



南开大学  
Nankai University

南 开 大 学

网 络 安 全 技 术

---

基于 RSA、DES 加密的 TCP 通信

---

学院：网络安全学院

年级：2021 级

班级：信息安全 1 班

学号：2111252

姓名：李佳豪

手机号：13191110713

2024 年 5 月 29 日

# 目录

一、 实验目的	1
二、 实验内容	1
三、 实验步骤	2
(一) 程序总览 . . . . .	2
(二) DES 部分 . . . . .	3
(三) RSA 部分 . . . . .	5
1. 选择两个不同的大素数 $p$ 和 $q_0$ . . . . .	5
2. 计算 $n \phi(n)$ . . . . .	7
3. 加密和解密 . . . . .	7
4. RSA.h . . . . .	8
(四) RSA 加密 DES 密钥 . . . . .	9
(五) 基于 socket 的 tcp 传输模块 . . . . .	11
(六) 消息模块和视图模块 . . . . .	12
(七) 控制模块 . . . . .	12
四、 实验结果	15
(一) RSA-单元测试 . . . . .	15
(二) RSA 分组加密单元测试 . . . . .	15
(三) 完整测试 . . . . .	16
五、 实验遇到的问题及其解决方法	17
(一) Rsa 分组加密、传输 . . . . .	17
(二) 64bit 随机数产生 . . . . .	17
(三) 编程问题 . . . . .	18
六、 实验结论	19

## 一、 实验目的

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。

## 二、 实验内容

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 “” 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。

**补充：**由于在 windows 下调试更方便，且 linux 和 Windows 下 socket 编程原理相同，因此延续上一次实验采用 windows。

### 三、 实验步骤

#### (一) 程序总览

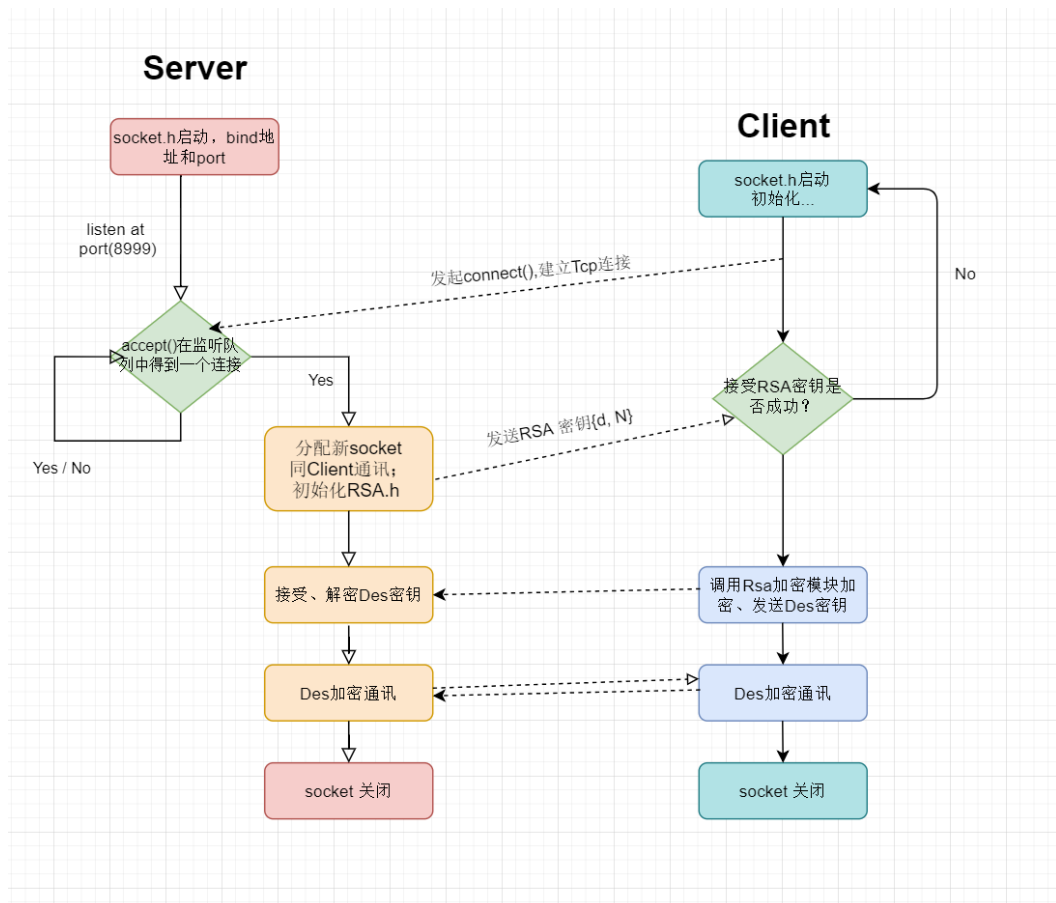


图 1: 程序总览

本次实验在上一次 DES 加密的 tcp 可靠通讯基础上, 增加了 RSA 部分, 我将程序分为如下模块: DES、RSA 加密模块、Socket 模块、消息模块和视图模块, 各个文件和模块有如下对应关系:

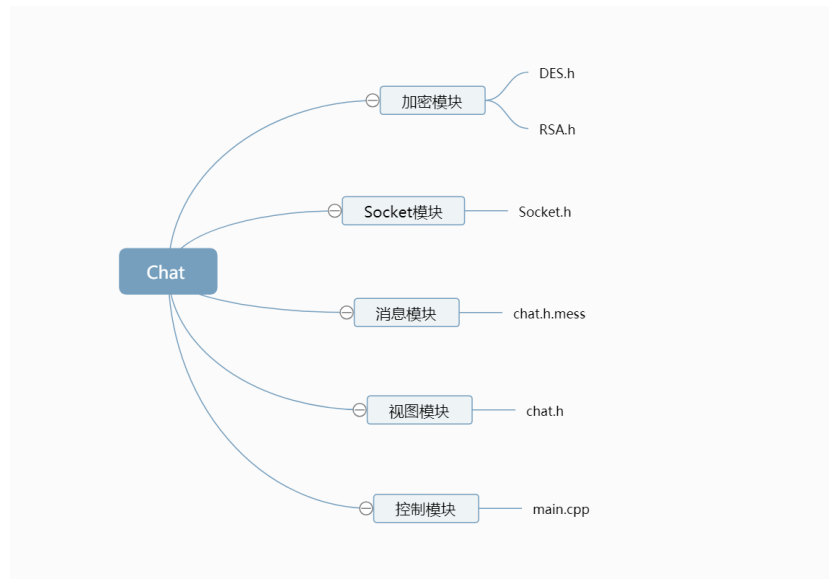


图 2

## (二) DES 部分

DES 模块提供所有接口如下:

### DES API

```

1  #pragma pack(1)
2  struct desTuple {
3      char name[4];
4      uint64_t k[4];
5      desTuple(uint64_t* N){
6          name[0] = 'D';
7          name[1] = 'E';
8          name[2] = 'S';
9          name[3] = '\0';
10         memcpy(k, N, 32);
11     }
12 };
13 #pragma pack()
14
15 uint64_t generate_random_key();
16
17 void set_key(uint64_t K);
18
19 void Encrypy(uint64_t plain, uint64_t* dst);
20
21 string Encrypy(const string& plain);
22
23 void Decrypy(uint64_t cyber, uint64_t* dst);
24
25 string Decrypy(const string& cyber);
  
```

```

26
27 uint64_t Get_key(uint64_t* Ks, int i, bool En);
28
29 void Des_Encrypt(uint64_t plain, uint64_t* dst, uint64_t* Ki, bool En);
30
31 void Key_exp(uint64_t* K, uint64_t* Ki);
32
33 void Rotate(uint32_t* src);
34
35 void Permute(uint64_t* src, uint64_t* dst, const uint8_t* table, int src_len,
    int dst_len);
36
37 void Substitute(uint64_t* src, uint64_t* dst);
38
39 string padString(const string& input, size_t block_size);
40
41 vector<string> splitIntoBlocks(const string& input, size_t block_size);
42
43 uint64_t block2ull(const string& block);

```

相比上一次实验，我增加了两个内容；

- 结构体 *desTuple*

```

struct desTuple
1 struct desTuple{
2     char name[4];
3     uint64_t k[4];
4     desTuple(uint64_t* N){
5         name[0] = 'D';
6         name[1] = 'E';
7         name[2] = 'S';
8         name[3] = '\0';
9         memcpy(k, N, 32);
10    }
11 };

```

考虑到传递 DES 私钥的条件，必须是 RSA 加密算法结束，Server 将密钥传递给 Client，此时 Client 才发送加密后的 DES 密钥信息，为了唯一区分这则信息，也为了更方便的发送，于是构造出结构体 *desTuple*。

**结构体内容解读如下：**

- 前四个字节是“DES”，是这则消息的签名；
- 后 4\*8 字节为加密后的 DES 密钥，每 8BYTE 为原 DES 密钥 16bit 加密后的结果。
- 随机生成 64 位 DES 密钥： *generate\_random\_key()*
  - 上一次实验中，我将 DES 密钥硬编码嵌入代码内。而这一次实验我们通过 RSA 加密 DES 密钥，若密钥仍然硬编码，则 RSA 加密的意义是没有的。

- 代码中通过使用 `random_device` 是一个均匀分布的整数随机数生成器，可生成不确定的随机数，相比 `random` 根据随机算法生成的伪随机数，`random_device` 可以生成高质量的随机数，可以理解为真随机数。

```

                                generate_random_key
1  uint64_t generate_random_key() {
2      std::random_device rd;
3      std::mt19937_64 rng(rd());
4      std::uniform_int_distribution<uint64_t> distribution(0, 0xfffffffffffffffff
        );
5      return distribution(rng);
6  }

```

### (三) RSA 部分

RSA (Rivest-Shamir-Adleman) 是一种非对称加密算法，被广泛用于数据加密和数字签名。算法使用了一对密钥：公钥和私钥。公钥用于加密数据，私钥用于解密数据。同时，私钥也用于对数字签名进行签名，而公钥用于验证签名的有效性。

#### 1. 选择两个不同的大素数 $p$ 和 $q$ 。

```

1  //RSA.cpp
2  RsaInit():
3      std::mt19937_64 rng(2); // 种子，可以选择时间作为seed
4      std::uniform_int_distribution<uint64_t> distribution(1, 65536); //
        设置范围
5
6      p = getRandom(16);
7      std::cout << "GET RNDOM NUMBER: " << p << std::endl;
8      q = getRandom(16);
9      std::cout << "GET RNDOM NUMBER: " << q << std::endl;

```

- **getRandom 函数生成素数**

函数内部通过 C++ 的标准库中的随机数生成器 `std::mt19937_64` 和均匀分布器 `std::uniform_int_distribution` 来生成一个范围在 1 到 65536 之间的随机数

**注：**在实际中应当生成更大的素数，但是在 RSA 计算过程中，16 位的  $p$ 、 $q$  中间计算过程将存在 64bit 的数，更大 bit 的数若无特殊硬件配合处理，则需要在编程层面考虑高精度数的计算，计算复杂度将大大增加。

```

1      uint64_t getRandom(uint8_t bits){
2          uint64_t base = 0;
3          do {
4              base = (uint64_t)1 << (bits - 1);
5              base += distribution(rng);
6          } while (!isPrime(base, 50));
7      }

```

```
8         return base;
9     }
```

- *isPrime()*

函数内部通过使用 Miller Rabin 素数判定法判断是否给定的数字是否为素数，算法原理参考[Miller Rabin 原理](#)

该算法本质上是一种随机化算法，能在  $k \log^3 n$  的时间复杂度下快速判断出一个数是否是素数，但具有一定的错误概率。因此通过连续 epoch 次进行 miller Rabin 判定，使得判断结果出错概率降低。

```
1 bool isPrime(uint64_t n, int epoch) {
2
3     if (n==1)
4         return 0;
5     if (n == 3 || n == 2)
6         return 1;
7
8     uint64_t d = n - 1;
9     int m = 0;
10    while (!(d % 2)) {
11        d /= 2;
12        m++;
13    }
14    for (int k = 0; k < epoch; k++)
15        if (!miller(m, d, n))
16            return 0;
17
18    return 1;
19
20 bool miller(int m, uint64_t d, uint64_t n) {
21
22     uint64_t a = 2 + rand() % (n - 2);
23     d = pow(a, d, n);
24
25     if ((d % n == 1) || (d % n == n - 1))
26         return 1;
27
28     for (int i = 0; i < m; i++) {
29         d = mul(d, n);
30         if (d % (n) == 1)
31             return 0;
32         if (d % (n) == n - 1)
33             return 1;
34     }
35     return 0;
36 }
```



## 2. 计算 $n \phi(n)$

$$n = p \times q$$

o

$$\phi(n) = (p - 1) \times (q - 1)$$

o

```
1      n = p * q;
2      f = (p - 1) * (q - 1);
```

计算公钥  $e$  和私钥  $d$

选择一个整数  $e$ , 满足  $1 < e < \phi(n)$  且  $e$  与  $\phi(n)$  互质。计算私钥  $d$ , 满足  $d \times e \equiv 1 \pmod{\phi(n)}$ 。

```
1      do
2      {
3          e = distribution(rng);
4          e |= 1;
5      } while (gcd(e, f) != 1);
6      d = euclid(e, f);
```

通过 `distribution` 随机生成 16 位的数字, 并且保证其为奇数, 之后辗转相除法 `gcd` 判断是否同  $\phi(n)$  互质, 则最后得到公钥  $e$

利用扩展欧几里算法, 计算  $e$  对于模数  $\phi(n)$  的逆元  $d$ , 即为私钥。

## 3. 加密和解密

加/解密的实质, 即将输入  $m$  自乘  $e$  次, 若使用快速模幂算法, 可以将复杂度降至  $O(\log e)$ 。

```
1      uint64_t pow(uint64_t a, uint64_t d, uint64_t p) {
2          // return a^n mod p
3          uint64_t res = 1;
4          while (d > 0) {
5              if (d & 1)
6                  res = mul(res, a, p);
7
8                  d = d >> 1;
9                  a = mul(a, p);
10         }
11         return res;
12     }
13     uint64_t RsaEncrypt(uint16_t m) {
14         return pow(m, e, n);
15     }
16     uint16_t RsaDecrypt(uint64_t c) {
17         return pow(c, d, n);
18     }
```

#### 4. RSA.h

RSA 整体提供的接口如下, 通过调用 *RsaInit()* 即可初始化 RSA 模块, 通过 *outState()* 打印 RSA 模块中 *p q N e d* 的值; 通过调用 *setRsaKey()* 设置私钥, 以供 Client 加密...

```
1 #pragma pack(4)
2 struct rsaTuple {
3     char sign[4];
4     uint64_t d;
5     uint64_t N;
6     rsaTuple(uint64_t d, uint64_t N) : d(d), N(N) {
7         sign[0] = 'R'; sign[1] = 'S';
8         sign[2] = 'A'; sign[3] = '\\0';
9     };
10 };
11 #pragma pack()
12
13 void RsaInit();
14
15 void setRsaKey(uint64_t d, uint64_t N);
16
17 void outState();
18
19 rsaTuple getKeyStruct();
20
21 uint64_t mul(uint64_t a, uint64_t b, uint64_t mod);
22
23 uint64_t mul(uint64_t a, uint64_t mod);
24
25 uint64_t pow(uint64_t a, uint64_t d, uint64_t p);
26
27 bool miller(int m, uint64_t d, uint64_t n);
28
29 bool isPrime(uint64_t n, int epoch);
30
31 uint64_t getRandom(uint8_t bits);
32
33 uint64_t gcd(uint64_t p, uint64_t q);
34
35 uint64_t euclid(uint64_t e, uint64_t t_n); // 扩展欧几里得
36
37 uint64_t RsaEncrypt(uint16_t m);
38
39 uint64_t* RsaEncrypt64(uint64_t m, uint64_t * rs);
40
41 uint16_t RsaDecrypt(uint64_t c);
42
43 uint64_t RsaDecrypt64(uint64_t* c);
44
```

```
45 #endif // HEADER_H
```

#### (四) RSA 加密 DES 密钥

在本次实验中，当 Client 连接到 Server 后，将收到 Server 发送的 RSA 私钥 $(d, N)$ ，之后 Client 将本次通话的 DES 密钥使用 $(d, N)$ 加密后，传输给 Server。

Client 端加密 DES 由于前面设计了 *RsaEncrypt()* 函数接受的是 16bit 的输入，因此需要将 64bit 的 DES\_key 分 **16bit 为一组**进行加密；而对应 Server 进行解密时也需要将这组加密的结果**分组解密**、拼接成原始的 DES\_key。

Client 分组加密 DES Key 代码如下：

- *desTuple*  
用于封装分组加密后的 DES 信息，前 4 个字节即为该消息签名。
- *RsaEncrypt64*  
将 64bit 的输入，每 16 位为一组调用 RsaEncrypt 函数加密，最后得到的四部分结果存入 rs 数组内
- *ThreadSendDesKey*: 发送 DES 消息。

```
1 #pragma pack(1)
2 struct desTuple {
3     char name[4];
4     uint64_t k[4];
5     desTuple(uint64_t* N){
6         name[0] = 'D';
7         name[1] = 'E';
8         name[2] = 'S';
9         name[3] = '\0';
10        memcpy(k, N, 32);
11    }
12 };
13 #pragma pack()
14
15 uint64_t* RsaEncrypt64(uint64_t m, uint64_t* rs) {
16     //64bit 内容分16bit为1组加密
17     //return rs[4]
18     uint16_t* tp = (uint16_t*) &m;
19     for (int i = 0; i < 4; i++, tp++)
20         rs[i] = RsaEncrypt(*tp) << (i * 16);
21
22     return rs;
23 }
24 void ThreadSendDesKey() {
25     uint64_t rs[4]{ 0 };
26     uint64_t des_key = 0x2140316;
27     RsaEncrypt64(des_key, rs);
28 }
```

```

29     desTuple desKeyTuple = desTuple(rs);
30     send(remote_sock, (char*)&desKeyTuple, 255, 0);
31     set_key(des_key);
32     cout << "Send DES key: " << hex << des_key << endl;
33 }

```

Server 端接受、解密 DESTuple 如下:

```

1
2 uint64_t RsaDecrypt64(uint64_t* c) {
3     uint64_t rs = 0;
4     for (int i = 0; i < 4; i++)
5         rs |= RsaDecrypt(c[i]) << (i * 16);
6
7     return rs;
8 }
9
10 uint64_t ThreadReciDesKey() {
11     //buf格式 DES:64 bit
12     //接受DES 密钥
13
14     uint64_t tp = 0;
15     auto st = chrono::steady_clock::now();
16     auto end = chrono::steady_clock::now();
17     auto duration = chrono::duration_cast<chrono::milliseconds>(end - st);
18
19     cout << "Accepting DES key ..." << endl;
20
21     while (1) {
22         char buf[255];
23         int len = recv(remote_sock, buf, 255, 0);
24         if (len && buf[0]=='D' && buf[1]=='E' && buf[2]=='S') {
25             //验证签名, 之后进行分组解密、拼接。
26             uint64_t* tmp = (uint64_t*)(buf + 4);
27             tp = RsaDecrypt64(tmp);
28             break;
29         }
30
31         end = chrono::steady_clock::now();
32         duration = chrono::duration_cast<chrono::milliseconds>(end - st);
33         if (duration.count() > 5000) {
34             // 设置超时时间, 若时间过长表面已不再安全。
35             cout << "Timeout" << endl;
36             break;
37         }
38     }
39
40     if (tp) {
41

```

```

42     cout << "Execution time: " << duration.count() << " ms" << std::endl;
43     cout<< "DES Key: " << hex << tp <<"\n";
44     return tp;
45 }
46 else{
47
48     cout<<"Accept DES key failed"<<endl;
49     return 0;
50 }
51 }

```

### (五) 基于 socket 的 tcp 传输模块

这一部分保持和上一次实验一致，几乎没有任何变化，它在整个工程中作用仅仅负责 ws232 库的初始化、错误处理以及 socket 相关变量的初始化。

*socket.h* 模块提供如下作用：

- *listen\_socket*: 作为服务器时的监听 socket
- *remote\_sock*: 通讯对方的 socket
- *FLag\_Recv*: 模块开启标志
- *StartServ(bool mode)*
  - mode == 0: Server 模式此模式下将开启 Server 模式，并且初始化 socket 环境和模块内 *listen\_socket* 等变量，开始监听端口
  - mode == 1: Client 模式此模式下将开启 Client 模式，初始化 socket 环境后 connect 目标 Server。
- *RecvFromRemote()*: 该函数将开启 *recv* 函数，将收到的 message 通过 *appendMess* 函数，将消息经过一定处理后放入全局消息队列。

#### Socket.h

```

1  #ifndef HEADER_SOCKEH
2  #define HEADER_SOCKEH
3  #include <thread>
4  #include<iostream>
5  #include<string>
6  #include<winsock2.h>
7  #include <WS2tcpip.h>
8  #pragma comment(lib, "ws2_32.lib")
9
10 extern bool FLag_Recv;
11 extern void appendMess(std::string s);
12
13 extern SOCKET listen_socket;
14

```

```

15 extern SOCKET remote_sock;
16
17 void StartServ(bool mode);
18
19 void RecvFromRemote(int id);
20
21 #endif // HEADER_SOCKEH

```

## (六) 消息模块和视图模块

这两部分也和上一次实验一致，消息模块用于存储用户之间的聊天内容、系统消息；视图模块提供了聊天界面和用户输入的地方

如下图进行总结：

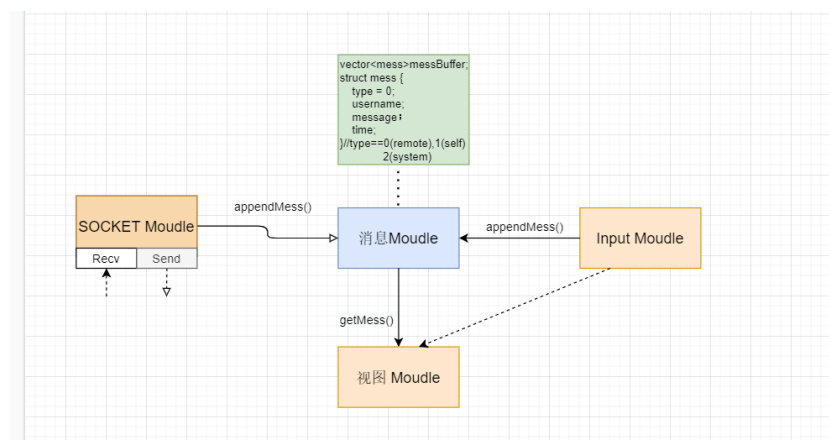


图 3: Mess

## (七) 控制模块

这一部分负责协调上面各个模块，将在 main 函数中体现：

**Server 部分：**

```

1 int Server_main() {
2     RsaInit();
3
4     outState();
5
6     StartServ(0);
7
8     // 发送私钥 (d, N)
9     rsaTuple rsaPrivateKey = getKeyStruct();
10    send(remote_sock, (char *) &rsaPrivateKey, 255, 0);
11
12    // 接受 Des key
13    uint64_t desKey = ThreadReciDesKey();
14    set_key(desKey);
15

```

```
16 cout << "prepare chat room....";
17 sleep_for(std::chrono::milliseconds(1000));
18
19 //开启各模块
20 system("cls");
21 thread t0 = std::thread(RecvFromRemote, 0);
22 thread t1 = std::thread(ThreadView, 1);
23 thread t2 = std::thread(ThreadInput, 2);
24
25 t0.join();
26 t1.join();
27 t2.join();
28
29 WSACleanup();
30 cout << "Exit\n";
```

### Client 部分:

```
1 int client_main(){
2     StartServ(1);
3
4     // 接受Rsa私钥 (d, N)
5     cout << "Accepting RSA key";
6     uint64_t rsa_e, rsa_N;
7     char buf[255];
8     while (1) {
9         int l = recv(remote_sock, buf, 255, 0);
10        if (l && buf[0] == 'R' && buf[1] == 'S') {
11            uint64_t* tmp = (uint64_t*)(buf + 4);
12            rsa_e = (*tmp);
13            rsa_N = (*(tmp+1));
14            break;
15        }
16
17    }
18    cout << "Get remote RSA public key: " << hex << rsa_e << ", " << hex <<
        rsa_N << endl;
19    setRsaKey(rsa_e, rsa_N);
20
21    // 发送DES key 格式: DES:64 bit
22    uint64_t rs[4]{ 0 };
23    uint64_t des_key = 0x2140316;
24    RsaEncrypt64(des_key, rs);
25
26    desTuple desKeyTuple = desTuple(rs);
27    send(remote_sock, (char*)&desKeyTuple, 255, 0);
28    cout << "Send DES key: " << hex << des_key << endl;
29
30    set_key(des_key);
```

31

...

NIKU



## 四、 实验结果

### (一) RSA-单元测试

测试代码:

```

1 //RSA 单元测试
2 RsaInit();
3 outState();
4 cout << "----- RSA TEST -----" << endl;
5
6 int input = 0x00001314;
7 cout << "\u001b[31m\u001b[1minput:" << hex << input <<endl << "\u001b[0m
8 ";
9
10 cout<<"\u001b[32m\u001b[1mRSA Encrypt Result:" << hex << RsaEncrypt(input
11 )<<endl << "\u001b[0m" ;
12
13 cout << "\u001b[31m\u001b[1mRSA Decrypt Result:" << hex << RsaDecrypt(
14 RsaEncrypt(input)) << "\u001b[0m" ;

```

可以看到红色部分分别对应加密前和解密后的结果，是正确的。

```

Microsoft Visual Studio 调试控制台
GET RANDOM NUMBER: 86627
GET RANDOM NUMBER: 37871
p: 0000000000015263
q: 00000000000093ef
n: 00000000c38ac36d
PUBLIC KEY: 000000000000245b
PRIVATE KEY: 000000003e80318f
N oc: 3280651117
PRIVATE KEY_oc: 1048588687
----- RSA TEST -----
input:1314
RSA Encrypt Result:9c43a68d
RSA Decrypt Result:1314
G:\A大一下\C++\DESS\64\Debug\DESS.exe (进程 25276)已退出，代码为 0。
按任意键关闭此窗口。 . .

```

图 4: Enter Caption

### (二) RSA 分组加密单元测试

测试代码:

```

1 void RSA_slice_test(){
2 RsaInit();
3 outState();
4 cout << "----- RSA Slice TEST -----" << endl;
5
6 uint64_t DES_key = generate_random_key();
7 cout << "\u001b[31m\u001b[1minput: " << hex << DES_key << endl << "\u001b[0m
8 [0m";
9
10 uint64_t * rs = new uint64_t[4];
11 RsaEncrypt64(DES_key, rs);

```

```

11
12     cout << "\u001b[32m\u001b[1mRSA Encrypt64 Result:" << "\u001b[0m"<<endl;
13     for (int i=0;i<4;++i)
14         cout << "    Slice[" << i + 1 << "]" << hex << rs[i] << endl;
15     cout << endl;
16
17     uint64_t* tmp = (uint64_t*)(rs);
18     cout << "\u001b[31m\u001b[1mRSA Decrypt64 Result: " << hex <<
19         RsaDecrypt64(tmp) << "\u001b[0m";

```

可以看到红色部分分别对应加密前和解密后的结果，是正确的。

```

G:\A大三下\C++\DESS\y64\Debug\DESS.exe
GET RANDOM NUMBER: 33721
GET RANDOM NUMBER: 96907
p: 00000000000083b9
q: 0000000000017a8b
N: 00000000c2c6af73
PUBLIC KEY: 000000000003227
PRIVATE KEY: 0000000009116e57
N_oc: 3267800947
PRIVATE KEY_oc: 2434145879
----- RSA Slice TEST -----
input: c5e889446e9cb65b
RSA Encrypt64 Result:
  Slice[1]60a06785
  Slice[2]b979c2a1
  Slice[3]3770cfd8
  Slice[4]80a2a879
RSA Decrypt64 Result: c5e889446e9cb65b

```

图 5: RSA Slice TEST

### (三) 完整测试

启动 Server 和 Client，查看输出日志并且可以正常通讯。

**建立 TCP 连接、双方交换 RSA 公钥和 DES 密钥**

```

G:\A大三下\C++\DESS\y64\Debug\DESS.exe
GET RANDOM NUMBER: 33739
GET RANDOM NUMBER: 51517
p: 00000000000083cb
q: 000000000000c93d
N: 000000006799ca5f
PUBLIC KEY: 00000000000083d4
PRIVATE KEY: 000000005f16f875
N_oc: 1738132063
PRIVATE KEY_oc: 1595340917
StartServer listen...
StartServer connectedAccepting DES key...
Execution time: 0 ms
DES Key: 2140316
SET KEY AS: 0x2140316prepare chat room....

G:\A大三下\C++\DESS\y64\Debug\DESS.exe
StartClient connectedAccepting RSA keyGet remote RSA public key: 3227, 1ca419ed
Read DES key: 2140316
SET KEY AS: 0x2140316, 0 10008200100
1 800160100
2 200002140
3 240000a00
4 4004000040a
5 4000000100a
6 8002045060
7 20000700860
8 1200484018
9 104811008
a 4040891220
b 400100024
c 101100890
d 10001012011
e 10080202000
f 8120203
prepare chat room....

```

图 6: 双方交换 RSA 公钥和 DES 密钥，Server(左) Client(右)

聊天过程: 后续部分实则和上一次实验一致了。

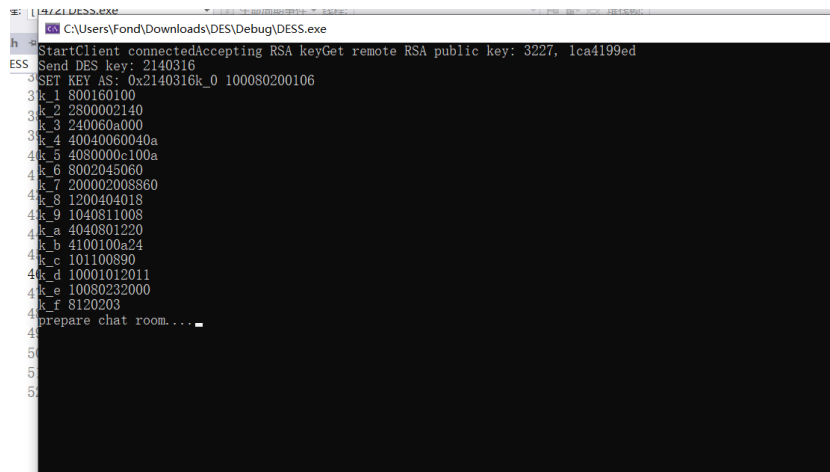
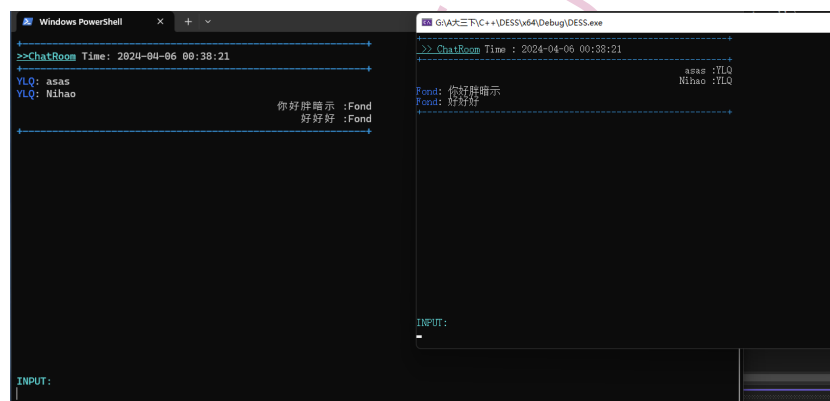


图 7: 聊天过程

在本地 127.0.0.1 展示这个版本的结果是我提交的 exe 文件



## 五、 实验遇到的问题及其解决方法

### (一) Rsa 分组加密、传输

#### 问题介绍:

我提供的原始 RSA 加密 API 的输入是 16bit 的数字, 但加密的内容是 64 bit, 因此需要将这 64bit 进行拆分或者其他方案解决。

#### 解决方案:

正如上文写到的, 在对 64 bit 长度的 DES 密钥加密时, 每次取其 16bit 进行加密, 最后将加密的 4 部分结果以数组形式、并且假如签名传输给对方; 对方验证签名内容时 DES 密钥消息, 且依次取数组内四部分内容解密、拼接即可。000

### (二) 64bit 随机数产生

#### 问题介绍:

由于这次实验时需要生成 64bit 的大素数, 且为了更安全的加密通讯, 因此随机数的安全也是不可或缺的, 而以往我使用的 random 库中的 rand 函数仅能生成 32 位的随机数、且根据随

机数发生器生成的伪随机数是不够安全的。

#### 解决方案：

c++ 11 提供了更接近真实随机的方案— `std::random_device` 据这篇中提到的：

“`std::random_device` 通过它是相当不透明的。你并不真正知道它是什么、它要做什么、或者它的效率如何”

但同样为代价的是，通过这种方式产生随机数效率将变低。并且，可以直接通过 `std::uniform_int_distribution<int> dist{1, max};` 指定任意大小的随机数字，64 位也是可以的。

### (三) 编程问题

#### 1. 结构体对齐

在传输 DES key 和 Rsa key 时，我均想通过构造简单协议，将协议使用结构体存储的方式，但是这里需要注意编译器会对结构体默认采取对齐，结构体的对齐是为了保证结构体成员在内存中的存储位置是按照预期的方式进行的，从而提高内存访问的效率。但同时也会导致直接通过指针偏移寻找成员变量可能有错。

比如这是我一开始的 DesTuple:

```
struct desTuple{
    char name[3];
    uint64_t k[4];
    desTuple(uint64_t* N) {
        name[0] = 'D';
        name[1] = 'E';
        name[2] = 'S';
        memcpy(k, N, 32);}
};
```

理论上这个结构体大小应该为  $3+4*8 = 35$  BYTE，但实际上这个结构体大小为 36BYTE。这是因为我的 vs 中，Struct 默认为 4 字节对齐

如下图，name[2] 和 k[0] 的地址并不相邻，而是中间空了 1BYTE，也就导致这个结构体比理论的 35 字节多了 1。

```
uint64_t * rs = new uint64_t[4];
RsaEncrypt64(DES_key, rs);
desTuple d(rs);

cout << "desTuple.name[2] addr: " << &d.name[2] << endl;
cout << "desTuple.k[0] addr: " << &d.k[0] << endl;
// cout << "&k[0]-&name[2]: " << &d.k[0] - &d.name[2] << endl;
```

```
GET RANDOM NUMBER: 75337
GET RANDOM NUMBER: 71999
p: 0000000000012649
q: 000000000001193f
N: 00000001434e8cf7
PUBLIC KEY: 0000000000003227
PRIVATE KEY: 000000002571ad87
N oc: 5424182663
PRIVATE KEY oc: 628206983
----- RSA Slice TEST -----
desTuple.name[2] addr: 000000FC0972F68E
desTuple.k[0] addr: 000000FC0972F68C
```

从而导致一开始我直接通过一个指向 DesTuple 的 uchar\* 类型指针 b，通过手调整偏移量  $b + 3$  获得 DesTuple.k 数组的地址时发生错误。

**解决方案：**可以通过精心设计结构体变量的排列，让每一个变量、数组大小都是 4 字节的倍数，让其中间不要有空缺；另外，通过设置 `#pragma pack(1)` 设置结构体对齐方式不要按照默认的 4 字节，而是 1 字节

```
#pragma pack(1)
```

```
struct desTuple{  
    char name[3];  
    uint64_t k[4];  
}  
#pragma pack()
```

## 2. 设置随机数种子，但是不随机

有很多时候我在使用 `std::uniform_int_distribution<uint64_t>` 生成随机数时，总是相同的结果，但是我也在函数开始时使用 `std::mt19937_64 rng(time(NULL));` 设置了种子。

### 解决方案：

仅仅在全局的一个地方设置随机数种子，而不是每一次在产生随机数的函数前进行设置。

## 六、 实验结论

这次实验最重要的部分是理解 RSA 加密的原理，考虑到各个算法细节和可以改进的地方，并且在上一次 DES 加密的 TCP 信息通讯的基础上，实现对 DES 密钥的加密传输，进一步提升通讯的安全。

让我收获比较大的，让我感受到了 RSA 算法这种非对称加密和上一次实验中 DES 加密的区别，在没有认真看实验手册前，我机器的按照网上 RSA 算法的流程进行实验编写，然后完全替代上一次实验中 DES 的作用，即每一次传输的消息我都经过 RSA 分组加密（此时我的 RSA 模块的输入为 32 位，输出为 64 位，但算法中间要时刻防止溢出，当然后面参照实验手册简化为 16 位-64 位），最后发现在通信过程中有明显的延迟。因为 RSA 一次加密的复杂度是密钥长度的平方成正比，而 DES 只是常数级别。

另外由于要在上一次实验的基础上做，也就需要上一次的代码有一定的可扩展性，因此我依照 MVC 样式进行重构了很多部分，收获也很大。