法的密钥分配。

(1) 要求在 Linux 操作系统中完成基于 RSA 算法的保密通信程序的编写。

(2) 程序必须包含 DES 密钥自动生成、RSA 密钥分配以及 DES 加密通讯三个部分。

(3) 要求程序实现全双工通信，并且加密过程对用户完全透明。

(4) 用能力的同学可以使用 select 模型或者异步 IO 模型对“基于 DES 加密的 TCP 通信”一章中 socket 通讯部分代码进行优化。

4.3.2 编程训练设计与分析

1. 程序总体流程

程序执行过程如下，在客户端与服务器建立连接后，客户端首先生成一个随机的 DES 密钥，在第二章的程序里要求密钥长度为 64 位，所以使用长度为 8 的字符串充当密钥；同时，服务端生成一个随机的 RSA 公钥/私钥对，并将 RSA 公钥通过刚刚建立起来的 TCP 连接发送到客户端主机；客户端主机在收到该 RSA 公钥后，使用公钥加密自己生成的 DES 密钥，并将加密后的结果发送给服务器端；服务器端使用自己保留的私钥解密客户端发过来的 DES 密钥 ，最后双方使用该密钥进行保密通信。程序的流程图如下图 4-2 所示。

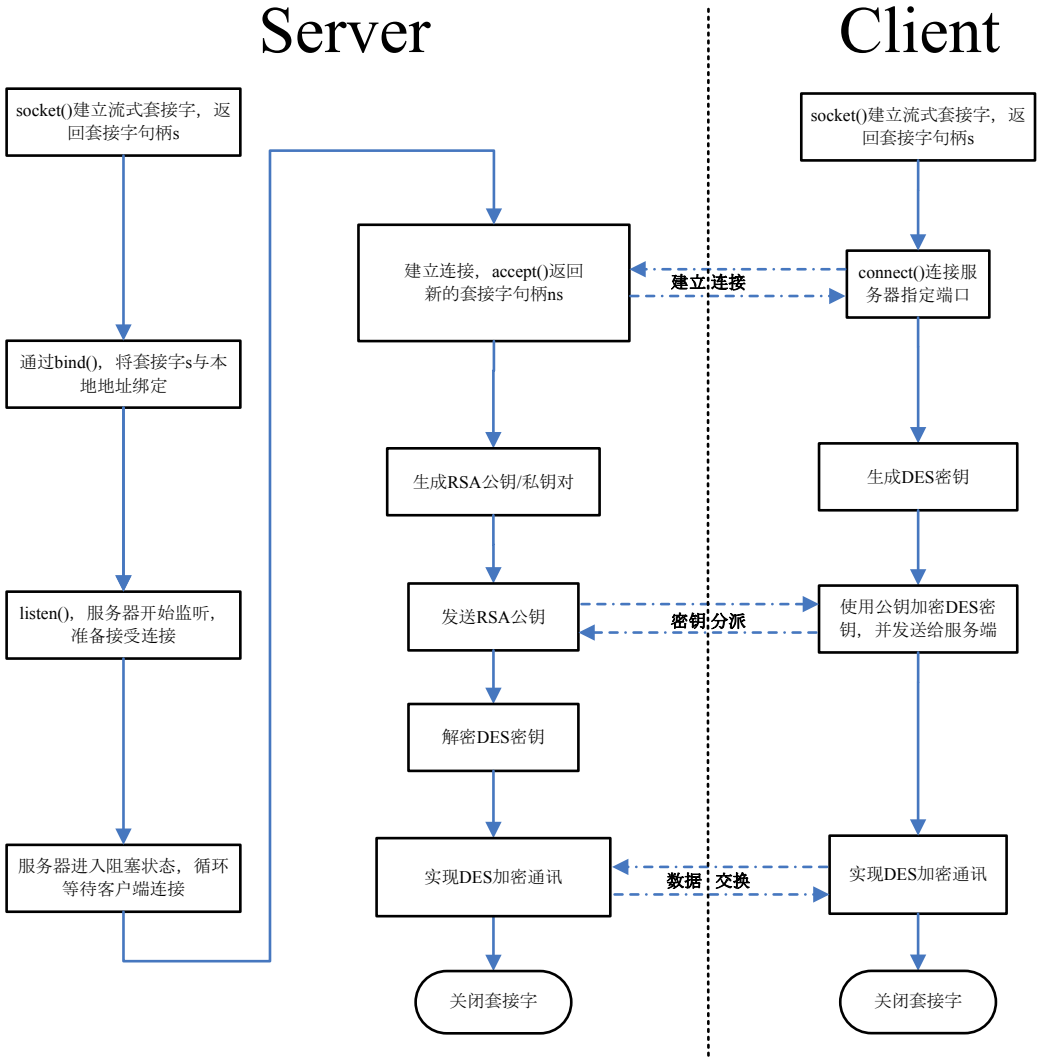

图 4-2 程序执行流程图

2. 模乘运算和模幂运算

模乘运算即计算两个数的乘积然后取模，代码如下。

```
inline unsigned __int64 MulMod(unsigned __int64 a, unsigned __int64 b, unsigned __int64 n)
{
    return (a% n) * (b% n) % n;
}
```

程序为了提升运算速度，根据求模运算的性质$\forall x,y,n|((x\%n)\times(y\%n))\%n =$（x×y）%n，优化了算法。

模幂运算即首先计算某数的若干次幂，然后对其结果进行取模运算，函数实现代码如下。

```
unsigned __int64 PowMod(unsigned __int64 base, unsigned __int64 pow, unsigned __int64 n)
{
```

```
    unsigned __int64    a=base, b=pow, c=1;
    while(b)
    {
        while(!(b & 1))
        {
            b>>=1;
            a=MulMod(a, a, n);
        }
        b--;
        c=MulMod(a, c, n);
    }
    return c;
}
```

由于 $\forall x,y,n \mid ((x\%n)\times(y\%n))\%n = (x\times y)\%n$，所以可以推出 $\forall x,z,n \mid ((x\%n)^z)\%n = (x^z)\%n$。此外 $\forall \alpha,\beta,\gamma \mid \gamma^{\alpha+\beta} = \gamma^{\alpha}\times\gamma^{\beta}, \gamma^{\alpha\times\beta} = \gamma^{\alpha^{\beta}}$。

在代码中，笔者根据上述公式进行优化。变量 a、b 分别代表计算过程中未完成乘方运算的底数和指数，c 为运算中间变量，用于存储运算结果。程序使用外层 while 循环用来遍历代表未完成乘方操作次数 b。 在该循环内，首先考虑未完成的乘方次数是否为偶数，如果为偶数，设 $b = 2\times k$，那么根据上述讨论，$a^b\%n = a^{2\times k}\%n = a^{2^k}\%n = (a^2\%n)^k\%n$。内层的 while 循环即完成此操作，每次循环，b 变为原来的 $\frac{1}{2}$，而 a 变为 $a^2\%n$。

内层循环在未完成乘方次数 b 为奇数时跳出，程序继续处理此种情况。基于上述讨论，设 $b = 2\times k+1$，则 $a^b\%n = a^{2\times k+1}\%n = ((a^{2k}\%n)\times a\%n)\%n$。在程序中首先将 b 值减 1，然后将 $a\%n$ 的值暂存如变量 c，而最后一次模 n 的操作将在 b=1 程序即将返回时执行。

故程序循环处理乘方操作，并将求模操作分布到乘方运算过程中，直到全部乘方操作运算完成，跳出外层循环并将结果返回。

3. 生成随机的大质数

本文使用基于 Fermat 定理“p 是一个质数，整数 a 与 p 互素，那么 $a^p \equiv 1\%p$”的 Rabin-Miller 质数测试方法进行质数判断，该方法是一种基于概率的质数判别方法。其利用此方法设计质数判别函数代码如下。

```
long RabinMillerKnl(unsigned __int64 &n)
{
    unsigned __int64    a, q, k, v;
    q=n - 1;
    k=0;
    while(!(q & 1))
    {
        ++k;
        q>>=1;
    }
```

程序首先计算出 q、k，使得 $n-1 = q\times 2\times k$，其中 q 是正奇数，k 是非负整数。

```
    a=2 + cRadom.Random(n - 3);
    v=PowMod(a, q, n);
    if(v == 1)
    {
        return 1;
    }
```

随机取一个 a，使$2 \le a < n-1$，然后计算$a^q \%n$，如果算$a^q \%n = 1$，通过测试，代表该数字可能是质数。

```
    for(int j=0;j<k;j++)
    {
        unsigned int z=1;
        for(int w=0;w<j;w++)
        {
            z*=2;
        }
        if(PowMod(a, z*q, n)==n-1)
            return 1;
    }
    return 0;
}
```

最后循环检验$a^{zq}\%n\ (z = 2^j)$的值，如果该值等于 n-1，则证明此数字可能是一个质数，否则本数为合数。

但是，基于 Rabin-Miller 的检验方法为概率算法，一个奇合数有四分之一的概率通过检验，故在实际应用中，需要通过多次重复检验，以增加验证正确的概率。

基于重复调用 Rabin-Miller 质数判别函数代码如下。

```
long RabinMiller(unsigned __int64 &n, long loop=100)
{
    for(long i=0; i < loop; i++)
    {
        if(!RabinMillerKnl(n))
        {
            return 0;
        }
    }
    return 1;
}
```

在完成质数判断函数RabinMiller的设计后，最终设计质数生成函数RandomPrime如下。

```
unsigned __int64 RandomPrime(char bits)
{
    unsigned __int64    base;
    do
    {
        base= (unsigned long)1 << (bits - 1);   //保证最高位是 1
        base+=cRadom.Random(base);      //再加上一个随机数
```

```
        base|=1;                                    //保证最低位是 1,即保证是奇数
    } while(!RabinMiller(base, 30)); //进行拉宾－米勒测试 30 次
return base; //全部通过认为是质数
}
```

该函数首先生成一个确保最高位是一（确保足够大）的随机奇数，然后，检验该奇数是否是质数，如该奇数不是质数，则重复该过程直到生成所需质数为止。

4. 求最大公约数

求最大公约数的代码运用了欧几里德辗转相除法。列出代码如下。

```
unsigned __int64 Gcd(unsigned __int64 &p, unsigned __int64 &q)
{
    unsigned __int64    a=p > q ? p : q;
    unsigned __int64    b=p < q ? p : q;
    unsigned __int64    t;
    if(p == q)
    {
        return p;   //两数相等,最大公约数就是本身
    }
    else
    {
        while(b)    //辗转相除法,gcd(a,b)=gcd(b,a-qb)
        {
            a=a % b;
            t=a;
            a=b;
            b=t;
        }
   return a;
    }
}
```

使用循环尝试法也可以计算两个数的公约数，但是运算效率较低。

5. 私钥生成

计算私钥主要就是计算 d 的值，根据第三节介绍的基础知识，d 必须满足$(d\times e - 1)\%\Phi(n) = 0$,所以,求d(已知e和$\Phi(n)$)的过程等价于寻找二元方程 $e\times d - \Phi(n)\times i = 1$的最大整数解（i 为另一未知量）。

程序实现代码如下。

```
unsigned __int64 Euclid(unsigned __int64 e, unsigned __int64 t_n)
{
    unsigned __int64 Max=0xffffffffffffffff-t_n;
    unsigned __int64 i=1;
```

```
    while(1)
    {
        if(((i*t_n)+1)%e==0)
        {
            return ((i*t_n)+1)/e;
        }
        i++;
        unsigned __int64 Tmp=(i+1)*t_n;
        if(Tmp>Max)
        {
            return 0;
        }
    }
    return 0;
}
```

程序在一个循环中不断从小到大尝试可能的 i 值，何时$\Phi(n)\times i+1$能被 e 整除，则返回对应的 d 值，程序返回，否则，一直不断尝试更大的直到数据超过阈值，则返回 0 表示密钥生成失败。

6. 密钥分配

为了方便程序编写，在示例代码中把 DES 加密解密和 RSA 加密解密的代码分别封装在类 CDesOperate 和 CRSASection 中。

在 CRSASection 中导出三个函数，分别用来进行加密，解密以及导出密钥。

(1) 加密函数 Encry

```
static UINT64 Encry(unsigned short nSorce, PublicKey &cKey)
{
    return PowMod(nSorce, cKey.nE, cKey.nN);
}
```

此函数是加密函数，使用公钥，通过模幂运算实现计算$C=M^e \bmod n$ 的加密过程，由于公钥本身并不始终保存在类的成员变量里，所以加密函数设计为 static，并通过参数传递公钥。

(2) 解密函数 Decry

```
    unsigned short Decry(UINT64 nSorce )
    {
        UINT64 nRes = PowMod(nSorce, m_cParament.d, m_cParament.n);
        unsigned short * pRes=(unsigned short*)&(nRes);
        if(pRes[1]!=0||pRes[3]!=0||pRes[2]!=0)
        {//error
            return 0;
        }
        else
        {
            return pRes[0];
```

```
        }
    }
```

函数 Decry()用于进行解密计算，即实现$M = C^d mod\ n$的计算。其中，密钥由保存在类成员变量中的结构实体 m_cParament 提供。

(3) 公钥获取函数 GetPublicKey

```
PublicKey GetPublicKey()
{
    PublicKey cTmp;
    cTmp.nE=this->m_cParament.e;
    cTmp.nN = this->m_cParament.n;
    return cTmp;
}
```

本函数用来输出当前使用的公钥，其中 PublicKey 是一个结构体，用于保存公钥中的两个整数。

加密函数的输入和解密函数输出为短整形变量，这是因为虽然理论上 RSA 可以加密，解密任意小于 n 的整数（n 为 64 位），但是在中间计算中仍可能生成大于 n 的临时变量。本程序为了简化编写并未使用专业大整数函数库，这就造成如果加密函数输入过大在进行乘法操作时溢出。故限制加密函数解密函数输入输出范围为短整形。

(4) 生成公钥私钥

在 socket 连接建立起来以后，首先服务端通过调用函数 RsaGetParam()初始化与 RSA 加密相关的各项参数，如公钥，私钥等。

程序代码如下。

```
RsaParam RsaGetParam(void)
{
    RsaParam            Rsa={ 0 };
    UINT64    t;
    Rsa.p=RandomPrime(16);           //随机生成两个素数
    Rsa.q=RandomPrime(16);
    Rsa.n=Rsa.p * Rsa.q;
    Rsa.f=(Rsa.p - 1) * (Rsa.q - 1);
    do
    {
        Rsa.e=m_cRadom.Random(65536);
        Rsa.e|=1;
    } while(Gcd(Rsa.e, Rsa.f) != 1);
    Rsa.d=Euclid(Rsa.e, Rsa.f);
    Rsa.s=0;
    t=Rsa.n >> 1;
    while(t)
    {
        Rsa.s++;
        t>>=1;
    }
    return Rsa;
```

```
}
```

这个函数会在类的构造函数中自动调用。

(5) DES 密钥分配

此后，程序会自动进行 DES 密钥的分配。

（1） 首先，服务端生成 RSA 密码的公钥与私钥，并将私钥通过 socket 传送到客户端。服务端操作代码如下。

```
CRSASection cRsaSection;
cRsaPublicKey = cRsaSection.GetPublicKey();
if(send(nAcceptSocket,
            (char                                          *)(&cRsaPublicKey),
sizeof(cRsaPublicKey),0)!=sizeof(cRsaPublicKey))
{
    perror("send");
    exit(0);
}
else
{
    printf("successful send the RSA public key. \n");
}
UINT64 nEncryptDesKey[DESKEYLENGTH/2];
if(DESKEYLENGTH/2*sizeof(UINT64) != TotalRecv(nAcceptSocket,
            (char *)nEncryptDesKey,DESKEYLENGTH/2*sizeof(UINT64),0))
{
    perror("TotalRecv DES key error");
    exit(0);
}
else
{
    printf("successful get the DES key. \n");
    unsigned short * pDesKey = (unsigned short *)strDesKey;
    for(int i = 0;i<DESKEYLENGTH/2;i++)
    {
        pDesKey[i] = cRsaSection.Decry(nEncryptDesKey[i]);
    }
}
printf("Begin to chat...\n");
SecretChat(nAcceptSocket,inet_ntoa(sRemoteAddr.sin_addr),strDesKey);
close(nAcceptSocket);
```

（2） 客户端首先调用函数 GerenateDesKey()生成随机 DES 密钥，然后获得服务端提供的 RSA 公钥，并使用该公钥加密 DES 密钥，然后将加密后 DES 密钥发回给服务端，从而实现 DES 密钥可靠共享。客户端操作代码如下。

```
printf("Connect Success!  \n");
GerenateDesKey(strDesKey);
printf("Create DES key success\n");
```

```
if(sizeof(cRsaPublicKey)==TotalRecv(nConnectSocket,(char *)&cRsaPublicKey,
                        sizeof(cRsaPublicKey),0))
{
    printf("Successful get the RSA public Key\n");
}
else
{
    perror("Get RSA public key ");
    exit(0);
}
UINT64 nEncryptDesKey[DESKEYLENGTH/2];
unsigned short *pDesKey = (unsigned short *)strDesKey;
for(int i = 0;i<DESKEYLENGTH/2;i++)
{
    nEncryptDesKey[i] = CRSASection::Encry(pDesKey[i],cRsaPublicKey);
}
if(sizeof(UINT64)*DESKEYLENGTH/2!=send(nConnectSocket, (char *)nEncryptDesKey,
                        sizeof(UINT64 )*DESKEYLENGTH/2,0))
{
    perror("Send DES key Error");
    exit(0);
}
else
{
    printf("Successful send the encrypted DES Key\n");
}
printf("Begin to chat...\n");
SecretChat(nConnectSocket,strIpAddr,strDesKey);
```

(6) 加密全双工通信

最后调用函数 SecretChat() 实现加密全双工通信。SecretChat()实现细节参见第二章相关叙述。

可见，基于 RAS 算法的密钥分配增加了原有程序的安全性。攻击者只能通过监听截获 RSA 公钥和使用 RSA 公钥加密后的 DES 密钥，却无法获得对应的 RSA 私钥，故无法解密 DES 密钥，进而可以保证 DES 加密通信的安全性。

此外，由于 RSA 算法执行运算量较大，所以只使用 RSA 算法用于密钥共享，而不是直接使用其加密通信内容，以降低系统资源消耗。

4.4 扩展与提高

4.4.1 RSA 安全性

RSA 算法是第一个能同时用于加密和数字签名的算法，也易于理解和操作。同时它也是应用范围最广的公钥密码体系，从提出到现在已近二十年，经历了各种攻击的考验，逐渐为

人们接受。目前，公众普遍认为 RSA 是最优秀的公钥方案之一。但是 RSA 的安全性依赖于大数的因子分解，但并没有从理论上证明破译 RSA 的难度与大数分解难度等价。也就是说，无法从理论上证明不可能存在一种不需要进行大数分解及可以破解 RSA 的算法，即 RSA 的重大缺陷是无法从理论上把握它的保密性能如何，而且密码学界多数人士倾向于因子分解不是 NPC 问题。目前也有很多科学家在相关领域进行研究。

目前，对 RSA 算法的攻击常用算法有以下三种方式：

（1） 穷举攻击：试图穷举所有可能的私钥；

（2） 数学攻击：方法多种多样，但其本质就是试图分解 n，求得 p 和 q，进而推算出密钥；

（3） 计时攻击：这类算法依赖于观测解密算法的运行时间，攻击者从算法运行时间中获得额外信息进行；

对抗穷举攻击，RSA 也是使用大的密钥空间，所以进行选择时 d 和 e 越大越好，但是密钥产生过程和加密解密过程都需要经过复杂的运算，所以这两个数选择的越大，加密解密所需的时间越长，需要程序员在两者之间选择最佳的平衡点。

数学攻击主要指因子分解攻击，即分解 n 为两个质因子，从而计算出 $\Phi(n) = (p-1)\times(q-1)$，而进一步根据 $d\times e \equiv 1 \% \Phi(n)$ 根据 e 确定 d 的取值，从而猜测密钥，进行破解；

目前，虽然分解具有大质数因子的 n 仍然是一个难题，但是已经逐渐被科学界攻破，现在破解固定长度共钥所需的时间正在逐年递减，所以，要想确保 RSA 算法的安全性，就必须保证 n 足够大，此外 RSA 的发明者建议 p 和 q 满足下列条件：

（1） p 和 q 长度应仅相差几位；

（2） (p - 1)和(q - 1)都应该有一个大的质因子；

（3） (p - 1)和(q - 1)的最大公约数应该比较小；

此外，已经证明若 e<n 且 d>n 1/4 ，则 d 容易被确定；

至于计时攻击，是一种通过记录计算机解密消息所用时间来确定私钥的一种攻击方式，类似于观察他人转动保险柜拨号盘的时间长短来猜测密码。计时攻击并非针对 RSA，而是可以攻击全部的公钥密码体系，所以其危害比较严重。例如，在攻击 RSA 算法时，因为在进行加密时所进行的模指数运算是逐位进行的，而位为 1 所花的运算比位为 0 的运算要多很多，故其通过观察得到多组信息与其加密时间，就可以尝试反推出私钥的内容。

程序编写者可以使用一些简单的办法防御此类攻击：

（1） 保证所有的幂运算在返回结果之前所用的时间相同；

（2） 在求幂算法中加入随机延时；

（3） 通过执行幂运算前将密文乘上一个随机数进行隐藏；

总之，就目前的技术水平来说，分解 n 依然是针对 RSA 算法最主要的攻击方法。现在，现有技术水平已能分解 140 位（十进制）的大素数。因此，模数 n 必须尽量选择大数，才能保证 RSA 算法的安全性。在实际应用中，1997 年后开发的系统，已经使用 1024 位密钥，而安全要求较高的证书认证机构应用 2048 位或以上密钥。

4.4.2 其他公钥密码体系

椭圆曲线密码学（ECC, Elliptic curve cryptography）是基于椭圆曲线数学的一种公钥密码的方法。椭圆曲线在密码学中的使用是在 1985 年由 Neal Koblitz 和 Victor Miller 分别独立提出的。

椭圆曲线密码体制来源于对椭圆曲线的研究，所谓椭圆曲线指的是由韦尔斯特拉斯（Weierstrass）方程所确定的平面曲线。其并非真的椭圆曲线，只是因为其方程形式类似求解椭圆形周长的公式故得其名，椭圆曲线密码体制中用到的椭圆曲线都是定义在有限域上的，即最终方程形式如下。$y^2 \bmod p = (x^3 + ax + b) \bmod p$。

首先定义椭圆曲线加法运算规则：若椭圆曲线上的三个点在同一条直线上，则其和为零（具体定义参考密码学相关图书）。

设两点 P 和 Q，则在方程 kP = Q 中，已知 k 和点 P 求点 Q 比较容易，反之已知点 Q 和点 P 求 k 却是相当困难的，这个问题称为椭圆曲线上点群的离散对数问题。椭圆曲线密码体制正是利用这个困难问题设计而来。椭圆曲线应用到密码学上最早是由 Neal Koblitz 和 Victor Miller 在 1985 年分别独立提出的。

椭圆曲线密码体制是目前已知的公钥体制中，对每比特所提供加密强度最高的一种体制。解椭圆曲线上的离散对数问题的最好算法是 Pollard rho 方法，其时间复杂度是完全指数阶的。当密钥大小为 150 时，破解需要 $3.8*10^{10}$ MIPS（Million Instructions Per Second 的缩写，每秒处理的百万级的机器语言指令数。这是衡量 CPU 速度的一个指标）年，当密钥增大到 234 时，破解时间就可达到 $1.6*10^{28}$ MIPS 年。而大家熟知的 RSA 所利用的是大整数分解的困难问题，目前对于一般情况下的因数分解的最好算法的时间复杂度是子指数阶的，当密钥长度为 512 时，只需要 $3x10^4$MIPS 年，即便密钥长度增长到 2048，破解时间也不过增加到 $3*10^{20}$ MIPS 年（MIPS 年是指每秒运行百万条指令的处理器运行一年的计算量）。也就是说当 RSA 的密钥使用 2048 位时，ECC 的密钥使用 234 位所获得的安全强度还高出许多。它们之间的密钥长度却相差达 9 倍，当 ECC 的密钥更大时它们之间差距将更大。可见 ECC 密钥短的优点是非常明显的。

国家标准与技术局和 ANSI X9 已经设定了最小密钥长度的要求，RSA 和 DSA 是 1024 位，ECC 是 160 位，相应的对称分组密码的密钥长度是 80 位。NIST 已经公布了一列推荐的椭圆曲线用来保护 5 个不同的对称密钥大小(80, 112, 128, 192, 256)。一般而言，二进制域上的 ECC 需要的非对称密钥的大小是相应的对称密钥大小的两倍。

在 2005 年 2 月 16 日，NSA 宣布决定采用椭圆曲线密码的战略作为美国政府标准的一部分，用来保护敏感但不保密的信息。

4.4.3 使用 Select 机制进行并行通信

1. Linux select IO 操作方式简介

为了提升程序效率，Linux 提供了 select 函数接口，用以同时管理若干个套接字或者句柄上的 IO 操作，通过该 API，程序可以同时监控多个 socket 或者句柄上的 IO 操作，进而可以免去开启多个进程的系统开销。

函数定义如下。 int select(int n,fd_set * readfds,fd_set * writefds,fd_set * exceptfds,struct timeval * timeout)。

其中，参数 n 代表最大文件句柄，必须赋值为监控的全部句柄中最大值加一，参数 readfds、writefds 和 exceptfds 对应三个句柄集合，是用来通知系统分别监控发生在对应集合中所包括句柄上的读，写或错误输出事件。

下面的一组宏用于操作上述句柄。

（1） FD_CLR(inr fd,fd_set* set)；用来清除句柄集合 set 中相关 fd 的项；

（2） FD_ISSET(int fd,fd_set *set)；用来测试句柄集合 set 中相关 fd 的项是否为真；

（3） FD_SET（int fd,fd_set*set)；用来设置句柄集合 set 中相关 fd 的项；

（4） FD_ZERO（fd_set *set)；用来清除句柄集合 set 的全部项；

参数 timeout 为结构 timeval，用来设置 select()的等待时间，其结构定义如下

```
struct timeval
{
time_t tv_sec;
time_t tv_usec;
};
```

如果参数 timeout 设为 NULL 则表示 select（）一直等待不会超时。

函数执行成功则返回文件句柄状态已改变的个数,如果返回 0 代表在句柄状态改变前已超、间，当有错误发生时则返回-1，错误原因存于 errno：

（1） EBADF： 文件句柄为无效的或该文件已关闭；

（2） EINTR： 此调用被中断；

（3） EINVAL： 参数 n 为无效；

（4） ENOMEM： 核心内存不足；

2. 使用 select 优化函数 SecretChat

使用 select()重写后的 SecretChat()代码如下。

```
void SecretChat(int nSock,char *pRemoteName, char *pKey)
{
    CDesOperate cDes;
    fd_set cHandleSet;
    struct timeval tv;
    int nRet;
```

初始化相关变量，并在一个循环中监控套接字和标准输入。

```
    while(1)
    {
        FD_ZERO(&cHandleSet);
        FD_SET(nSock, &cHandleSet);
        FD_SET(0, &cHandleSet);
        tv.tv_sec = 1;
        tv.tv_usec = 0;
        nRet = select(nSock>0? nSock+ 1:1, &cHandleSet, NULL, NULL, &tv);
```

这里可见，程序只监控套接字和标准输入上的读操作。并分别进行处理；此外，每循环一次，程序就必须重新设置句柄集合中的内容。

```
        if(nRet< 0)
        {
            printf("Select ERROR!\n");
            break;
        }
        if(0==nRet)
        {
```

```
            continue;
        }
```

在排除超时和出错的可能后，程序通过下面的两个 if 分别判断套接字和标准输入上是否发生 io 操做，如有发生，则调用 recv()进行读取，并处理获得的数据。

```
        if(FD_ISSET(nSock,&cHandleSet))
        {
            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;
                    }
                }
            }
        }

        if(FD_ISSET(0,&cHandleSet))
        {
            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;
                }
            }
        }
    }
}
```

4.4.4 使用异步 IO 进行通信优化

1. 同步 IO 操作和异步 IO 操作的比较

Linux 还提供一种异步 IO 机制对程序效率进行优化。一般同步 IO 调用会在系统 IO 操作完成后返回，在该 IO 未能完成时调用函数会将调用进程挂起等待；而异步 IO 调用会在系统调用后直接返回，在系统内核真正完成调用后，通过消息或者回调函数通告调用进程本次 IO 执行结果。图 4-3 和 4-4 展示了 Linux 系统同步 IO 操作和异步 IO 操作的区别：

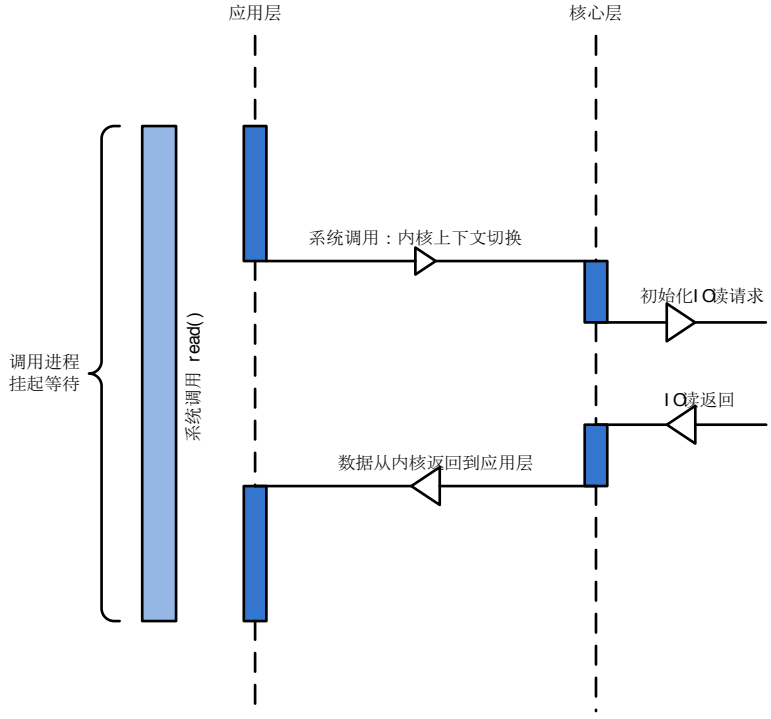

图 4-3 Linux 同步 IO 执行过程

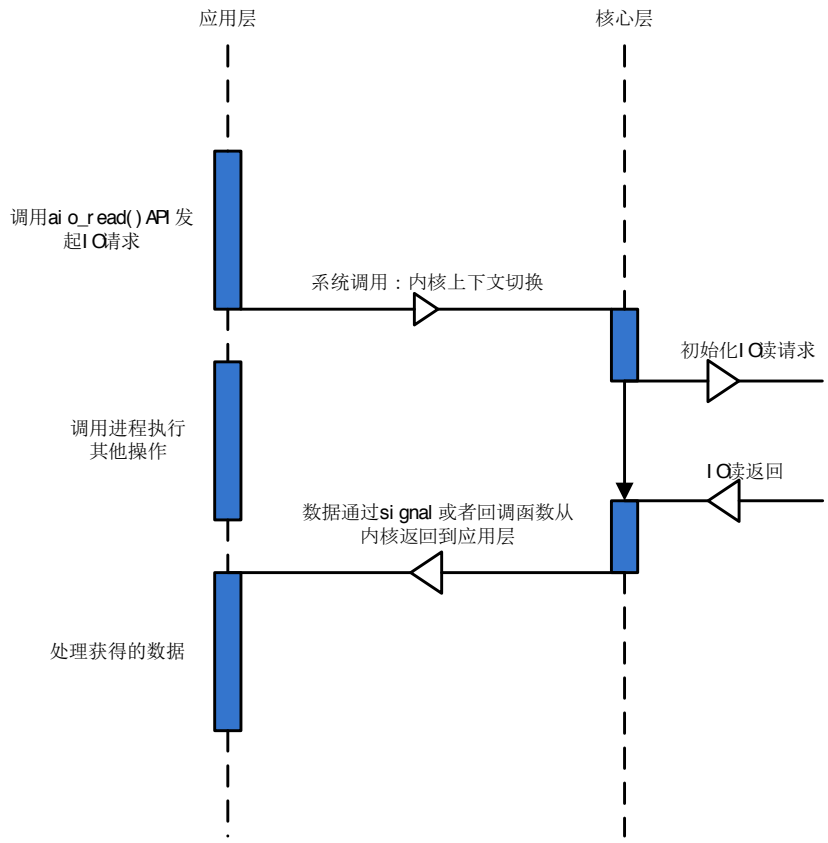

图 3-5 Linux 异步 IO 执行过程

使用异步 IO 需要包含头文件<aio.h>，并在编译时指定编译选项 g++ chat.cpp -lrt。

异步 IO 调用主要需要以下 API：

（1） aio_read()：请求异步读操作；

（2） aio_error()：检查异步请求的状态；

（3） aio_return()：获得完成的异步请求的返回状态；

（4） aio_write()：请求异步写操作；

（5） aio_suspend()：挂起调用进程，直到一个或多个异步请求已经完成(或失败)；

（6） aio_cancel()：取消异步 I/O 请求；

（7） lio_listio()：发起一系列 I/O 操作；

其中，由于在异步非阻塞 I/O 中，程序可以同时发起多个传输操作。所以程序要求每个传输操作都有惟一的上下文，以便在收到内核 IO 完成通知时区分该 IO 通知来自哪个 IO 请求。这个工作由 aiocb 结构完成。该结构包含了有关传输的所有信息，包括为数据准备的用户缓冲区，通知的方式，回调函数的地址和参数等。在调用 IO 请求 API 时，程序为该结构体赋值，并将该结构体的地址以参数形式传入内核。

2. 使用异步 IO 优化函数 SecretChat

下述示例程序就是通过异步 IO 进一步优化函数 SecretChat()执行效率。

```
void SecretChat(int nSock,char *pRemoteName, char *pKey)
{
    pid_t nPid;
    nPid = fork();
    sem_t bStop;
```

```
    sem_init(&bStop,0,0);
    if(nPid != 0)
    {
        SockinSingle *pSockin = new SockinSingle(pKey,nSock,bStop);
        aio_read(pSockin->m_pReq);
        sem_wait(&bStop);
    }
    else
    {
        StdinSingle *pStdin = new StdinSingle(pKey,nSock,bStop);
        aio_read(pStdin->m_pReq);
        sem_wait(&bStop);
    }
    sem_destroy(&bStop);
}
```

程序创建两个进程，分别在标准输入和 socket 上发起一个异步读操作，然后就通过 sem_wait()等待程序结束。所有的数据操作都在对应 IO 结束后的回调函数中进行。

由于监控 socket 和监控标准输入的两个进程工作流程基本一致。所以在此只分析监控标准输入的进程。监控 socket 的相关代码可参考随书光盘内容。

监控标准输入的进程由类 StdinSingle 完成：

```
class StdinSingle
{
public:
    StdinSingle(char *pKey,int nSock,sem_t &bStop)
        :m_bStop(bStop)
    {
        memcpy(m_strKey,pKey,8);
        m_strKey[8]=0;
        this->m_nSock=nSock;
        bzero(&strStdinBuffer, BUFFERSIZE);
        this->m_pReq = new aiocb;
        bzero( (char *)m_pReq, sizeof(struct aiocb) );
        m_pReq->aio_fildes = 0;
        m_pReq->aio_buf = strStdinBuffer;
        m_pReq->aio_nbytes = BUFFERSIZE;
        m_pReq->aio_offset = 0;
        m_pReq->aio_sigevent.sigev_notify = SIGEV_THREAD;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._function                =
    StdinReadCompletionHandler;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._attribute = NULL;
        m_pReq->aio_sigevent.sigev_value.sival_ptr = this;
    };
    ~StdinSingle()
    {
```

```
        delete m_pReq;
        bzero(&strStdinBuffer, BUFFERSIZE);
    };
    char m_strKey[9];
    int m_nSock;
    aiocb *m_pReq;
    sem_t &m_bStop;
    static void StdinReadCompletionHandler( sigval_t sigval )
    {
        StdinSingle* pThis = (StdinSingle*)sigval.sival_ptr;
        if (aio_error( pThis->m_pReq ) == 0)
        {
            int nSize = aio_return( pThis->m_pReq );
            CDesOperate cDes;
            int nLen = BUFFERSIZE;

    cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pThis->m_strKey,8)
;
            SockoutSingle *pSockoutSingle = new SockoutSingle(
pThis->m_nSock,
pThis->m_strKey,
pThis->m_bStop);
            aio_write(pSockoutSingle->m_pReq);
            if(0==memcmp("quit",strStdinBuffer,4))
            {
                printf("Quit!\n");
                sem_post(&pThis->m_bStop);
                exit(0);
            }
        }
        delete pThis;
        return;
    };
};
```

这个类主要在构造函数中初始化相关 aiocb 结构，并指定其回调函数指针为 StdinReadCompletionHandler。

在该函数中，程序处理从标准输入读到的数据，并将其加密，然后通过 SockoutSingle 类调用 aio_write(pSockoutSingle->m_pReq)将加密后的数据Ê发送。

SockoutSingle 类的代码如下。

```
class SockoutSingle
{
public:
    char m_strKey[9];
    int m_nSock;
```

```
    aiocb *m_pReq;
    sem_t &m_bStop;
    SockoutSingle(int nSock,char *pKey,sem_t &bStop)
        :m_bStop(bStop)
    {
        memcpy(m_strKey,pKey,8);
        m_strKey[8]=0;
        this->m_nSock=nSock;
        this->m_pReq = new aiocb;
        bzero( (char *)m_pReq, sizeof(struct aiocb) );
        m_pReq->aio_fildes = nSock;
        m_pReq->aio_buf = strEncryBuffer;
        m_pReq->aio_nbytes = BUFFERSIZE;
        m_pReq->aio_offset = 0;
        m_pReq->aio_sigevent.sigev_notify = SIGEV_THREAD;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._function                =
    SockoutReadCompletionHandler;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._attribute = NULL;
        m_pReq->aio_sigevent.sigev_value.sival_ptr = this;
    };
    ~SockoutSingle()
    {
        delete m_pReq;
    };
    static void SockoutReadCompletionHandler( sigval_t sigval )
    {
        SockoutSingle* pThis = (SockoutSingle*)sigval.sival_ptr;
        if (aio_error( pThis->m_pReq ) == 0)
        {
            int nSize = aio_return( pThis->m_pReq );
            if(nSize != BUFFERSIZE)
            {
                perror("Error Send!\n");
            }
            else
            {
                StdinSingle *pStdin = new StdinSingle(pThis->m_strKey,
pThis->m_nSock,
pThis->m_bStop);
                aio_read(pStdin->m_pReq);
            }
        }
        delete pThis;
        return;
```

```
    }
};
```

这个类的工作流程和 StdinSingle 基本相同，其工作主要在回调函数 SockoutReadCompletionHandler 中完成：

当加密数据 Ê 发送完成后，函数 SockoutReadCompletionHandler()被调用，该函数在标准输入上发起读操作， 从而驱动类 StdinSingle 继续工作。标准输入读入数据后会再次调用类 StdinSingle 中指定的回调函数 StdinReadCompletionHandler()。如此循环，直到用户输入“quit”命令，类 StdinSingle 通过 sem_post(&pThis->m_bStop)设置信号量，通知主进程结束,并自行退出。

异步模型的执行效率优势如何体现呢？每次 Linux 系统调用就会在内核和用户之态之间进行一次上下文切换，需要消耗系统资源，使用异步 IO 模型，两个主进程在发起第一次读操作后就通过 sem_wait()等待，直到结束直接退出，整个过程中都不需要进行上下文切换，而真正进行数据操作的回调函数也都是在 IO 状态变化的时刻由内核自动调用。可见，这个模型可以消除无谓进程上下文切换所需的资源，进而大大提升系统的执行效率。

此外，当系统同时处理若干 socket 上的并发数据时，异步 IO 可以使单个进程具有监督多个 socket 上数据的能力，从而大量节约所需进程数目，大幅降低所需系统资源，提升程序效率。