

网络安全技术

实 验 报 告

学 院 网络空间安全学院
年 级 2021
学 号 2112060
姓 名 孙 璐

目录

一、 实验目的	1
二、 实验内容	1
三、 实验原理	1
1. RSA	1
2. 全双工通信	2
3. I/O 多路复用	2
四、 程序流程	6
五、 代码步骤	7
1. DES 密钥生成	7
2. RSA 的实现	7
3. I/O 多路复用	13
4. 消息发送与接收	19
六、 实验结果	21
七、 实验遇到的问题及其解决方法	22
八、 实验结论	22

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

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。

三、实验原理

1. RSA

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 ，加解密过程如下

$$(1) \quad C = M^e \% n$$

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

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

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 $n = p \times q$ ；函数 $\Phi(n)$ 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 $\Phi(n)$ 互质且小于 $\Phi(n)$ ，公钥 $\{e, n\}$ 已经确定；最后确定 d ，使得 $d \times e \equiv 1 \% \Phi(n)$ ，即 $(d \times e - 1) \% \Phi(n) = 0$ ，至此，私钥 $\{d, n\}$ 也被确定。

2. 全双工通信

由于 `accept` 函数、`read`、`write`、`recv`、`send` 等函数都是阻塞式的，在同一个进程之中，只要有任何一个函数没有执行完毕，处于阻塞状态，之后的函数与功能就不能处理，很难实现点对点的 Server-Client 全双工通信。因为全双工通信是非阻塞式的通信方式，即使对方没有回复消息，都可以随时发送。

3. IO 多路复用

IO 多路复用就是通过一个进程可以监视多个描述符，一旦某个描述符就绪，能够通知程序进行相应的读写操作。与多进程和多线程技术相比，IO 多路复用技术的优势是系统开销小，系统不必创建进程或线程，也不必维护这些进程，从而大大减小了系统的开销。目前支持 I/O 多路复用的系统调用有 `select`、`pselect`、`poll`、`epoll`。但 `select`、`poll`、`epoll` 本质上都是同步 I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步 I/O 则无需自己负责进行读写，异步 I/O 的实现会负责把数据从内核拷贝到用户空间。

(1) Select

```
int select(int maxfdpl, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
```

其中参数 `maxfdpl` 代表最大文件句柄，`readfds`、`writefds`、`exceptfds` 对应三个句柄集合，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。

- `FD_CLR(int fd, fd_set *set)`；用来清除句柄集合 `set` 中相关 `fd` 的项；
- `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

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

- `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, 当有错误发生时则返回 `-1`, 错误原因存于 `errno`:

- `EBADF`: 文件句柄为无效的或该文件已关闭;
- `EINTR`: 此调用被中断;
- `EINVAL`: 参数 `n` 为无效;
- `ENOMEM`: 核心内存不足;

`select` 调用过程如下:

1) 用户进程需要监控某些资源 `fds`, 在调用 `select` 函数后会阻塞, 操作系统会将用户线程加入这些资源的等待队列中。

2) 直到有描述副就绪(有数据可读、可写或有 `except`)或超时(`timeout` 指定等待时间, 如果立即返回设为 `null` 即可), 函数返回。

3) `select` 函数返回后, 中断程序唤起用户线程。用户可以遍历 `fds`, 通过 `FD_ISSET` 判断具体哪个 `fd` 收到数据, 并做出相应处理。

`select` 的缺点

1) 每次调用 `select` 都需要将进程加入到所有监视 `fd` 的等待队列, 每次唤醒都需要从每个队列中移除。这里涉及了两次遍历, 而且每次都要将整个 `fd_set` 列表传递给内核, 有一定的开销。

2) 能监听端口的数量有限, 单个进程所能打开的最大连接数由 `FD_SETSIZE` 宏定义, 监听上限就等于 `fds_bits` 位数组中所有元素的二进制位总数, 其大小是 32 个整数的大小。

3) 当函数返回时, 系统会将就绪描述符写入 `fd_set` 中, 并将其拷贝到用户空间。进程被唤醒后, 用户线程并不知道哪些 `fd` 收到数据, 还需要遍历一次。

(2) `Poll`

```
int poll(struct pollfd* fds, int nfds, int timeout);
```

Poll 的参数:

fds: struct pollfd 类型的数组, 存储了待检测的文件描述符, struct pollfd 有三个成员:

- fd: 委托内核检测的文件描述符
- events: 委托内核检测的 fd 事件 (输入、输出、错误), 每一个事件有多个取值
- revents: 这是一个传出参数, 数据由内核写入, 存储内核检测之后的结果

nfds: 描述的是数组 fds 的大小

timeout: 指定 poll 函数的阻塞时长

poll 函数与 select 原理相似, 都需要来回拷贝全部监听的文件描述符, 不同的是:

1) poll 函数采用链表的方式替代原来 select 中 fd_set 结构, 因此可监听文件描述符数量不受限。

2) poll 函数返回后, 可以通过 pollfd 结构中的内容进行处理就绪文件描述符, 相比 select 效率要高。

3) 新增水平触发: 也就是通知程序 fd 就绪后, 这次没有被处理, 那么下次 poll 的时候会再次通知同个 fd 已经就绪。

poll 缺点

和 select 函数一样, poll 返回后, 需要轮询 pollfd 来获取就绪的描述符。事实上, 同时连接的大量客户端在一时刻可能只有很少的处于就绪状态, 包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间, 而不论这些文件描述符是否就绪, 它的开销随着文件描述符数量的增加而线性增大。随着监视的描述符数量的增长, 其效率也会线性下降。

(3) Epoll

```
int epoll_create(int size); //创建一个 epoll 的句柄, size 用来告诉内核这个监听的数目一共有多大
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

epoll 使用一个文件描述符管理多个描述符, 将用户进程监控的文件描述符的事件存放到内核的一个事件表中, 这样在用户空间和内核空间只需拷贝一次。

- epoll_create

创建一个 epoll 的句柄,参数 size 并非限制了 epoll 所能监听的描述符最大个数,只是对内核初始分配内部数据结构的一个建议。

当 epoll 句柄创建后,它就会占用一个 fd 值,在 linux 中查看 /proc/ 进程 id/fd/,能够看到这个 fd,所以 epoll 使用完后,必须调用 close() 关闭,否则可能导致 fd 被耗尽。

- epoll_ctl

事件注册函数,将需要监听的事件和需要监听的 fd 交给 epoll 对象。

OP 用三个宏来表示:添加 (EPOLL_CTL_ADD)、删除 (EPOLL_CTL_DEL)、修改 (EPOLL_CTL_MOD)。分别表示添加、删除和修改 fd 的监听事件。

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

events 可以是以下几个宏的集合:

EPOLLIN : 表示对应的文件描述符可以读(包括对端 SOCKET 正常关闭);

EPOLLOUT: 表示对应的文件描述符可以写;

EPOLLPRI: 表示对应的文件描述符有紧急的数据可读(这里应该表示有带外数据到来);

EPOLLERR: 表示对应的文件描述符发生错误;

EPOLLHUP: 表示对应的文件描述符被挂断;

EPOLLET: 将 EPOLL 设为边缘触发(Edge Triggered)模式,这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT: 只监听一次事件,当监听完这次事件之后,如果还需要继续监听这个 socket 的话,需要再次把这个 socket 加入到 EPOLL 队列里

- epoll_wait

等待 epfd 上的 io 事件,最多返回 maxevents 个事件。参数 timeout 是超时时间(毫秒,0 会立即返回,-1 将不确定,也有说法说是永久阻塞)。

1) epoll_wait 调用 ep_poll,当 rdlist 为空(无就绪 fd)时挂起当前进程,直到 rdlist 不空时进程才被唤醒。

2) 文件 fd 状态改变(buffer 由不可读变为可读或由不可写变为可写),导致相应 fd 上的回调函数 ep_poll_callback() 被调用。

3) ep_poll_callback 将相应 fd 对应 epitem 加入 rdlist,导致 rdlist 不空,进程被唤醒,epoll_wait 得以继续执行。

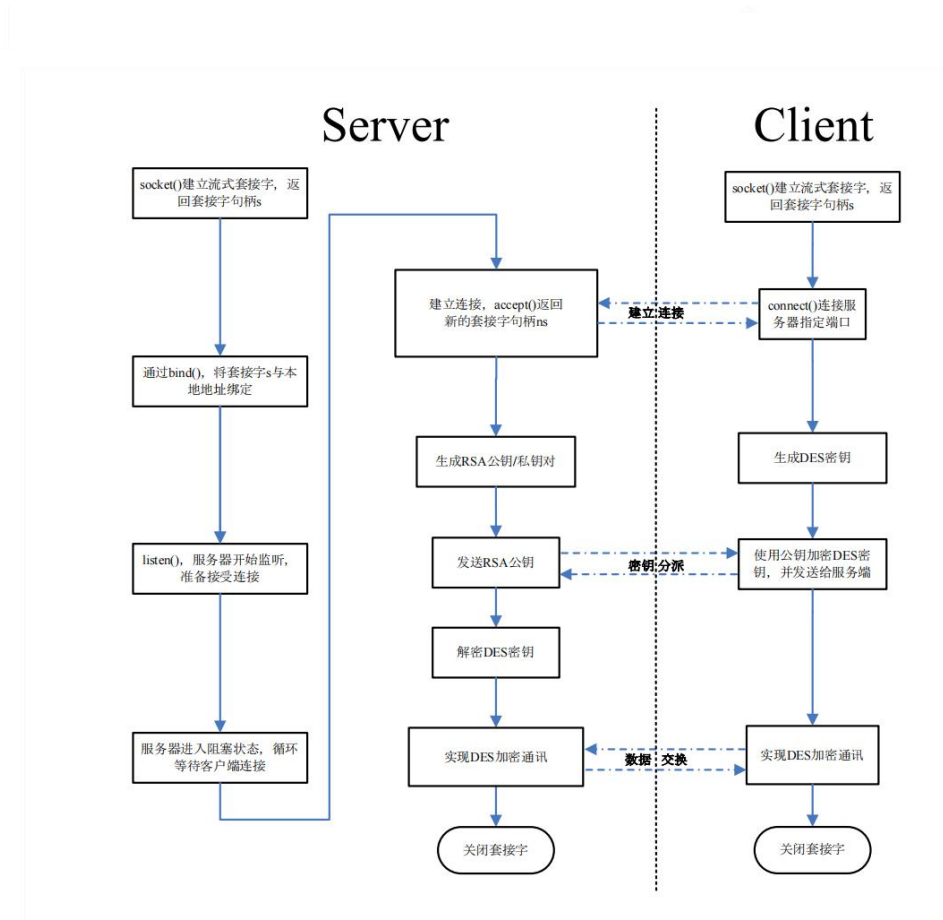
4) ep_events_transfer 函数将 rdlist 中的 epitem 拷贝到 txlist 中,并

将 rdlist 清空。

5) ep_send_events 函数，它扫描 txlist 中的每个 epitem，调用其关联 fd 对用的 poll 方法。此时对 poll 的调用仅仅是取得 fd 上较新的 events（防止之前 events 被更新），之后将取得的 events 和相应的 fd 发送到用户空间（封装在 struct epoll_event，从 epoll_wait 返回）。

四、程序流程

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。



五、代码步骤

1. DES 密钥生成

```
string DES_key_gen()
{
    string des_key;

    // 随机生成DES密钥
    srand((unsigned)time(NULL));
    for (int i = 0; i < 8; i++)
    {
        char temp = 65 + rand() % 26;
        des_key += temp;
    }

    return des_key;
    // cout << "Client: The plaintext of the DES key: " << des_key << endl;
}
```

2. RSA 的实现

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 ，加解密过程如下

$$(3) \quad C = M^e \% n$$

$$(4) \quad M = C^d \% n = (M^e)^d \% n = M^{d \times e} \% n = M$$

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

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 $n = p \times q$ ；函数 $\Phi(n)$ 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 $\Phi(n)$ 互质且小于 $\Phi(n)$ ，公钥 $\{e, n\}$ 已经确定；最后确定 d ，使得 $d \times e \equiv 1 \% \Phi(n)$ ，即 $(d \times e - 1) \% \Phi(n) = 0$ ，至此，私钥 $\{d, n\}$ 也被确定。

(1) 平方乘算法计算 $a^m \% n$

首先写出指数 m 的二进制，从最高位开始，如果当前位是 1，则将结果乘以底数 a ，然后对结果取模 n ；如果当前位是 0，则不进行操作。

每处理完一位，将底数 a 自乘，并对结果模 n 。

对每一位进行相同的处理，直到所有位都已处理完毕，最终得到的结果就是 $a^m \% n$ 的结果

```
// 计算 (a * b) % n 的乘法函数
unsigned long long multi(unsigned long long a, unsigned long long b, unsigned long long n)
{
    unsigned long long s = a;
    unsigned long long t = b;

    unsigned long long result = 0;
    a %= n;

    while (b > 0)
    {
        if (b % 2 == 1)
        {
            result = (result + a) % n;
        }
        a = (a * 2) % n;
        b /= 2;
    }

    // cout << "(a = " << s << " * b = " << t << ") % " << "n = " << n << " = " << result << endl;
    return result;
}
```

```
//平方乘法
unsigned long long square_and_multiply(unsigned long long base, unsigned long long exp, unsigned long long modulus)
{
    unsigned long long s = base;
    unsigned long long t = exp;

    unsigned long long result = 1;
    base = base % modulus;
    bitset<1024> binaryExponent(exp);

    for (int i = 0; i < binaryExponent.size(); i++)
    {
        // 如果指数的当前位是1, 乘以当前的 base
        if (binaryExponent[i] == 1)
        {
            result = (result * base) % modulus;
        }
        // 底数平方
        base = (base * base) % modulus;
    }

    // cout << "(a = " << s << " ^ b = " << t << ") % " << "n = " << modulus << " = " << result << endl;
    return result;
}
```

(2) Miller_Rabin 算法判定素数性质

首先对简单情况进行判断, 小于等于 1 的数不在讨论范围内; 2 和 3 是素数; 偶数一定不是素数。

```
// Miller-Rabin算法判定素数性质
bool Miller_Rabin(unsigned long long n)
{
    // 当n小于等于1时, 返回false
    if (n <= 1)
    {
        return false;
    }
    // 2和3是素数
    if (n <= 3)
    {
        return n == 2 || n == 3;
    }
    // 如果n是偶数, 则不是素数
    if (n % 2 == 0)
    {
        return false;
    }
}
```

寻找 s 和 t, 满足 $2^s * t = n - 1$ 。通过 while 循环实现

```

// 寻找 s 和 t, 将 n-1分解成2^s * t, 其中t是奇整数
unsigned long long t = n - 1;
unsigned long long s = 0;

while (t % 2 == 0)
{
    t /= 2;
    s++;
}

```

然后在 for 循环里进行如下判断。在区间 $[2, n-2]$ 中随机取一个整数 b , 计算 $r_0 = b^t \bmod n$ 。如果 $b^t \bmod n$ 等于 1 或 $n-1$, 那么 n 可能为素数; 如果存在一个 $r \in [0, s-1]$, 满足 $r_0^{2^r} \bmod n = -1$, 那么 n 可能是素数。但是如果上面两个条件都不成立, 那么 n 一定是合数。

```

// 进行 k 次测试, k = n - 3
// 随机选择一个在区间 [2, n-2] 中的整数 b, 计算 r0 = b^t mod n
// 如果 b^t mod n = 1 或 n - 1, 则 n 可能是素数
// 如果存在一个 r, 0 <= r < s, 使得 r0^{2^r} mod n = -1, n 可能是素数
// 如果上述两个条件都不成立, 则 n 一定是合数

for (int i = 0; i < 100; i++)
{
    // 随机选择基数 b, 范围在 [2, n-2]
    unsigned long long b = 2 + rand() % (n - 4);
    // long long b = 2 + i;

    // 计算 a^t % n
    unsigned long long r0 = square_mult(b, t, n);
    // unsigned long long r0 = square_and_multiply(b, t, n);

    unsigned long long r1;

    // 如果 r0 结果为 1 或 n-1 或 0, 则通过检验, 重选 b 继续下一次测试
    if (r0 == 1 || r0 == n - 1)
    {
        continue;
    }

    // 如果 r0 != 1 或 r0 != n-1, 则计算 r1 = r0^2 mod n
    // 进行 s-1 次平方运算
    else
    {
        for (int r = 0; r < s; r++)
        {
            r1 = multi(r0, r0, n);

            // 如果 r1 = 1, 则 n 一定是合数
            if (r1 == 1)
            {
                return false;
            }

            // 如果 r1 = n-1, 则通过检验, 重选 b 继续下一次测试
            if (r1 == n - 1)
            {
                break;
            }

            // 更新 r0 为 r1, 准备下一轮的平方取模运算
            r0 = r1;
        }

        // 如果循环结束时, r1 仍然不等于 n-1, 仍未找到 n 可能是素数的证据, 则 n 不是素数
        if (r1 != n - 1)
        {
            return false;
        }
    }

    // 测试中都没有发现 n 不是素数的证据, 则 n 可能是素数
    return true;
}

```

(3) 质数生成

首先要生成一个随机奇数，然后通过 Miller_Rabin 算法判定他是不是质数，如果不是，那么重复生成直到产生一个质数

```
// 质数生成函数
// 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。
unsigned long long RandomPrime(char bits)
{
    unsigned long long base;
    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_Isprime(base, 30)); //测试 30 次
    }
    return base; //全部通过认为是质数
}
```

(4) Euler 函数

Euler 函数，返回小于 n 且与 n 互质的正整数个数。但是在 rsa 中， $n = p \times q$ ， $\phi(n) = (p-1) * (q-1)$ 。

```
// 返回小于n且与n互质的正整数个数
unsigned long long Euler(unsigned long long n)
{
    unsigned long long res = n, a = n;
    for (unsigned long long i = 2; i * i <= a; ++i)
    {
        if (a % i == 0)
        {
            res = res / i * (i - 1); //先进行除法是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;
}
```

欧拉函数的计算方法是通过对于 n 进行质因数分解，然后利用欧拉函数的性质来计算。

欧拉函数计算公式如下，其中 p_1, \dots, p_r 是 n 的因子。

$$\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \dots (1 - \frac{1}{p_r})$$

在这段函数中，对 n 进行质因数分解，遍历 2 到 \sqrt{n} 的所有可能因子 i ，如果 i 是 n 的因子，更新 $res = res / i * (i - 1)$ ，同时不断除以 i 直到 a 不能再整除 i 。

(5) RSA 私钥生成

选择一个任意正整数 e ，使其与 $\phi(n)$ 互质且小于 $\phi(n)$ ，公钥 $\{e, n\}$ 已经确定；最后确定 d ，使得 $d \times e \equiv 1 \pmod{\phi(n)}$ ，即 $(d \times e - 1) \% \phi(n) = 0$ ，至此，私钥 $\{d, n\}$ 也被确定。

0xffffffffffffffff 是一个 64 位无符号整数的最大值。

检查 $(i \times t + 1)$ 是否能被 e 整除，如果是，将 d 设置为 $((i \times t + 1) / e)$ ，然后

返回该值；如果 $(i \times t + 1)$ 不能被 e 整除，继续递增 i 直到满足 $(i \times t + 1)$ 是否能被 e 整除或者超过了 MAX 值。

```
// 私钥生成
unsigned long long Euclid(unsigned long long e, unsigned long long t)
{
    unsigned long long Max = 0xffffffffffffffff - t;
    unsigned long long i = 1;
    unsigned long long d = 0;

    while (1)
    {
        if (((i * t) + 1) % e == 0)
        {
            return d = ((i * t) + 1) / e;
        }

        i++;
        unsigned long long tmp = (i + 1) * t;
        if (tmp > Max)
        {
            return 0;
        }
    }

    return 0;
}
```

(6) RSA 公钥对私钥对生成

随机生成两个大素数 p 和 q ， $n=p \times q$ 。 $\phi(n)=(p-1) \times (q-1)$ 。选择一个任意正整数 e ，使其与 $\phi(n)$ 互质且小于 $\phi(n)$ ， $\{e, n\}$ 就是公钥对。

```
// 服务器端生成RSA公私钥对
unsigned long long RSA_key_pair_gen(RSA_key_pair& key_pair)
{
    // 生成公钥(e, n)
    unsigned long long Prime_p = RandomPrime(16); // 随机生成两个素数
    unsigned long long Prime_q = RandomPrime(16);
    unsigned long long n = Prime_p * Prime_q;
    unsigned long long f = (Prime_p - 1) * (Prime_q - 1);

    unsigned long long euler = Euler(n);
    unsigned long long e;

    while (1)
    {
        e = rand() % 65536 + 1;
        e |= 1;
        if (gcd(e, f) != 1)
        {
            break;
        }
    }

    cout << "RSA_publickey: " << "{ e: " << e << ", n: " << n << } " << endl;
}
```

最后确定 d ，使得 $d \times e \equiv 1 \pmod{\phi(n)}$ ，即 $(d \times e - 1) \pmod{\phi(n)} = 0$ ，至此，私钥 $\{d, n\}$ 也被确定。

```

// 生成私钥(d, n)
unsigned long long max = 0xffffffffffffff - euler;
unsigned long long i = 1;
unsigned long long d = 0;

while (1)
{
    if (((i * euler) + 1) % e == 0)
    {
        d = ((i * euler) + 1) / e;
        break;
    }
    i++;
    unsigned long long temp = (i + 1) * euler;
    if (temp > max)
    {
        break;
    }
}

cout << "RSA_privatekey : " << "d: " << d << ", n: " << n << " " << endl;
cout << endl;

// 如果循环结束后d值仍然为0表示密钥生成失败
if (d == 0)
{
    return -1;
}

```

将 RSA 公钥对存储为字符串的形式，中间用 “,” 分隔开

```

key_pair.public_key_e = e;
key_pair.private_key_d = d;
key_pair.n = n;
key_pair.en = to_string(e) + "," + to_string(n);
return 0;

```

(7) 客户端使用收到的 RSA 公钥对加密 DES 密钥

将 8 个字母 string 类型的 DES 密钥转为 64 位整数。64 位整数拆成 4 份，每份 16 位，对每一份使用 RSA 公钥执行一次 RSA 加密，再将结果转为 string 字符串，每一份加密结果用 “,” 分开。

```

// 客户端使用服务器的公钥加密DES密钥，64位为需要分四次加密
// 使用服务器公钥，计算 C = (M^e) mod n
// 返回值是4个string类型的16位加密结果

string client_encry(string des_key, unsigned long long public_key_e, unsigned long long n)
{
    // 将8个字母的string类型DesKey转为64位int类型
    unsigned long long _des_key = 0;
    for (unsigned int i = 0; i < des_key.length(); ++i)
    {
        _des_key += des_key[i];
        if (i != des_key.length() - 1)
        {
            _des_key <<= 8;
        }
    }

    // 64位int拆成4份，每份16位
    unsigned short* p_res = (unsigned short*)&_des_key;
    unsigned short M[4];
    for (int i = 0; i < 4; ++i)
    {
        M[i] = p_res[i];
    }

    string result;
    string result_out;
    // 对每一份执行加密函数，并将4个16位数字转成string，用逗号分隔
    for (int i = 3; i >= 0; --i)
    {
        string temp = to_string(PowMod(M[i], public_key_e, n));
        result += temp;
        // result_out += temp;
        result += ',';
    }

    // cout << "result_out: " << result_out << endl;
    return result;
}

```


(8) 服务器端使用私钥解密客户端发来的 DES 密钥

服务器端使用自己保存的 RSA 私钥对，对收到的经 RSA 公钥对加密后的 DES 密钥进行解密。

读取被分割成 4 份用“,”分割的字符串，对每一份字符串转换成 unsigned long long 类型的整数，使用 RSA 私钥对进行解密，并转换成对应的两个字符，得到解密后的 DES 密钥。

```
// 服务器使用私钥解密客户端发来的DES密钥
// 使用服务器私钥，计算 M = (C^d) mod n
string server_decry(string key_info, unsigned long long private_key_d, unsigned long long n)
{
    string des_key = "";
    int pos = 0;
    for (int i = 0; i < 4; i++)
    {
        string temp = "";
        for (; key_info[pos] != ','; ++pos)
        {
            temp += key_info[pos];
        }
        ++pos;
        // cout << i << endl;
        // string temp = key_info.substr(2 * i, 2 * i + 1);
        // cout << i << endl;

        unsigned long long Ci = strtoull(temp);
        unsigned long long n_res = PowMod(Ci, private_key_d, n);
        unsigned short* p_res = (unsigned short*)&n_res;

        if (p_res[1] != 0 || p_res[2] != 0 || p_res[3] != 0)
        {
            // error
            // printf("server server_decry() error!\n");
            return 0;
        }
        else // p_res[0]是16bit数字，可以转成2个字母
        {
            des_key += short2str(p_res[0]);
        }
    }

    // cout << "des_key" << des_key << endl;

    return des_key;
}
```

3. I/O 多路复用

(1) 服务器端与客户端创建实例

在服务器端的主循环中，使用 `epoll_create` 创建了一个 `epoll` 实例 `epoll_fd`。

`fcntl(serverSocket, F_SETFL, O_NONBLOCK)` 设置了服务器套接字 `serverSocket` 为非阻塞模式。此时调用 `recv`、`send` 等 I/O 函数不会阻塞程序的执行，而是立即返回。这样可以避免服务器在等待客户端连接时被阻塞，从而提高了服务器的并发性能。

配置 `ep_event` 结构体，表示要监听的事件。使用 `epoll_ctl` 将服务器套接字 `serverSocket` 添加到 `epoll` 实例中，以监听客户端连接请求的到达。设置 `ep_event.events = EPOLLIN | EPOLLET`，指定监听的事件类型为可读事件和边沿触发模式。

配置 `ep_input` 结构体，表示要监听标准输入的可读事件并使用边缘触发模

式。设置 `ep_input.events = EPOLLIN | EPOLLET`，指定标准输入事件类型为可读事件和边缘触发模式。

```
int epoll_fd = epoll_create(256); // 生成一个专用的epoll文件描述符
if (epoll_fd < 0)
{
    cout << "Server: Server epoll_create failed ! Error: " << strerror(errno) << endl;
    return -1;
}

struct epoll_event ep_event, ep_input; // 针对监听的fd_skt, 创建2个epoll_event
struct epoll_event ret_events[20]; // 数组用于回传要处理的事件

fcntl(serverSocket, F_SETFL, O_NONBLOCK); // 设置非阻塞
// ep_event用于注册事件
ep_event.data.fd = serverSocket;
ep_event.events = EPOLLIN | EPOLLET;
// 用于控制某个文件描述符上的事件 (注册, 修改, 删除)
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, serverSocket, &ep_event) < 0)
{
    cout << "Server: Epoll_ctl ep_event failed . Error: " << strerror(errno) << endl;
    return -1;
}

// 给serverSocket绑定监听标准输入的文件描述符
fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
ep_input.data.fd = STDIN_FILENO;
ep_input.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, STDIN_FILENO, &ep_input) < 0)
{
    cout << "Server: Epoll_ctl ep_input failed . Error: " << strerror(errno) << endl;
    return -1;
}

int ret_num = 20;
char server_des_cipherkey[BUFFER_SIZE]{};
memset(server_des_cipherkey, 0, sizeof(server_des_cipherkey));
```

```
int epoll_fd = epoll_create(256); // 生成一个专用的epoll文件描述符
if (epoll_fd < 0)
{
    cout << "Client: Client epoll_create failed ! Error: " << strerror(errno) << endl;
    return -1;
}

struct epoll_event ep_event, ep_input; // 针对监听的socket, 创建2个epoll_event
struct epoll_event ret_events[20]; // 数组用于回传要处理的事件

fcntl(clientSocket, F_SETFL, O_NONBLOCK); // 设置非阻塞
// ep_event用于注册事件
ep_event.data.fd = clientSocket;
ep_event.events = EPOLLIN | EPOLLET;
// 用于控制某个文件描述符上的事件 (注册, 修改, 删除)
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, clientSocket, &ep_event) < 0)
{
    cout << "Client: Epoll_ctl ep_event failed . Error: " << strerror(errno) << endl;
    return -1;
}

// 给clientSocket绑定监听标准输入的文件描述符
fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
ep_input.data.fd = STDIN_FILENO;
ep_input.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, STDIN_FILENO, &ep_input) < 0)
{
    cout << "Client: Epoll_ctl ep_input failed . Error: " << strerror(errno) << endl;
    return -1;
}

int ret_num = 20;
```

(2) 服务器端

A) Epoll_wait 等待 epoll 实例中的事件


```

cin.ignore(2048, '\n');
while (1)
{
    //用于轮寻I/O事件的发生
    switch (int event_num = epoll_wait(epoll_fd, ret_events, ret_num, epoll_timeout))
    {
        // 返回值等于0表示超时
    case 0:
        cout << "Server: Epoll_wait: Time out!" << endl;
        break;
        // 返回值小于0表示出错
    case -1:
        cout << "Server: Epoll_wait: Wrong!" << endl;
        break;
    }
}

```

调用 `epoll_wait` 等待事件的发生。`epoll_fd` 是 `epoll` 实例的文件描述符，`ret_events` 是用于存储发生事件的数组，`ret_num` 是最多等待并返回的事件数量，`epoll_timeout` 是超时时间。`epoll_wait` 函数会阻塞程序直到有事件发生或者超时。如果返回值是 0 表示超时，如果返回值为-1 表示出错。返回值大于 0 将进入循环处理事件。

B) 有新的客户端连接请求

a) 如果事件来自服务器监听的套接字 `serverSocket`，则表示有新的客户端连接请求。获取客户端的地址信息和套接字描述符。

```

// 有来自客户端新的连接
if (ret_events[i].data.fd == serverSocket)
{
    //等待客户端的连接

    sockaddr_in* clientAddr = new sockaddr_in();
    socklen_t clientAddrLen = sizeof(clientAddr);
    int client_Socket = acceptClient(serverSocket, (sockaddr_in*)clientAddr, &clientAddrLen);
}

```

b) 在服务器端生成 RSA 密钥对

获取客户端的信息后，服务器端调用 `RSA_key_pair_gen()`生成 RSA 密钥对

```

// 生成RSA公钥私钥对
cout << "##### RSA_KEY_PAIR_GEN #####" << endl;
// RSA_key_pair key_pair;
while (1)
{
    if (RSA_key_pair_gen(key_pair) == 0)
    {
        // cout << "True!" << endl;
        break;
    }
}

```

c) 监听客户端的消息

将 RSA 公钥对发送给客户端，并将客户端套接字添加到 `epoll` 实例中监听客户端的消息。

```

client = new CLIENT();
client->client_Socket = client_Socket;
client->client_addr = clientAddr;
client->rsa_keypair = key_pair;

fcntl(client_Socket, F_SETFL, O_NONBLOCK);
struct epoll_event new_client_event;
new_client_event.events = EPOLLIN | EPOLLET;
new_client_event.data.ptr = client;

// 将RSA公钥对发送给客户端
if (send(client_Socket, (void*)rsa_keypair, sizeof(rsa_keypair), 0) < 0)
{
    cout << "Server: Failed to send RSA_key_pair_public_key." << endl;
    return -1;
}
else
{
    cout << "Server: Send RSA_key_pair_public_key successfully!!" << endl;
}

// epoll监听客户端发来的消息
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_Socket, &new_client_event) < 0)
{
    cout << "Epoll_ctl error!" << endl;
    return -1;
}

```

C) 可读事件

else if (ret_events[i].events == EPOLLIN): 如果事件是可读事件。

a) 标准输入 STDIN_FILENO 可读事件，调用 Server_send_thread 处理

```

// 标准输入
if (ret_events[i].data.fd == STDIN_FILENO)
{
    Server_send_thread(client->client_Socket);
}

```

b) 如果不是标准输入事件，且还未获取 des 密钥，那么应该先获取客户端发来的 des 密钥

通过 recv 接收客户端发来的消息，用服务器端自己的 RSA 私钥对进行解密得到 des 密钥的明文。收到正确的密钥后向客户端发送 GetDESKeySuccess 消息表明服务器端已收到密钥。

```

else
{
    if (getdeskey == false)
    {
        // 进行正常通信
        CLIENT* client = (CLIENT*)ret_events[i].data.ptr;

        if (recv(client->client_Socket, (void*)server_des_cipherkey, BUFFER_SIZE, 0) < 0)
        {
            cout << "Server: Fail to receive the ciphertext of DES key from the client!" << endl;
        }
    }
    else
    {
        cout << "Server : The ciphertext of DES key is: " << server_des_cipherkey << endl;

        // char转string
        string str_des_cipherkey = server_des_cipherkey;
        // cout << "Server : The ciphertext of DES key is(string): " << server_des_cipherkey << endl;

        string des_key = "";
        des_key = server_decry(str_des_cipherkey, key_pair.private_key_d, key_pair.n);
        cout << "Server: The plaintext of DES key is: " << des_key << endl;

        getdeskey = true;

        char des_key[8192];
        for (int i = 0; i < des_key.size(); i++)
        {
            des_key[i] = des_key[i];
        }
        des_key[des_key.size()] = '\0';

        BlockFromStr(bkey, des_key); // 获取密钥

        char getdeskeysuccess[BUFFER_SIZE]{};
        memset(getdeskeysuccess, 0, BUFFER_SIZE);

        strcpy(getdeskeysuccess, "GetDESKeySuccess");
        EncryptMsg(getdeskeysuccess, bkey);
        send(client->client_Socket, (void*)getdeskeysuccess, sizeof(getdeskeysuccess), 0);

        cout << "##### Let's chat #####" << endl;
    }
}

```

c) 如果已经获取到 des 密钥，则调用 Server_recv_thread 进行消息接收。

```

else
{
    Server_recv_thread(client->client_Socket);
}

```

(3) 客户端

A) Epoll_wait 等待 epoll 实例中的事件

```

cin.ignore(2048, '\n');
while (1)
{
    //用于轮寻I/O事件的发生
    switch (int event_num = epoll_wait(epoll_fd, ret_events, ret_num, epoll_timeout))
    {
        // 返回值等于0表示超时
        case 0:
            cout << "Client: Epoll_wait: Time out!" << endl;
            break;
        // 返回值小于0表示出错
        case -1:
            cout << "Client: Eppoll_wait: Wrong!" << endl;
            break;
        // 返回值大于0表示事件的个数
    }
}

```

调用 `epoll_wait` 等待事件的发生。`epoll_fd` 是 `epoll` 实例的文件描述符，`ret_events` 是用于存储发生事件的数组，`ret_num` 是最多等待并返回的事件数量，`epoll_timeout` 是超时时间。`epoll_wait` 函数会阻塞程序直到有事件发生或者超时。如果返回值是 0 表示超时，如果返回值为 -1 表示出错。返回值大于 0 将进入循环处理事件。

B) 循环处理事件

if (ret_events[i].events == EPOLLIN): 如果事件是可读事件。

- a) 如果是标准输入 STDIN_FILENO 可读事件, 则调用 Client_send_thread 向服务器端发送消息

```
// 标准输入
if (ret_events[i].data.fd == STDIN_FILENO)
{
    Client_send_thread(clientSocket); // 给服务器发消息
}
```

- b) 如果不是标准输入事件的可读事件, 且还没获取到服务器的 RSA 公钥对, 则应先从服务器接受 RSA 公钥对。如果成功收到 RSA 的公钥对, 则应使用 RSA 公钥对加密 DES 密钥发送给服务器端。

```
if (getrsakey == false)
{
    // 接收服务器的公钥对
    int len_n = recv(clientSocket, (void*)server_rsa_keypair, BUFFER_SIZE, 0);
    cout << "Client: RSA_keypair from the server: " << server_rsa_keypair << endl;
    // char转string
    str_rsa_keypair = server_rsa_keypair;

    // cout << "Client: RSA_keypair from the server: " << str_rsa_keypair << endl;

    int pos = str_rsa_keypair.find(",", 0);
    unsigned long long rsa_public_key_e = strtouint(str_rsa_keypair.substr(0, pos));
    unsigned long long n = strtouint(str_rsa_keypair.substr(pos + 1, str_rsa_keypair.size()));
    // cout << "Client: RSA_keypair_publickey_e is: " << rsa_public_key_e << endl;
    // cout << "Client: RSA_keypair_publickey_n is:" << n << endl;
    getrsakey = true;

    // 使用RSA的公钥加密DES密钥
    string encrypted_DesKey = client_encry(des_key, rsa_public_key_e, n);
    cout << "Client: The ciphertext of the DES key: " << encrypted_DesKey << endl;

    char encrypted_DesKey_[8192];
    for (int i = 0; i < encrypted_DesKey.size(); i++)
    {
        encrypted_DesKey_[i] = encrypted_DesKey[i];
    }
    encrypted_DesKey_[encrypted_DesKey.size()] = '\0';

    // 给服务器发送加密过的DES密钥
    if (send(clientSocket, (void*)encrypted_DesKey_, sizeof(encrypted_DesKey_), 0) < 0)
    {
        cout << "Client: Fail to send the ciphertext of DES key successfully! " << endl;
    }
    else
    {
        cout << "Client: Send the ciphertext of DESkey successfully!!" << endl;
    }

    cout << "##### Let's chat #####" << endl;
}
```

- c) 如果已经获取到 rsa 公钥对, 则调用 Client_recv_thread 进行消息接收。

```
else
{
    Client_recv_thread(clientSocket); // 接收服务器消息
}
```

4. 消息发送与接收

为实现全双工通信，服务器端与客户端的消息发送与消息接收两个功能需求分别用两个函数实现。与第一次作业的多线程实现的两个函数类似。

(1) 服务器端

A) 消息发送

获取输入时从 cin 换成 read(STDIN_FILENO, server_send_buffer, BUFFER_SIZE)函数

```
void Server_send_thread(int client_Socket)
{
    char server_send_buffer[BUFFER_SIZE];

    // while (!exit_flag)
    // {
    //     // 发送消息给客户端
    memset(server_send_buffer, 0, BUFFER_SIZE);
    //     cout << "Please input the message that the server will transmit: " << endl;
    //     cin.getline(server_send_buffer, BUFFER_SIZE - 1);
    int count = 0;
    int sum = 0;
    while ((count = read(STDIN_FILENO, server_send_buffer, BUFFER_SIZE)) > 0)
    {
        sum += count;
    }
    if (count == -1 && errno != EAGAIN)
    {
        cout << "Server_send_thread: Get_message wrong!" << endl;
        return; // 发送信息给客户端读取错误
    }

    if (strcmp(server_send_buffer, "exit") == 0)
    {
        exit_flag = true;
        // break;
        return;
    }

    EncryptMsg(server_send_buffer, bkey);

    cout << "Server_send_thread: The ciphertext that the server will transmit: " << server_send_buffer << endl;

    if (send(client_Socket, (void*)server_send_buffer, BUFFER_SIZE, 0) < 0)
    {
        cout << "Error: failed to send message to client." << endl;
        // break;
        return;
    }
    // }
    // pthread_exit(NULL);
}
```

B) 消息接收


```

char server_recv_buffer[BUFFER_SIZE] {};

// while (!exit_flag)
// {
//     // 接收客户端发送的消息
memset(server_recv_buffer, 0, BUFFER_SIZE);

int recv_len = recv(client_Socket, (void*)server_recv_buffer, BUFFER_SIZE, 0);

cout << "Server_recv_thread: The ciphertext that received from client: " << server_recv_buffer << endl;

if (recv_len < 0)
{
    cout << "Failed to receive message from client." << endl;
    // break;
    return;
}
if (recv_len == 0)
{
    cout << "Client closed the connection." << endl;
    exit(0);
    // break;
    return;
}

// char* recv_decrypt_buffer = DecryptMsg(Server_recv_buffer, bkey);
// 显示客户端发送的消息
// cout << "Received message from client: plaintext: " << recv_decrypt_buffer << endl;

// 显示客户端发送的消息
DecryptMsg(server_recv_buffer, bkey);
cout << "Server_recv_thread: The plaintext that the server received from client: ";
cout << server_recv_buffer << endl;
// }
// pthread_exit(NULL);

```

(2) 客户端

A) 消息发送

获取输入时从 cin 换成 read(STDIN_FILENO, client_send_buffer, BUFFER_SIZE)函数

```

void Client_send_thread(int clientSocket)
{
    char client_send_buffer[BUFFER_SIZE] {};

    // while (!exit_flag)
    // {
    //     // 发送消息给服务器
memset(client_send_buffer, 0, BUFFER_SIZE);

// cout << "Please input the message that the client will transmit: " << endl;
// cin.getline(client_send_buffer, BUFFER_SIZE - 1);

int count = 0;
int sum = 0;
while ((count = read(STDIN_FILENO, client_send_buffer, BUFFER_SIZE)) > 0)
{
    sum += count;
}

if (count == -1 && errno != EAGAIN)
{
    cout << "Send_send_thread: Get_message wrong!" << endl;
    return; // 发送信息给客户端读取错误
}

if (strcmp(client_send_buffer, "exit") == 0)
{
    exit_flag = true;
    cout << "Client_send_thread: exit" << endl;
    // break;
    return;
}

EncryptMsg(client_send_buffer, bkey);
cout << "Client_send_thread: ciphertext that the client will transmit: " << client_send_buffer << endl;

if (send(clientSocket, (void*)client_send_buffer, BUFFER_SIZE, 0) < 0)
{
    cout << "Error: failed to send message to server." << endl;
    // break;
    return;
}
// }
// pthread_exit(NULL);

```

B) 消息接收

```
void Client_rcv_thread(int clientSocket)
{
    char client_rcv_buffer[BUFFER_SIZE] {};
    // char receive_decrypt_buffer[BUFFER_SIZE] {};

    // while (!exit_flag)
    // {
        // 接收服务器发送的消息
        memset(client_rcv_buffer, 0, BUFFER_SIZE);
        // memset(receive_decrypt_buffer, 0, BUFFER_SIZE);

        int rcv_len = recv(clientSocket, (void*)client_rcv_buffer, BUFFER_SIZE, 0);
        cout << "Client_rcv_thread: The ciphertext that received from server: " << client_rcv_buffer << endl;
        if (rcv_len < 0)
        {
            cout << "Failed to receive message from server." << endl;
            // break;
            return;
        }

        if (rcv_len == 0)
        {
            cout << "Server closed the connection." << endl;
            exit(0);
            // break;
            return;
        }

        // char *receive_decrypt_buffer= DecryptMsg(receive_buffer, bkey);

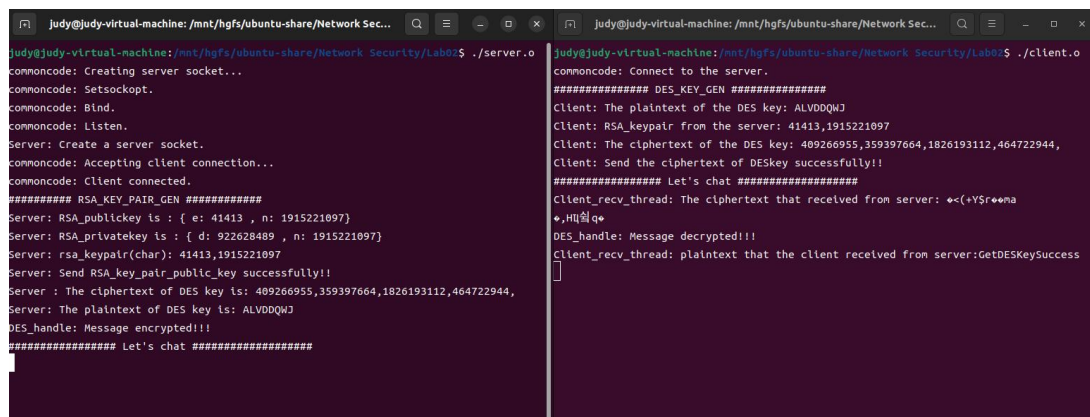
        // 显示服务器发送的消息
        // cout << "Received message from server: plaintext: " << receive_decrypt_buffer << endl;

        // 显示服务器发送的消息
        DecryptMsg(client_rcv_buffer, bkey);
        cout << "Client_rcv_thread: plaintext that the client received from server:";
        cout << client_rcv_buffer << endl;
        // }
        // pthread_exit(NULL);
    }
}
```

六、实验结果

(1) DES 密钥生成、RSA 密钥生成及传输

在客户端与服务器建立连接后，客户端首先随机生成一个长度为 8 的字符串充当 DES 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。



```
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security/Lab02$ ./server.o
commoncode: Creating server socket...
commoncode: Setsockopt.
commoncode: Bind.
commoncode: Listen.
Server: Create a server socket.
commoncode: Accepting client connection...
commoncode: Client connected.
##### RSA_KEY_PAIR_GEN #####
Server: RSA_publickey is : { e: 41413 , n: 1915221097}
Server: RSA_privatekey is : { d: 922628489 , n: 1915221097}
Server: rsa_keypair(char): 41413,1915221097
Server: Send RSA_key_pair_public_key successfully!!
Server: The ciphertext of DES key is: 409266955,359397664,1826193112,464722944,
Server: The plaintext of DES key is: ALVDDQWJ
DES_handle: Message encrypted!!!
##### Let's chat #####

judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security/Lab02$ ./client.o
commoncode: Connect to the server.
##### DES_KEY_GEN #####
Client: The plaintext of the DES key: ALVDDQWJ
Client: RSA_keypair from the server: 41413,1915221097
Client: The ciphertext of the DES key: 409266955,359397664,1826193112,464722944,
Client: Send the ciphertext of DESkey successfully!!
##### Let's chat #####
Client_rcv_thread: The ciphertext that received from server: 409266955,359397664,1826193112,464722944,
DES_handle: Message decrypted!!!
Client_rcv_thread: plaintext that the client received from server: Get DES Keys success
```

(2) 加解密 (服务器端→客户端, 即服务器端发送, 客户端接收)

服务器端获取到输入的消息后，使用 DES 密钥进行加密，将加密后的密文发送给客户端，客户端接收到消息后，使用 DES 密钥对密文消息进行解密，得到消息的明文。

```
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Sec... Q H x
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security/Lab0$ ./server.o
commoncode: Creating server socket...
commoncode: Setsockopt.
commoncode: Bind.
commoncode: Listen.
Server: Create a server socket.
commoncode: Accepting client connection...
commoncode: Client connected.
##### RSA_KEY_PAIR_GEN #####
Server: RSA_publickey is : { e: 41413 , n: 1915221097}
Server: RSA_privatekey is : { d: 922628489 , n: 1915221097}
Server: rsa_keypair(char): 41413,1915221097
Server: Send RSA_key_pair_public_key successfully!!
Server : The ciphertext of DES key is: 409266955,359397664,1826193112,464722944,
Server: The plaintext of DES key is: ALVDDQWJ
DES_handle: Message encrypted!!!
##### Let's chat #####
goodgoodgoodgood
DES_handle: Message encrypted!!!
Server_send_thread: The ciphertext that the server will transmit: 2..000+2..000+000
S)0w
sdavkjabbvjkbvjskdbvl
DES_handle: Message encrypted!!!
Server_send_thread: The ciphertext that the server will transmit: X
L00000j=0000."o>Gkoo

judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Sec... Q H x
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security/Lab0$ ./client.o
commoncode: Connect to the server.
##### DES_KEY_GEN #####
Client: The plaintext of the DES key: ALVDDQWJ
Client: RSA_keypair from the server: 41413,1915221097
Client: The ciphertext of the DES key: 409266955,359397664,1826193112,464722944,
Client: Send the ciphertext of DESkey successfully!!
##### Let's chat #####
Client_recv_thread: The ciphertext that received from server: 0<($>reona
0,HU0q0
DES_handle: Message decrypted!!!
Client_recv_thread: plaintext that the client received from server:GetDESKeySuccess
Client_recv_thread: The ciphertext that received from server: 2..000+2..000+000S)0w
DES_handle: Message decrypted!!!
Client_recv_thread: plaintext that the client received from server:goodgoodgoodgood
Client_recv_thread: The ciphertext that received from server: X
L00000j=0000."o>Gkoo
DES_handle: Message decrypted!!!
Client_recv_thread: plaintext that the client received from server:sdavkjabbvjkbvjsk
dbvl
[]
```

(2) 加解密(客户端→服务器端, 即客户端发送, 服务器端接收)

客户端端获取到输入的消息后，使用 DES 密钥进行加密，将加密后的密文发送给服务器端，服务器端接收到消息后，使用 DES 密钥对密文消息进行解密，得到消息的明文。

```

Server_rcv_thread: The ciphertext that received from client: 00'De|00;/0000eZ0000
8007B
DES_handle: Message decrypted!!!
Server_rcv_thread: The plaintext that the server received from client: asdfghjklqwe
rtuolopzxcvbnm
DES_handle: Message encrvpted!!!

Server_rcv_thread: The ciphertext that received from client: 0G00/00000000Eh0|8uI
00Re9 000|50000D0
.Z0
DES_handle: Message decrypted!!!
Server_rcv_thread: The plaintext that the server received from client: vsdagvvhvvhv
kcdvabvnaaEotewrchur10bv7bduktkhe1kvb1a1bhuk1
vsdagvvhvvhvsksdvajhvgaufgiewrbvriqbwjbjwkvbsjkbv1ajkbvjkl
DES_handle: Message encrypted!!!
Client_send_thread: ciphertext that the client will transmit: 0G00/00000000Eh0|8uI
00Re9 000|50000D0
.Z0

```

七、实验遇到的问题及其解决方法

(1) Linux 下 C++ Socket 编程与 Windows 下还是有一定不同的，两个环境下使用的库有的是不兼容的，有的函数在 Windows C++下能用但 Linux C++是不能使用的。比如创建线程的函数，Windows 中使用的是 `CreateThread`，而在 Linux 环境下使用的是 `pthread_create`。能使用的指针类型也不同，比如 Linux 下要用 `pthread_t` 类型，而 Windows 下可以使用 `lparam` 等。还有很多

这样的例子，通过报错提示和网络查询，一步一步修改解决了类似的问题，进一步熟悉了 Linux 下 C++编程。

(2) 使用 `epoll` 进行 IO 多路连接复用，由于第一次接触，在代码编写上出现了一定的问题，通过逐步调试解决了代码报错的问题。在使用 IO 多路复用，可能会遗漏某些错误处理逻辑，导致程序出现异常或者崩溃。在关键的操作（如 `epoll_create`、`epoll_ctl`、`epoll_wait` 等）后，及时检查返回值并处理可能的错误情况。使用日志系统记录错误信息，便于排查和调试问题。当程序中涉及多个文件描述符和不同类型的事件时，事件处理逻辑可能会变得复杂，难以管理和调试。可以将事件处理逻辑模块化，使用函数或类封装不同类型的事件处理逻辑，提高代码的可读性和维护性。

八、实验结论

通过本实验，我们深入学习了 RSA 算法的原理与应用，了解了 DES 加密算法的基本原理，以及如何结合 TCP 通信实现安全的通信系统。在实验中，我们成功地实现了 RSA 密钥对的生成和传输，DES 密钥的生成和加密通信，并使用了 I/O 多路复用技术来实现全双工通信，提高了通信效率和性能。同时，我们也遇到了一些问题，如密钥的安全性、全双工通信的实现等，但通过认真分析和解决，最终顺利完成了实验。这次实验使我们更加熟悉了加密算法和网络编程技术，为进一步深入学习和应用提供了良好的基础。