网 络 安 全 技 术

实 验 报 告

学　　院：网络空间安全学院
年　　级：2021 级
专　　业：信息安全、法学双学位
学　　号：2113203
姓　　名：付政烨
手 机 号：15719340667
完成日期：2024 年 4 月 21 日

目　录

一、 实验目的

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“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. 要求程序实现双全工通信，并且加密过程对用户完全透明。

三、 实验步骤及实验结果

(一) 实验架构

1. 安全加密模型

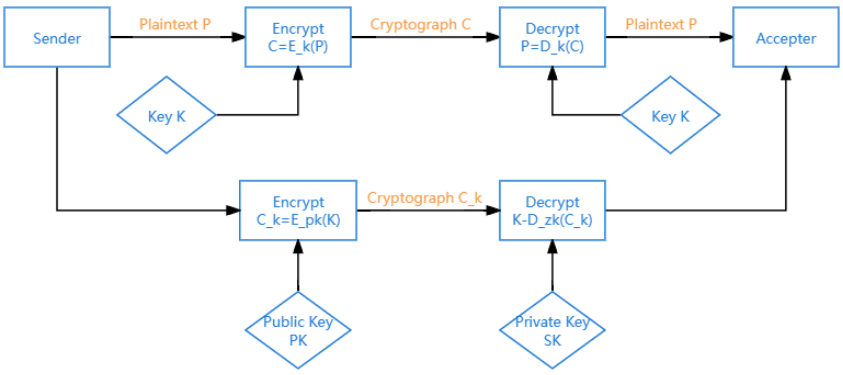

图 3.1 对称密码通信

在数字通信中，为确保信息传输的安全性，常采用对称加密和非对称加密相结合的方式进行加密通信。在本模型中，使用的对称加密算法为 DES（Data Encryption Standard），其特点是加密和解密使用相同的密钥。由于对称加密算法的密钥分发问题，需要借助非对称加密算法来安全地传输对称加密的密钥。

非对称加密算法中，RSA（Rivest-Shamir-Adleman）算法是一种广泛使用的方法。它基于一个重要的原则，即每个参与者都拥有一对密钥：公钥和私钥。公钥是公开信息，可被任何人使用来加密信息；私钥则是保密的，仅用于解密信息。

在本加密通信模型中，发送方（发方）首先生成一个 DES 密钥用于对称加密。为了安全地将这个密钥传递给接收方（收方），发方使用收方的公钥对 DES 密钥进行 RSA 加密。这样，即使在传输过程中被截获，由于没有私钥，第三方也无法解密得到原始的 DES 密钥。接收方收到经过 RSA 加密的 DES 密钥后，可利用自己的私钥进行解密，从而安全获得 DES 密钥。之后，双方就可以使用这个 DES 密钥来进行后续的对称加密通信，保证信息传输的机密性和完整性。

（二） RSA 算法部分

1. 原理简介

(1) 算法步骤

RSA 算法的实现步骤如下：

1. 选择两个大质数 p 和 q。
2. 计算它们的乘积 $n = p \times q$，n 作为公钥和私钥的一部分。
3. 计算欧拉函数 $\phi(n) = (p-1)(q-1)$。
4. 选择一个整数 e，使得 e 与 $\phi(n)$ 互质，并且 $1 < e < \phi(n)$。
5. 计算 e 相对于 $\phi(n)$ 的模逆元 d，满足 $ed \equiv 1 \pmod{\phi(n)}$。

(2) 加密与解密

- **加密**：如果消息为 m，加密后的消息 c 计算为 $c = m^e \mod n$。
- **解密**：如果加密后的消息为 c，解密后的消息 m 计算为 $m = c^d \mod n$。

2. 代码实现

函数名称	功能描述
NewRsaKeyPair	服务器端生成 RSA 公私钥对，存储于 RSAKeyPair 结构中。
ClientEncry	客户端使用服务器的公钥加密 DES 秘钥，并返回加密后的字符串。
ServerDecry	服务器使用私钥解密客户端发送来的 DES 秘钥，并返回解密后的字符串。
PowMod	计算并返回 ($a^q \mod n$) 的结果，用于加密和解密操作中的模幂运算。
RandomPrime	生成一个随机质数，用于 RSA 公私钥对的生成中。
Euler	返回小于 n 且与 n 互质的正整数的个数，用于 RSA 算法中求解欧拉函数 $\phi(n)$ 的值。
RabinMillerKnl	利用 Rabin-Miller 原理检测一个数是否为质数，为 RabinMiller 函数提供核心判断逻辑。
RabinMiller	多次调用 RabinMillerKnl 来提高判断质数的准确性，用于生成高质量的随机质数。
GreatestCommonDivisor	判断两个数是否互质，用于 RSA 算法中密钥生成阶段以确保选定的密钥满足互质条件。

表 3.1 RSA 算法中的关键函数及其功能

在本次实验的 RSA 加密算法的实现中，CRsaOperate 类是核心，包含了生成密钥对、加密和解密数据的功能。类的方法 NewRsaKeyPair 负责生成 RSA 公私钥对，这一过程利用 RandomPrime 函数来生成随机质数，并使用 Euler 函数来计算欧拉函数值，确保密钥的有效性。ClientEncry 和 ServerDecry 方法分别处理客户端的加密需求和服务器端的解密需求，其中加密和解密都依赖于 PowMod 函数来执行模幂运算。此外，RabinMiller 函数通过多次调用 RabinMillerKnl 来提高

质数判断的准确率，而 GreatestCommonDivisor 用于检查生成的公私钥是否互质，确保 RSA 算法的数学要求得到满足（详见表表 3.1 ）。

(1) 快速幂算法

PowMod 函数实现了模幂运算，即计算 $(base^{pow})\%n$ 的结果。这是一种在密码学和数值计算中非常常用的操作，尤其是在大数运算时，直接计算可能导致非常大的中间结果，因此采用了高效的算法——快速幂算法（也称为" 指数平方法"）。

```
int64 CRsaOperate::PowMod(int64 base, int64 pow, int64 n) {
    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;
}
```

函数使用两层循环来逐步计算最终的结果：

外层循环检查 b 是否为 0。只要 b 不为 0，就继续执行内层循环。这个循环的作用是处理每一位二进制表示中的 1，直到 b 为 0 为止。内层循环首先检查 b 的当前最低位是否为 1：

- 如果不为 1，即当前位为 0，则通过 b »= 1（即 b 右移一位）来移除这一位。这时，我们需要对 a 进行平方（$a = MulMod(a, a, n)$），因为二进制中每右移一位，代表的是将幂乘以 2，所以基数也需要平方来匹配幂的增加。
- 如果为 1，我们减少 b（即 b–），使得最低位变为 0（因为从二进制的角度来看，减 1 操作会将最低位的 1 变为 0）。此时，将 a 乘以当前累计的结果 c，并取模（$c = MulMod(a, c, n)$），因为这一位贡献了其权重（即当前 a 的

值)。

当 b 减为 0 时，外层循环结束，此时 c 中存储的即为 $(base^{pow})\%n$ 的结果。

(2)Rabin-Miller 质数测试：RabinMillerKnl 和 RabinMiller 函数

Rabin-Miller 质数测试是一种概率性的质数判定方法。它基于数论中的几个重要理论，主要是费马小定理和二次探测。该测试对于大数的质数测试特别有效，因为它比传统的试除法更快。

```
bool CRsaOperate::RabinMillerKnl(int64 n) {
    int64 q = n - 1, k = 0;
    while (!(q & 1)) {
        ++k;
        q >>= 1;
    }
    int64 a = 2 + rand() % (n - 2); // 随机数a满足 2≤a < n - 1
    if (PowMod(a, q, n) == 1) {
        return true;
    }
    for (int64 j = 0; j < k; j++) {
        int64 z = 1 << j; // 等价于2^j
        if (PowMod(a, z * q, n) == n - 1) {
            return true;
        }
    }
    return false;
}
```

函数首先将 n-1 分解为 $2^k * q$ 的形式，其中 q 是奇数。这一步是通过不断右移 n-1（除以 2）实现的，同时计数移位的次数 k。选择一个随机数 a，使其在 [2, n-2] 范围内。这个 a 被用作后续的模幂运算的基数。使用 PowMod 函数计算 $a^q\%n$。如果结果为 1，则 n 可能是质数，函数返回 true。对于 j 从 0 到 k-1，计算 $a^{(2^j * q)}\%n$。如果其中任何一个结果等于 n-1，则 n 可能是质数，函数返回 true。如果以上检验均未能证明 n 可能是质数，则函数返回 false，即 n 不是质数。

```
bool CRsaOperate::RabinMiller(int64 n, int time) {
    for (int i = 0; i < time; ++i) {
        if (!RabinMillerKnl(n)) {
            return false;
        }
    }
    return true;
}
```

该函数通过多次调用 RabinMillerKnl 来增强质数判断的准确性。参数 time 指定了测试的轮数，每一轮都会重新选择随机数 a 进行测试。如果在任何一轮中，RabinMillerKnl 返回 false（即 n 不是质数的概率较高），则整个函数返回 false。只有当所有测试轮次都认为 n 可能是质数时，才最终返回 true。

(3) 质数生成：RandomPrime 函数

```
int64 CRsaOperate::RandomPrime(char bits) {
    ...
    do {
        base = (unsigned long)1 << (bits - 1); //保证最高位是 1
        base += rand() % base; //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)); //测试 30 次
    return base; //全部通过认为是质数
}
```

这个函数生成指定位数的随机质数。它通过位操作保证生成的数是奇数，并且最高位为 1。然后通过 Rabin-Miller 测试多次确认其为质数。

(4) 互质判断和欧拉函数计算：GreatestCommonDivisor 和 Euler 函数

GreatestCommonDivisor 函数使用了著名的辗转相除法（也称欧几里得算法）来判断两个数 p 和 q 是否互质。辗转相除法的基本原理是：两个正整数的最大公约数等于其中较小的数和两数相除余数的最大公约数。

```
bool CRsaOperate::GreatestCommonDivisor(int64 p, int64 q) {
    int64 a = p > q ? p : q;
    int64 b = p < q ? p : q;
    int t;
    if (p == q) {
        return false;
    }
    else {
        while (b) {
            a = a % b;
            t = a;
            a = b;
            b = t;
        }
        if (a == 1) {
            return true;
        }
        else {
            return false;
        }
    }
}
```

具体步骤如下：

- 将输入的两个数按大小顺序分别赋值给 a 和 b。
- 使用循环进行辗转相除，每次循环中用较大数 a 除以较小数 b，然后用 b 替换 a，用余数替换 b，直到 b 变为 0。
- 当 b 为 0 时，a 的值即为 p 和 q 的最大公约数。如果 a 为 1，则说明两数互质，函数返回 true；否则返回 false。

欧拉函数 φ(n) 表示的是小于或等于 n 的正整数中与 n 互质的数的数目。此函数的计算方法依赖于 n 的质因数分解：

```
int64 CRsaOperate::Euler(int64 n) {
```

```
    int64 res = n, a = n;
    for (int64 i = 2; i * i <= a; ++i) {
        if (a % i == 0) {
            res = res / i * (i - 1); //先进行除法是为了防止中间数据的溢出
            while (a % i == 0) {
                a /= i;
            }
        }
    }
    if (a > 1) {
        res = res / a * (a - 1);
    }
    return res;
}
```

首先，将 n 的初始值赋给变量 res 和 a。使用一个循环来检查每一个小于或等于 sqrt(a) 的数 i，查看 i 是否是 a 的因子。如果 i 是 a 的因子，就通过以下公式更新 res 的值：$res = res/i * (i - 1)$。这个公式基于欧拉函数的性质：如果 $n = p1^k1 * p2^k2 * ... * pm^kmφ(n) = n * (1 - 1/p1) * (1 - 1/p2) * ... * (1 - 1/pm)$。然后，去除 a 中所有的 i 因子，直到 a 不能再被 i 整除。在循环结束后，如果 a 仍大于 1，说明 a 本身是一个质数，我们需要对 res 进行最后一次更新。这样计算的结果 res 就是小于或等于 n 且与 n 互质的整数的数量。

(5)RSA 公私钥对生成：NewRsaKeyPair 函数

首先调用 RandomPrime 函数生成两个 16 位的随机质数 primeP 和 primeQ。这两个质数是构建 RSA 加密安全性的基础。在 RSA 算法中，选择的质数需要足够大，以确保密钥的安全性。

```
int64 primeP = RandomPrime(16);
int64 primeQ = RandomPrime(16);
```

接下来，函数计算 n，即两个质数的乘积。n 作为公钥和私钥的一部分。然后计算欧拉函数值 euler，即小于 n 且与 n 互质的正整数的数量。欧拉函数值是

选择公钥 e 和计算私钥 d 的重要参考。

```
int64 n = primeP * primeQ;
int euler = Euler(n);
```

选择一个与 euler 互质的 e 作为公钥的一部分。这里通过 GreatestCommonDivisor 函数验证 e 是否与 euler 互质。如果互质，则循环结束。选择的 e 通常是一个小于 65536 的随机数，确保了一定的安全性。

```
int64 e;
while(1) {
    e = rand() % 65536 + 1;
    // e = rand() % (euler - 1) + 1;
    if (GreatestCommonDivisor(e, euler)) {
        break;
    }
}
```

私钥 d 的计算较为复杂，需要满足 $(d * e)\%euler = 1$。这一步通过遍历寻找符合条件的 d。循环遍历每一个 i，计算表达式 $((i * euler) + 1)\%e$，直到找到一个使表达式为 0 的 i，从而求得 d。如果超过了预设的最大尝试值 max，循环将终止。

```
int64 max = 0 xffffffffffffffff - euler;
int64 i = 1, d = 0;
while(1) {
    if ( ((i * euler) + 1) % e == 0) {
        d = ((i * euler) + 1) / e;
        break;
    }
    i++;
    int64 temp = (i + 1) * euler;
    if (temp > max){
        break;
    }
}
```

最后，如果成功找到满足条件的 d，则将公钥和私钥存入 keyPair 结构，并返回成功。如果在预设范围内未找到合适的 d，则返回错误。通过上述步骤，NewRsaKeyPair 函数能够生成一对有效的 RSA 密钥，用于加密和解密操作，保证通信的安全性。

```
if (d == 0) {
    return RSA_KEY_PAIR_ERR;
}

keyPair.publicKey_e = e;
keyPair.secretKey_d = d;
keyPair.n = n;
keyPair.en = to_string(e) + "," + to_string(n);

return SUCCESS;
```

(6) 客户端加密：ClientEncry 函数

i. 转换 DES 密钥

函数首先将字符串格式的 DES 密钥转换为一个 64 位的整数。这个步骤是通过以下代码实现的：

```
int64 int64DesKey = 0;
for (unsigned int i = 0; i < DesKey.length(); ++i) {
    int64DesKey += DesKey[i];
    if (i != DesKey.length() - 1) {
        int64DesKey <<= 8;
    }
}
```

每个字符（DesKey[i]）首先被转换为其对应的 ASCII 数值。对于每个字符（除了最后一个字符），当前的整数值（int64DesKey）会向左移动 8 位。这样做是为了为下一个字符的 ASCII 值腾出空间，确保所有字符能顺利拼接成一个连续的 64

位整数。这个循环完成后，int64DesKey 包含了 DES 密钥的完整数值表示，其中每个字符占用了 8 位。

ii. 分割密钥为四个部分

DES 密钥现在是一个 64 位的整数，接下来这个整数被分割为四个 16 位的部分，代码如下：

```
unsigned short* pRes = (unsigned short*)&int64DesKey;
unsigned short M[4];
for (int i = 0; i < 4; ++i) {
    M[i] = pRes[i];
}
```

将 int64DesKey 的地址转换为 unsigned short* 类型，这样就可以将这个 64 位整数视为一个 16 位数值的数组。通过循环，将这四个 16 位部分复制到数组 M 中。这样，M[0] 至 M[3] 每一个都包含了原始密钥的一部分。

iii. 对每个部分执行 RSA 加密

每个 16 位的部分使用服务器的公钥进行 RSA 加密：

```
string result;
for (int i = 3; i >= 0; --i) {
    string temp = to_string(PowMod(M[i], publicKey_e, n));
    result += temp;
    result += ',';
}
```

使用 PowMod 函数对每个 16 位部分进行加密。PowMod 函数实现了模幂运算 $(M^e) \mod n$，其中 M 是待加密的消息部分，e 是公钥的指数部分，n 是模数。每个加密后的数字转换成字符串，并添加逗号作为分隔符，拼接成最终的结果字符串。

iv. 返回加密结果

函数返回一个字符串，包含了四个用逗号分隔的加密数字。这个字符串可以被发送到服务器，服务器可以使用相应的私钥进行解密。通过以上步骤，客户端成功地使用公钥将 DES 密钥加密为一个可安全传输的格式，确保了数据在传输

过程中的安全性。

(7) 服务器解密：ServerDecry 函数

i. 解析加密数据

服务器首先需要处理接收到的加密数据，该数据是一串以逗号分隔的加密数字字符串。解析这些数据的代码如下：

```
string DesKey = "";
int pos = 0;

for (int i = 0; i < 4; i++) {
    string tempStr = "";
    for (; keyInfo[pos] != ','; ++pos) {
        tempStr += keyInfo[pos];
    }
    ++pos;
```

通过循环迭代处理 keyInfo 字符串，使用 pos 变量来跟踪当前的索引位置。对于每个逗号之前的字符序列，它们被认为是一个加密的数值，并被逐一提取出来。：通过检测逗号来确定每个加密数值的结束点，并将每个数值作为一个单独的字符串 tempStr 提取出来，为后续的解密操作做准备。

ii. 对每个部分执行 RSA 解密

提取出加密数值后，服务器将使用其私钥对这些数值进行解密：

```
int64 Ci = fromStrToInt64(tempStr);
int64 nRes = PowMod(Ci, secretKey_d, n);
unsigned short * pRes = (unsigned short*)&nRes;
```

使用 fromStrToInt64 函数将提取的字符串 tempStr 转换为 64 位整数 Ci，这个整数代表加密后的数据。使用 PowMod 函数进行 RSA 解密，计算表达式 $(C^d) \mod n$，其中 C 是提取的加密数据，d 是私钥的指数部分，n 是相同的模数。解密结果 nRes 是一个 64 位整数，通过转换为 unsigned short* 来访问其 16 位段。

iii. 检查解密结果并组装密钥

解密后需要检查结果是否正确，并组装原始的 DES 密钥：

```
if (pRes[1] != 0 || pRes[2] != 0 || pRes[3] != 0) { // error
    perror ("sever␣ServerDecry()␣err");
    return 0;
} else {
    DesKey += fromShortToString(pRes[0]);
}
```

检查解密结果的高位部分（即 pRes[1], pRes[2], pRes[3]）是否为 0，这是确保解密结果是有效的 16 位数据的重要步骤。如果这些部分不为 0，说明解密过程出现错误，函数将报错并终止。如果没有错误，将 16 位的 pRes[0] 转换为对应的两个字符并添加到 DesKey 字符串中。这一步骤是通过 fromShortToString 函数实现的，确保正确地重构出原始的 DES 密钥。

iv. 返回解密后的 DES 密钥

函数返回一个字符串，包含了解密并重构后的原始 DES 密钥，这个密钥现在可以用于加密和解密数据通信中的信息。由此，服务器成功地使用私钥解密了客户端发送的加密 DES 密钥，并且确保了整个过程的正确性和安全性。

（三） 聊天程序部分

1. 客户端实现原理

(1) 初始化和连接

客户端启动时，首先创建一个 TCP socket，并通过 connect 函数连接到服务器地址。服务器地址可以是用户输入的 IP 地址，或者默认的服务器地址。连接成功后，客户端与服务器之间的通信通道建立。

(2) 密钥协商

在数据交换之前，客户端与服务器进行密钥协商。客户端首先接收服务器发送的 RSA 公钥。然后使用此公钥加密一个随机生成的 DES 密钥，并将加密后的密钥发送给服务器。此步骤确保双方建立一个安全的 DES 密钥，用于后续的加密通信。

```
int KeyAgreement(sockaddr_in serverAddr, int fd_skt) {
    CRsaOperate rsa;
    CDesOperate des;
    DES_KEY = CDesOperate::GenerateDesKey(); // 生成随机DES密钥
    string strSMsg;
    if (recvMsgFromServer(serverAddr, fd_skt, KEY_AGREE_SEND_KEY, &strSMsg) !=
        SUCCESS) { // 接收服务器的公钥对
        perror("client KeyAgreement()-recvMsgFromServer()-1 err");
        return CLIENT_KEY_AGRRE_ERR;
    }
    int pos = strSMsg.find(",", 0);
    int64 e = fromStrToInt64(strSMsg.substr(0, pos));
    int64 n = fromStrToInt64(strSMsg.substr(pos + 1, strSMsg.size()));
    string encryDesKey = rsa.ClientEncry(DES_KEY, e, n); // 使用公钥加密DES密钥

    if (sendMsgToServer(serverAddr, fd_skt, KEY_AGREE_SEND_KEY, encryDesKey) !=
        SUCCESS) { // 给服务器发送加密过的DES秘钥
        perror("client KeyAgreement() err");
        return CLIENT_KEY_AGRRE_ERR;
    }

    if (recvMsgFromServer(serverAddr, fd_skt, KEY_AGREE_CONFIRM) != SUCCESS) {
        // 接收服务器的确认DES接收成功
        perror("client KeyAgreement()-recvMsgFromServer()-2 err");
        return CLIENT_KEY_AGRRE_ERR;
    }

    return SUCCESS;
}
```

客户端密钥协商代码主要位于 KeyAgreement 函数中。该函数的作用是生成随机 DES 密钥，接收服务器的公钥，使用公钥加密 DES 密钥，并发送加密后的 DES 密钥给服务器。最后，它接收服务器确认 DES 密钥已正确接收的消息。

(3) 数据交换

密钥协商成功后，客户端使用 DES 加密数据，并发送加密后的数据给服务器。客户端使用 epoll 监听来自服务器的消息和标准输入的消息。如果是从服务器接收的消息，则进行解密并显示；如果是用户的输入，则加密后发送给服务器。

sendMsgToServer 函数在用户输入数据后使用 DES 加密数据，并发送加密后的数据到服务器。

```
int sendMsgToServer(sockaddr_in serverAddr, int fd_skt, int KeyAgreement, string msg) {
    char cMsg[MSG_SIZE];
    memset(cMsg, 0, sizeof(cMsg));

    if (DES_KEY == "") {
        perror("client miss DES_KEY");
        return MISS_DES_KEY;
    }
    string encryResult; // 加密结果
    CDesOperate des;
    if (des.Encry(msg.c_str(), DES_KEY, encryResult) != 0) { // 加密
        perror("client sendMsgToServer() err");
        return DES_ENCRY_ERR;
    }

    encryResult = ":" + encryResult;
    encryResult = MSG_HEAD_DATA + encryResult; // 加密完毕后添加头部信息

    memset(cMsg, '\0', MSG_SIZE);
    for (int i = 0; i < encryResult.length(); i++) { // 加密结果string转char[]
        cMsg[i] = encryResult[i];
    }
    cMsg[encryResult.size()] = '\0';

    if (send(fd_skt, cMsg, strlen(cMsg), 0) < 0) { // send，客户端向服务端发消息
```

```
        perror("client␣sendMsgToServer()␣-␣send()␣err");
        return CLIENT_SEND_ERR;
    }

    return SUCCESS;
}
```

recvMsgFromServer 函数接收服务器发来的加密数据，然后使用 DES 密钥进行解密，并显示解密后的消息。

```
int recvMsgFromServer(sockaddr_in serverAddr, int fd_skt, int KeyAgreement, string *
    strSMsg) {
    char sMsg[MSG_SIZE];
    memset(sMsg, 0, sizeof(sMsg));
    int sLen = recv(fd_skt, sMsg, sizeof(sMsg), 0); // recv，接收服务器发来的消息
    if(sLen <= 0) {
        perror("client␣recvMsgFromServer()␣-␣recv()␣err");
        return CLIENT_RECV_ERR;
    }
    sMsg[sLen] = '\0';

    string decryResult = "";
    CDesOperate des;
    if (des.Decry(sMsg, DES_KEY, decryResult) != 0) { //解密
        perror("client␣recvMsgFromServer()␣err");
        return DES_DECRY_ERR;
    }

    cout << "Receive␣message␣from␣<" << inet_ntoa(serverAddr.sin_addr) << ">:␣" <<
        decryResult << endl;
    return SUCCESS;
}
```

(4) 关闭连接

客户端可以通过发送特定命令（如”quit”）来关闭连接。收到此命令后，客户端关闭 socket 并终止程序。

2. 服务器实现原理（实现详见源码）

1. **初始化和等待连接**：服务器在启动时创建一个 TCP socket 并绑定到一个端口上，然后开始监听连接请求。使用 epoll 作为 IO 多路复用的解决方案，可以高效地处理多个连接。当有新的客户端连接请求时，服务器接受连接并创建一个新的 socket 用于与该客户端的通信。
2. **密钥协商**：对于每个新的客户端连接，服务器生成一对 RSA 公钥和私钥，发送公钥给客户端。接收到客户端用公钥加密的 DES 密钥后，使用私钥进行解密，获得用于后续加密通信的 DES 密钥。然后向客户端发送确认消息，表明密钥协商成功。
3. **数据交换**：密钥协商完成后，服务器进入数据交换阶段。服务器使用 epoll 监听来自客户端的数据和标准输入。接收到来自客户端的加密数据后，使用 DES 密钥进行解密并处理。如果是服务器的标准输入，则加密后发送给所有客户端。
4. **关闭连接**：服务器同样可以接收特定命令来关闭与客户端的连接或终止服务器程序。

（四） 实验结果

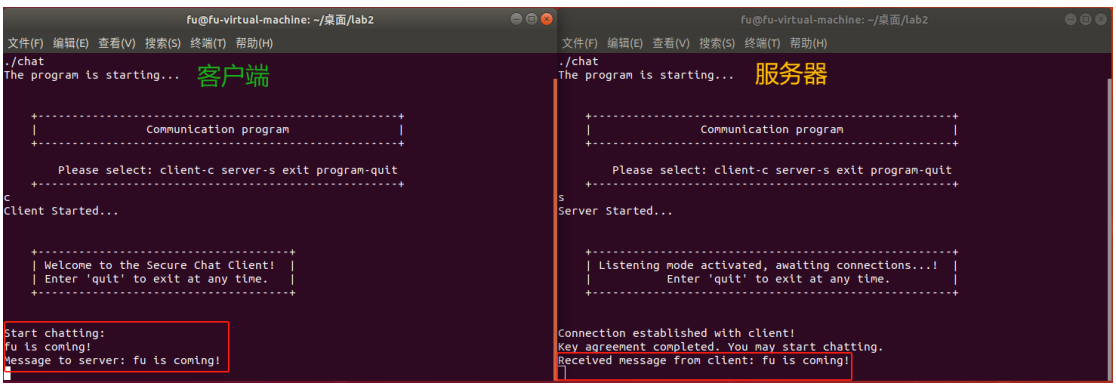

图 3.2

四、 实验遇到的问题及其解决方法

1. **RSA 密钥生成的效率问题**：在生成 RSA 密钥对的过程中，我们需要找到大的质数。最初，我们使用了简单的试除法来检测质数，但这种方法在面对大数时效率极低，严重影响了密钥生成的速度。为了解决这个问题，我们改用了 Rabin-Miller 质性测试算法。Rabin-Miller 算法是一个概率性算法，相比试除法，它在保证一定准确性的同时大大提高了检测大质数的速度。此外，通过多次测试来提高结果的可靠性，确保了密钥的强度和安全性。
2. **密钥协商的安全问题**：在客户端和服务器之间的密钥协商过程中，我们最初直接通过网络发送 DES 密钥，这样做存在明显的安全隐患，因为未加密的密钥可以被中间人轻易截获。为了确保通信安全，我们引入了 RSA 算法来加密这些密钥。客户端使用服务器的公钥加密 DES 密钥后发送给服务器，服务器再用私钥解密。这种方法确保了即使密钥在传输过程中被截获，没有私钥的第三方也无法解密，从而保护了数据的安全性。
3. **数据加密和解密的效率问题**：在数据通信过程中，每次发送消息都需要加密，接收时则需要解密。最初，我们发现这一过程的 CPU 占用较高，影响了通信的实时性。通过优化算法实现和选择更有效的编程方法（例如，使用更有效的库函数），我们提高了加解密的效率。此外，引入异步 IO 操作，减少了加解密操作对主线程的阻塞，从而提高了整体的通信效率。
4. **客户端和服务器程序稳定性问题**：在初期测试中，程序偶尔会因为未处理的异常而崩溃。例如，网络断开或密钥错误导致的异常没有被妥善处理。通过在关键位置添加异常处理机制，并对网络状态和数据有效性进行检查，增强了程序的健壮性。

五、 实验结论

在完成这个关于 RSA 算法和 DES 加密的实验中，我获得了宝贵的学习经验和深刻的理解，这些经验和理解不仅涉及理论知识，也包括实际的编程实践。这次实验不仅加深了我对 RSA 算法的工作原理的理解，也让我熟悉了在 Linux 环境下使用 C++ 进行网络编程的技术。

通过动手实现 RSA 密钥生成、加密和解密的过程，我对非对称加密的内在机制有了更加直观的认识。在编写代码过程中，我需要确保所有数学运算都是安全和有效的，尤其是涉及到大数操作和模运算时。这不仅测试了我的编程技能，也促使我更深入地理解了算法的数学基础。此外，这次实验也让我体会到了加密技术在实际通信系统中的应用。通过将 RSA 算法与 DES 对称加密算法结合使用，我能够设计一个既安全又高效的加密通信系统。这种结合使用的方案克服了单独使用对称或非对称加密算法的各自缺点，提供了一个健壮的安全解决方案。

在实验过程中，我遇到了一些挑战，特别是在密钥交换和管理上。如何安全地交换密钥，如何保证通信双方的密钥同步，这些问题都需要仔细考虑。解决这些问题的过程增强了我的问题解决能力，并加深了我对网络安全领域的兴趣。通过这次实验，我也学习到了如何在 Linux 平台上使用高级编程技术进行网络编程，包括使用 sockets 进行数据传输，以及使用 epoll 技术处理多客户端连接。这些技能对于我的职业发展极为重要，它们提高了我的技术能力，并为我将来在 IT 和网络安全领域的工作打下了坚实的基础。

参 考 文 献

[1] 吴功宜. 网络安全高级软件编程技术 [M]. 清华大学出版社,2010.