

网络安全技术

实 验 报 告

学 院 计算机学院
年 级 2017
班 级 1 班
学 号 1711425
姓 名 曹元议

2020 年 5 月 20 日

目录

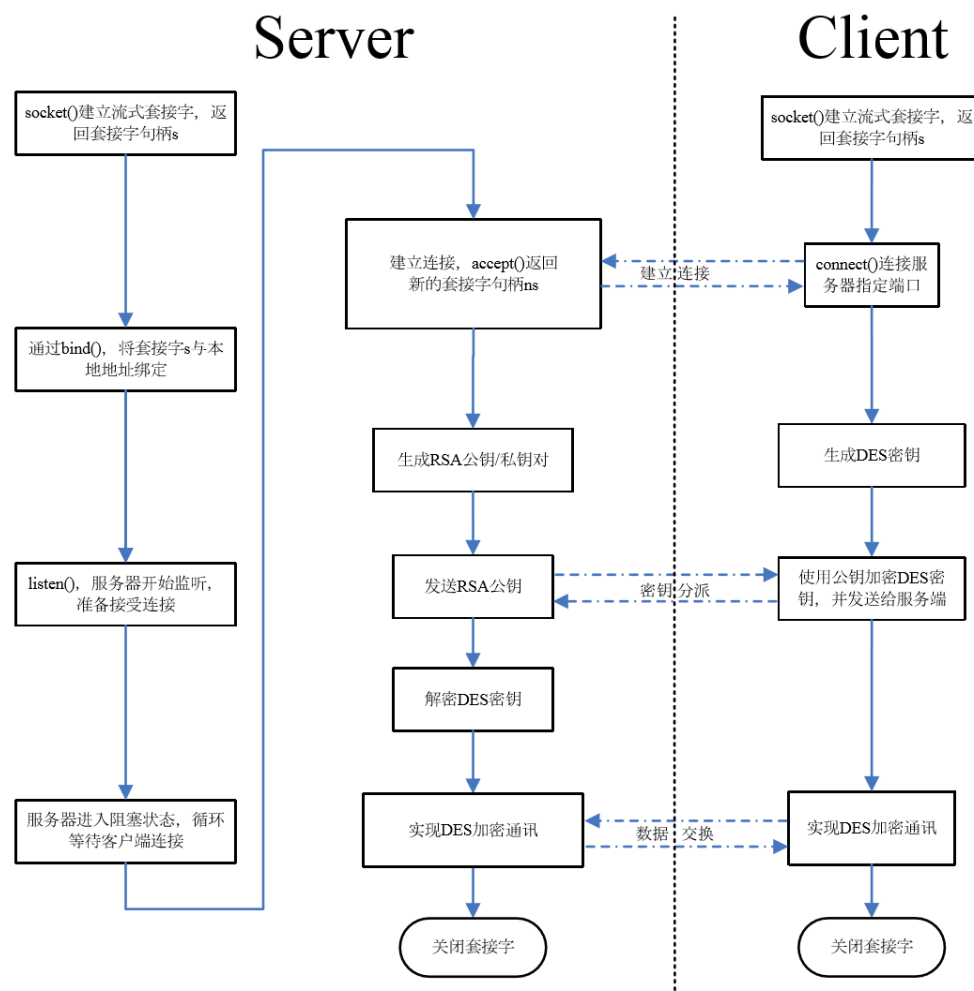
一、实验目的.....	1
二、实验内容.....	1
2.1 RSA 算法简述.....	2
2.1.1 RSA 加密解密的过程.....	3
2.1.2 产生大素数.....	4
2.1.3 快速模幂运算.....	4
2.1.3 欧几里得算法（辗转相除法）及扩展欧几里得算法.....	4
2.2 select 模型和异步 I/O 模型概述（以 Java 为例）.....	5
三、实验步骤及实验结果.....	8
3.1 RSA 算法的实现.....	9
3.1.1 快速模幂运算的实现.....	9
3.1.2 生成大素数的实现.....	10
3.1.3 RSA 公钥和私钥的生成.....	13
3.1.4 RSA 加密与解密.....	16
3.2 TCP 通信模块的改写.....	20
3.2.1 通信线程的建立和消息通信的实现.....	21
3.2.2 线程退出的实现.....	32
3.2.3 客户端断线重连的实现.....	34
3.3 实验结果展示.....	37
3.3.1 运行程序的方法.....	37
3.3.2 多线程聊天测试.....	39
四、实验遇到的问题及其解决方法.....	50
4.1 如果使用 BigInteger 类产生指定范围的随机数.....	50
4.2 如何提升实现的 RSA 算法的速度.....	51
五、实验结论.....	51

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 RSA 算法的基本编程方法
4. 了解异步 I/O 接口的基本工作原理

二、实验内容

本次实验的总体要求是基于上次实验的内容，编写 RSA 算法模块。实现 DES 密钥自动生成，并基于 RSA 算法进行密钥共享。程序执行的大致流程如下图所示：



从上图中概括出要点就是客户端向服务器发起连接之后，客户端先随机生成 DES 密钥，服务器先随机生成 RSA 密钥对，然后服务器将 RSA 公钥发送给客户端，

客户端使用服务器发来的 RSA 公钥加密随机生成的 DES 密钥发送给服务器端，然后服务器端使用自己的 RSA 私钥解密得到 DES 密钥，之后客户端和服务端通信就使用这个 DES 密钥加密通信。

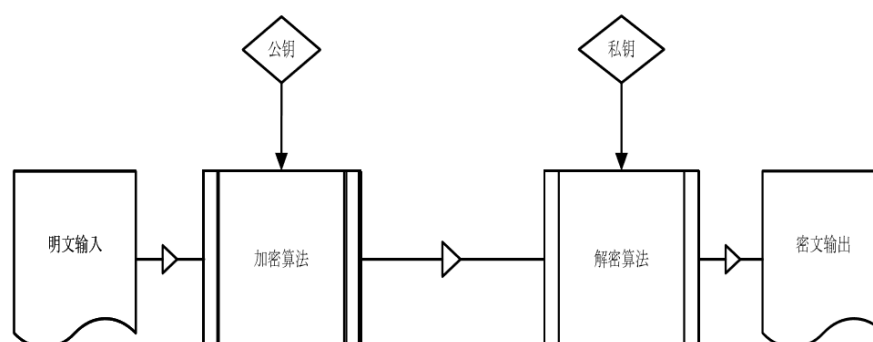
2.1 RSA 算法简述

RSA 加密算法是目前地球上最重要的加密算法，而且最安全的加密算法之一。它属于公钥密码体系（非对称密码体系）。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。上次实验使用到的 DES 算法就属于对称密码体系。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。

公钥密码体系由以下部分组成：

- (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：公钥和私钥总是成对出现，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。

下图可以概括出公钥密码体系的原理：



RSA 加密算法是一种典型的公钥加密算法。RSA 算法的原理其实比较简单，它就是基于模下指数幂运算，解密思想是寻求模下逆元素。其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。只要其密钥的长度足够长并且私钥不被泄露，那么用 RSA 加密的信息的安全性就可以保证。随着越来越长的整数被分解，人们普遍使用的 RSA 密钥的长度会越来越长，会影响 RSA 算法的实用性。因此其他的公钥密码体系也被提出，例如椭圆曲线公钥体系。人们也在努力寻找其他可替代 RSA 的公钥密码体系。

当然，RSA 算法有一个非常大的缺点，就是由于 RSA 进行的都是大数计算，使得 RSA 算法的速度比同样安全级别的对称密码算法慢很多（1/1000 左右），无论是软件还是硬件实现。所以 RSA 一般只用于少量数据加密。一般来说保密通信时一般还是使用较短的对称加密算法密钥、使用对称加密算法加密通信内容，使用 RSA 加密这个密钥。

2.1.1 RSA 加密解密的过程

例如甲乙双方要进行保密通信。

1. 首先，甲方随机选取两个大素数 p 和 q ，并计算 $n = p \cdot q$ 。
2. 然后甲方根据公式计算 $\phi(n) = (p-1) \cdot (q-1)$ 。其中 $\phi(n)$ 为欧拉函数，表示在所有小于 n 的正整数里，与 n 互素的整数个数。
3. 然后甲方随机生成正整数 e ，条件是 $1 < e < \phi(n)$ 且 $\gcd(e, \phi(n)) = 1$ 。
4. 最后甲方求出 e 在模 $\phi(n)$ 下的模反元素 d ，即 $ed \equiv 1 \pmod{\phi(n)}$ ，可以通过“扩展欧几里得算法”求解或者不断代入从小到大的一系列值尝试求解。因此模反元素 d 可以表示为： $d = e^{-1} \pmod{\phi(n)}$ 。需要注意这里的 -1 只是属于模反元素的表示法，不是模幂运算。
5. 甲方将 (e, n) 公开作为公钥，将 d 、 p 、 q 和 $\phi(n)$ 保密并用 (d, n) 作为私钥。
6. 乙方将明文 M 用甲方的公钥加密为密文 C 并送给甲方，那么 RSA 加密算法为（原则上 M 是正整数并且要小于 n ，可以通过获取明文字符串的编码值，例如 UTF-8 编码，得到数 M ）： $C = M^e \pmod{n}$
7. 甲方收到密文 C 后使用自己的私钥将其解密，RSA 解密算法为：

$$M = C^d \pmod{n}$$

2.1.2 产生大素数

上文没有提到随机大素数 p 和 q 应该如何生成。事实上精确判断这个数是否是素数的算法（例如素数筛法）对于大整数来说复杂度太高。因此在实际应用中一般使用概率算法来判断这个大整数是不是素数，最常用的概率算法是 Miller-Rabin 算法。Miller-Rabin 算法判断正奇数 n 是否是素数的过程如下：

1. 首先找到 k 和 q 满足： $n-1=2^k q$ ，其中 q 为奇数
2. 随机选取整数 a ，范围为 $1 < a < n-1$
3. 如果 $a^q \bmod n \neq 1$ 且对所有 $1 \leq j \leq k-1$ 有 $a^{2^j q} \bmod n \neq n-1$ ，则认为 n 是合数。
否则认为 n 是素数。

事实上，执行一次 Miller-Rabin 算法的误判概率为 $1/4$ 。所以需要多次执行该算法来降低误判概率，例如执行 m 次误判的概率为 $(1/4)^m$ 。例如令 $m=20$ ，误判概率为 $2^{-40} < 10^{-12}$ ，小到可以忽略不计。所以当 m 足够大是一直都没有判断 n 是合数，则 n 可以被认为是素数。因此这个算法还是可以基本满足快速产生大素数的需求。

2.1.3 快速模幂运算

要计算形如 $a^x \bmod n$ 的结果。如果按照普通的方法来计算，即先计算乘方后计算除法，一是乘方的结果会太大，而且乘法和除法的代价也会比较大。尤其是对于大整数来说时间开销相当大。因此对于 $n = b_k \dots b_1 b_0$ （ b_i 为 n 的二进制位， n 的二进制位的个数为 $k+1$ ）通常采用其他更快的算法来计算 $a^x \bmod n$ 的结果：（该算法的输出 g_0 即为 $a^x \bmod n$ ）

1. 令 $g_k = a$
2. 对于 $0 \leq i \leq k-1$ ，不断计算 g_i 如下：
$$g_i = \begin{cases} (g_{i+1} \cdot g_{i+1}) \bmod n & \text{if } b_i \neq 1 \\ (((g_{i+1} \cdot g_{i+1}) \bmod n) \cdot a) \bmod n & \text{else} \end{cases}$$

2.1.3 欧几里得算法（辗转相除法）及扩展欧几里得算法

欧几里得算法（辗转相除法）可以用来求两个数最大公约数，递归式如下：

$$\gcd(a,b) = \begin{cases} \gcd(b, a \bmod b) & b > 0 \\ a & b = 0 \end{cases}$$

扩展欧几里得算法可以用来求解形如 $ax + by = \gcd(a,b)$ 的二元一次方程的一组整数解。当然也可以直接用于求两个数的最大公约数。

上述 e 在模 $\phi(n)$ 下的模反元素 d 的条件是 $ed \equiv 1 \pmod{\phi(n)}$ ，可以化为 $(ed - 1) \bmod \phi(n) = 0$ ，即 $ed - k \cdot \phi(n) = 1$ 。也就是说要求出模反元素 d 需要求出以 d 和 k 为未知数的二元一次方程 $ed - k \cdot \phi(n) = 1$ 的整数解，并且只需要保留 d 的值即可。其中 1 就是 e 和 $\phi(n)$ 的最大公约数，一定是 1，否则方程无解。

在扩展欧几里得算法中， a 和 b 的求法保持原样。而在第 i 次递归的过程中， x 和 y 的求法如下（其中 x' 和 y' 为第 $i+1$ 次递归过程中 x 和 y 的值）：

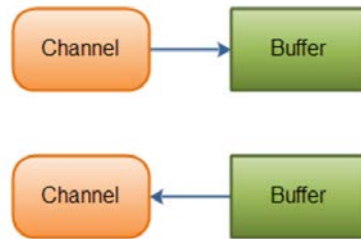
$$\begin{cases} x = y' \\ y = x' - \left\lfloor \frac{a}{b} \right\rfloor y' \end{cases}$$

在实验文档中，求模反元素 d 的方法是令方程 $ed - k \cdot \phi(n) = 1$ 的 $k = 1, k = 2, \dots$ 一直往上尝试可能的 k 值，直到满足 $(k \cdot \phi(n) + 1) \bmod e = 0$ 的条件为止，这样就找到了该方程的最大整数解。这样的效率相对来说还是比较低。

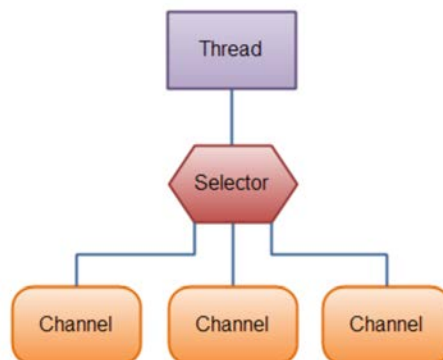
2.2 select 模型和异步 I/O 模型概述（以 Java 为例）

这次实验由于是对上一次实验的修改，因此我使用的编程语言还是 Java。实现 select 机制进行并行通信或者要使用异步 I/O（AIO）进行通信是本次实验的提高要求，如果在 Java 上实现的话可以使用 Java 的 NIO 框架（`java.nio.Non-blocking I/O`，非阻塞 I/O）。在本次的实验中，我使用了 Java 异步 I/O（AIO）改写了 TCP 通信的部分。

不同于 Java 的标准 I/O（`java.io`）面向字节流和字符流（并且从流中读出的所有字节都不被缓存），Java 的 NIO 中的所有 I/O 都是从 `channel`（通道）开始的，面向的是通道和缓冲区（例如 `ByteBuffer`），数据总是从 `channel` 中读到 `buffer` 内，或者从 `buffer` 缓冲区写入到 `channel` 中。一个网络连接也可以代表一个 `channel`。而且 `java.io` 中一个流只能读（输入流）或者写（输出流），是单向的；`java.nio` 的缓冲区既可以读又可以写。

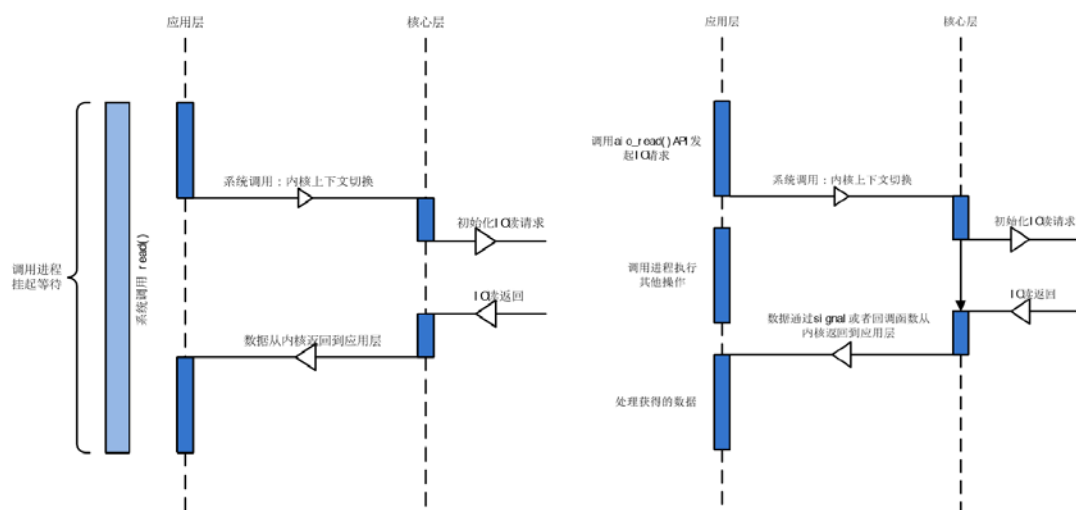


通过 `java.nio.channels.Selector`（I/O 多路复用器）可以实现 `select` 机制，可以同时管理多个 `channel` 上的 I/O 操作，如此可以实现单线程管理多个 `channel`，也就是可以管理多个网络连接。原理图如下：



如果要使用 `Selector`，必须要先把 `Channel` 注册到 `Selector` 上，然后就可以调用 `Selector` 的 `select()` 方法。这个方法会进入阻塞，直到有一个 `channel` 的处于就绪状态。当方法返回后，线程可以通过轮询每个 `channel` 处理这些 `channel` 对应的事件。在这里使用到的 `channel` 定义在包 `java.nio.channels` 中，`SocketChannel` 对应客户端的 `Socket` 类、`ServerSocketChannel` 对应服务器端的监听 `ServerSocket` 类。

同步 I/O 通信和异步 I/O 通信的对比图如下（左：同步 I/O 通信，右：异步 I/O 通信）：



可以发现，如果是同步 I/O，在应用进程进行读写系统调用的时候，会进行内核上下文切换到操作系统内核代码，内核执行完之后才返回到应用进程，在内核执行相关代码的时候，在此期间应用进程被阻塞等待直到内核执行完返回。而异步 I/O 与同步 I/O 不同之处是内核执行代码的时候，应用进程不被阻塞，还可以执行其他代码，然后内核执行完相应代码时会通过信号通知给应用进程或者通过自定义的回调函数返回到应用进程。

在包 `java.nio.channels` 定义的两个通道用于异步网络通信：

AsynchronousSocketChannel：客户端异步 `socket`（对应 `Socket` 类）；

AsynchronousServerSocketChannel：服务器异步 `socket`（对应 `ServerSocket` 类）。

上次使用的对应于客户端的 `Socket` 类和服务端监听的 `ServerSocket` 都是同步的 `socket`，需要通过实际的流进行数据读写。而使用了异步的 `channel` 之后，对数据流的读写转为对数据缓冲区（`ByteBuffer`）的读写，而且针对与读、写、客户端连接服务器、服务器接受客户端的连接都要定义相应的回调函数（通过实现 `CompletionHandler` 接口并定义里面的函数 `completed()` 和 `failed()`，如果操作执行成功则回调 `completed()`，否则回调 `failed()`）。

使用了 Java 的 NIO 框架之后，客户端程序的工作过程包含以下基本的步骤：

1. 通过 `SocketChannel` 连接到指定服务端的 IP 地址的端口号（`InetSocketAddress`）的服务器。若服务器端响应，则建立客户端到服务器的通信线路。若连接失败，会出现异常或者回调相应的 `failed()` 函数。
2. 创建读数据/写数据缓冲区对象来读取服务端数据或向服务端发送数据。
3. 关闭 `SocketChannel`：断开客户端到服务器的连接。

服务器端程序的工作过程包含以下基本的步骤：

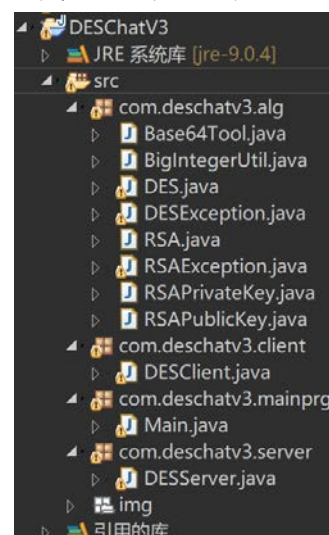
1. 通过 `ServerSocketChannel` 绑定 IP 地址和端口号，用于监听客户端的连接请求。
2. 通过 `ServerSocketChannelImpl` 的 `accept()` 方法监听连接请求：如果客户端请求连接，则接受连接，创建一个与该客户端通信的 `SocketChannel` 对象。
3. 创建读数据/写数据缓冲区对象来读取客户端数据或向客户端发送数据。
4. 某客户端访问结束，就关闭 `SocketChannel` 对象，这样服务器与该客户端的连接断开。
5. 如果不再接收任何客户端的连接的话，调用 `close()` 关闭 `ServerSocketChannel`。

三、实验步骤及实验结果

本次实验的步骤为：编写代码→测试实验结果。

本次实验的编程部分主要分为两个步骤：DES 算法的实现、改写 TCP 通信模块。和上次一样，为了不让输出的结果过于凌乱，我采用了【Swing + 命令行 + 文件流】的方式实现。Swing 用来实现可视化界面（在这次实验中并未改动上次实验的可视化界面中控件的布局），可视化界面包括客户端和服务器的通信日志和消息的发送；命令行用于调试，并显示 DES 加密和解密的输入和输出；Java 的错误流输出到错误日志文件流中，可由错误日志文件查看异常堆栈信息。

这里主要也是展示与原理相关的代码，报告中没有展示的代码可以查阅项目文件的源代码。与可视化界面相关的代码也会尽量省略。项目文件的源代码组织如下：



至于项目命名为 DESChatV3 而不是 DESChatV2 的原因：DESChatV3 是在 DESChatV2 的基础上将 TCP 通信模块改成了异步 I/O 实现，本意上是如果这次实验的截止时间前没有实现异步通信，则使用 DESChatV2 提交。DESChatV2 是这次在雨课堂提交的 DESChatV3 的备份项目，是只实现了本次最基本要求的项目，没有改动 TCP 通信的框架。DESChatV2 是在 DESChat（上次实验在雨课堂上提交的项目）的基础上，增加了 RSA 算法模块，并增加了 TCP 通信过程中 RSA 公钥的共享。因此 DESChatV2 的代码不再展示和提交，源代码可以去 GitHub 仓库中查阅：<https://github.com/LoveBettygirl/tcp-des-chat>

这次实验虽然没有改动可视化界面中控件的布局，但是适当调整了在可视化界面的显示：不在客户端界面上显示 DES 密钥，转为在命令行中显示，防止攻击者越过 RSA 密钥直接知道了 DES 密钥；客户端未连接时在显示服务器 IP 的位置显示“未连接到客户端”。

3.1 RSA 算法的实现

和上次实验不同的是，这次实验将 DES 算法和 RSA 算法相关的模块整合到了 `com.deschatv3.alg` 中。

在包 `com.deschatv3.alg` 中，类 `RSA` 用于 RSA 算法的实现，这次的实现参考了实验文档中部分实现思路。由于 RSA 需要大数计算，因此 RSA 算法的实现借助了 `java.math.BigInteger` 类，这是 Java 自带的大数类，可以实现任意长度的大数计算。这样，我在本次实验中可以产生安全性与当前主流的 1024 bits 长度的密钥相当的 RSA 密钥。当然，这个类就本身已经实现了很多为 RSA 算法服务的函数，例如生成随机大素数的 `probablePrime()` 函数以及实现模幂运算的 `modPow()` 函数。由于实验目的本身就是为了学习 RSA 算法从里到外的原理，因此这里的实现不使用这些为 RSA 算法服务的函数，会将模幂运算和生成大素数的过程都重新实现一遍。

3.1.1 快速模幂运算的实现

类 `RSA` 的 `modPow()` 函数用于进行快速模幂运算，根据第 2.1.3 节的算法实现。`modPow()` 函数如下：

```
1 /**
```

```

2  * 快速模幂运算
3  * @param base 被除数底数
4  * @param index 被除数指数
5  * @param mod 除数
6  * @return 模除运算结果
7  */
8  private static BigInteger modPow(BigInteger base, BigInteger index,
    BigInteger mod) {
9      String indexBinary = index.toString(2);
10     BigInteger result = new BigInteger(base.toString(10), 10);
11     for (int i = 1 ; i < indexBinary.length(); i++) {
12         result = result.multiply(result).mod(mod);
13         if (indexBinary.charAt(i) == '1')
14             result = result.multiply(base).mod(mod);
15     }
16     return result;
17 }

```

我将该函数标记为 **static** 函数,是为了在静态和非静态函数中都能使用此函数。

3.1.2 生成大素数的实现

首先, `milllerRabin()` 函数是执行一次 Miller-Rabin 算法的函数, 根据 2.1.2 节的 Miller-Rabin 算法实现。

```

1  /**
2   * 通过 Miller-Rabin 算法测试一个正奇数是不是素数 (误判概率为 1/4)
3   * @param n 要测试的正奇数
4   * @return true 则 n 被认为是素数, 否则为合数
5   */
6  private boolean millerRabin(BigInteger n) {
7      if (!n.testBit(0))
8          throw new RSAException("n must > 0 and must be an odd number");
9      // 首先要找出 q 和 k 使得 n-1=2^k*q
10     BigInteger q = n.subtract(BigInteger.ONE), k = new BigInteger("0");
11     // q 不断除以 2 (右移 1 次) 直到结果为奇数,
12     // 右移的次数就是 k
13     while (!q.testBit(0)) {
14         k = k.add(BigInteger.ONE);
15         q = q.shiftRight(BigInteger.ONE.intValue());
16     }
17     // 随机选取的 a > 1 且 a < n - 1

```

```

18     /*BigInteger a = new BigDecimal(Math.random()).multiply(new
    BigDecimal(n.subtract(new BigInteger("3"))))
19         .toBigInteger().add(BigInteger.TWO);*/
20     BigInteger a = genRandom(BigInteger.ONE, n.subtract(BigInteger.ONE));
21     // 测试  $a^q \pmod n$  的值是否为1, 为1 则是素数
22     if (modPow(a, q, n).compareTo(BigInteger.ONE) == 0)
23         return true;
24     // 测试  $a^{(2^j \cdot q)} \pmod n$  的值是否为  $n-1$  ( $j \geq 1$  且  $j \leq k-1$ ), 有一个是则认为是素数
25     for (BigInteger j = new BigInteger("1"); j.compareTo(k) < 0; j =
    j.add(BigInteger.ONE)) {
26         q = q.shiftLeft(BigInteger.ONE.intValue());
27         if (modPow(a, q, n).compareTo(n.subtract(BigInteger.ONE)) == 0)
28             return true;
29     }
30     return false;
31 }

```

genRandom 是生成范围为(min,max)的 BigInteger 随机数, 思路是产生 $[0, 2^{\text{bitLen}-1}]$ 的随机数, 然后检查是不是在(min,max)的范围内, 如果不是就重新生成直到在(min,max)的范围内为止。这么实现的原因见 4.1 节。

```

1  /**
2   * 生成 BigInteger 类型的随机整数, 范围: (min, max)
3   * @param min 随机整数下界
4   * @param max 随机整数上界
5   * @return 生成的随机整数
6   */
7  private BigInteger genRandom(BigInteger min, BigInteger max) {
8      BigInteger b;
9      do {
10         b = new BigInteger(max.bitLength(), r);
11     } while (b.compareTo(min) <= 0 || b.compareTo(max) >= 0);
12     return b;
13 }

```

由于一次执行的误判概率还是比较高, 因此要通过 testPrime() 函数执行多次 millerRabin() 函数来使误判概率降低到可忽略的程度上。

```

1  /**
2   * 多次执行 Miller-Rabin 算法测试一个正奇数是不是素数
3   * @param n 要测试的正奇数
4   * @return true 则 n 被认为是素数, 否则为合数
5   */
6  private boolean testPrime(BigInteger n) {

```

```

7     int loop = getLoop();
8     for (int i = 0; i < loop; i++) {
9         if (!millerRabin(n))
10            return false;
11     }
12     return true;
13 }

```

要执行的次数由 `getLoop()` 函数获得，通过该数的长度来决定执行 `millerRabin()` 函数的次数。这段代码来自 JDK 1.8 中 `BigInteger` 类的实现(严格意义上在这里使用这段代码是不对的，使用这段代码的原因见第 4.2 节)：

```

1  /**
2   * 获取不同 bitLen 对应的应执行 Miller-Rabin 算法的次数
3   * @return 应执行 Miller-Rabin 算法的次数
4   */
5  private int getLoop() {
6      int rounds;
7      if (bitLen < 100) {
8          rounds = 50;
9      } else if (bitLen < 256) {
10         rounds = 27;
11     } else if (bitLen < 512) {
12         rounds = 15;
13     } else if (bitLen < 768) {
14         rounds = 8;
15     } else if (bitLen < 1024) {
16         rounds = 4;
17     } else {
18         rounds = 2;
19     }
20     return rounds;
21 }

```

最终定义 `randomPrime()` 函数来生成随机的大素数：

```

1  /**
2   * 随机生成大素数
3   * @return 随机生成的大素数
4   */
5  private BigInteger randomPrime() {
6      BigInteger random;
7      do {
8          random = new BigInteger(bitLen, r);
9          random.setBit(bitLen - 1); // 设置生成的随机数最高位为1，保证足够大

```

```

10     if (!random.testBit(0)) { // 保证这个数为奇数
11         random = random.setBit(0);
12     }
13     } while(!testPrime(random));
14     return random;
15 }

```

首先利用 `BigInteger` 类的构造函数随机产生一个范围为 $[0, 2^{\text{bitLen}} - 1]$ 的随机数, 然后再设置它的最高位为 1 使得这个数足够大并设置它的最低位为 1 保证它是奇数, 最后对这个数执行 `testPrime()` 测试是不是素数。如果不是, 就重复这个过程直到产生的数是素数为止。

3.1.3 RSA 公钥和私钥的生成

在类 `RSA` 的构造函数中生成 `RSA` 的公钥和私钥, 过程基于 2.1.1 节的内容:

```

1  /**
2   * 构造方法
3   * @param bitLen p 和 q 的长度
4   */
5  public RSA(int bitLen) {
6      this.bitLen = bitLen;
7      p = randomPrime();
8      q = randomPrime();
9      n = p.multiply(q);
10     euler =
        p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
11     do {
12         e = genRandom(BigInteger.ONE, euler);
13     } while(gcd(e, euler).compareTo(BigInteger.ONE) != 0);
14     // 求 e 对 euler 的逆元 d
15     d = calculateD(e, euler);
16     publicKey = new RSAPublicKey(e, n);
17     privateKey = new RSAPrivateKey(d, n);
18 }

```

其中 `euler` 是欧拉函数 $\phi(n) = (p-1) \cdot (q-1)$ 。生成的 `e` 一定要同时满足这两个条件: $1 < e < \phi(n)$ 且 $\text{gcd}(e, \phi(n)) = 1$, 这里采用的方法是生成范围是 $1 < e < \text{euler}$ 随机数, 然后在检查 `e` 和 `euler` 的最大公约数是否为 1, 如果不为 1 则重复这个过程。以下是使用非递归的辗转相除法计算两个数的最大公约数:

```

1  /**

```

```

2  * 辗转相除法求最大公约数
3  * @param a 被除数
4  * @param b 除数
5  * @return 最大公约数
6  */
7  private BigInteger gcd(BigInteger a, BigInteger b) {
8      if (b.compareTo(BigInteger.ZERO) == 0)
9          return a;
10     BigInteger mod = a.mod(b);
11     while (mod.compareTo(BigInteger.ZERO) != 0) {
12         a = b;
13         b = mod;
14         mod = a.mod(b);
15     }
16     return b;
17 }

```

对于 d 的求法没有使用实验文档中提供的方法，而是使用 2.1.3 节的扩展欧几里得算法实现：

```

1  /**
2   * 欧几里得扩展算法
3   * @param a 被除数
4   * @param b 除数
5   * @return 最大公约数
6   */
7  private BigInteger exgcd(BigInteger a, BigInteger b){
8      if(b.compareTo(BigInteger.ZERO) == 0) {
9          x = new BigInteger("1");
10         y = new BigInteger("0");
11         return a;
12     }
13     BigInteger result = exgcd(b,a.mod(b));
14     BigInteger temp = x;
15     x = y;
16     y = temp.subtract(a.divide(b).multiply(y));
17     return result;
18 }
19
20 /**
21  * 求  $e$  的反模  $d$ （找到扩展欧几里得算法求出的最小正整数解  $x$ ）
22  * @param a 对应  $e$ 
23  * @param k 对应欧拉函数  $euler$ 

```



```

24  * @return e 的反模 d
25  */
26  private BigInteger calculateD(BigInteger a, BigInteger k){
27      BigInteger d = exgcd(a, k);
28      // 判断最大公约数是否为1，否则无解
29      if(d.compareTo(BigInteger.ONE) == 0) {
30          return x.mod(k.abs()); // 求出的 x 可能为负，需要转为最小正整数解
31      }
32      else
33          return new BigInteger("-1");
34  }

```

注意由于通过扩展欧几里得算法求出的 x 可能为负值，需要将其转变为范围为 $0 \sim |k|$ 的正整数。`BigInteger` 类的 `mod()` 函数为取模运算，如果取余运算的结果为负值会在这个负值的基础上加上除数使最终结果大于 0 并且小于除数，需要保证传入的除数值也要大于 0 。

为了方便起见，我在包 `com.deschatv3.alg` 中又定义了两个类：`RSAPublicKey` 和 `RSAPrivateKey`，分别代表 RSA 的公钥的私钥：

```

1  public class RSAPublicKey implements Serializable {
2
3      private static final long serialVersionUID = -2071297560602018291L;
4      private BigInteger e;
5      private BigInteger n;
6
7      public RSAPublicKey(BigInteger e, BigInteger n) {
8          this.e = e;
9          this.n = n;
10     }
11
12     public BigInteger getE() {
13         return e;
14     }
15
16     public BigInteger getN() {
17         return n;
18     }
19
20     @Override
21     public String toString() {
22         return "(" + e + ", " + n + ")";
23     }
24 }

```

```

1  public class RSAPrivateKey implements Serializable {
2
3      private static final long serialVersionUID = -7021466273660477038L;
4      private BigInteger d;
5      private BigInteger n;
6
7      public RSAPrivateKey(BigInteger d, BigInteger n) {
8          this.d = d;
9          this.n = n;
10     }
11
12     public BigInteger getD() {
13         return d;
14     }
15
16     public BigInteger getN() {
17         return n;
18     }
19
20     @Override
21     public String toString() {
22         return "(" + d + ", " + n + ")";
23     }
24 }

```

3.1.4 RSA 加密与解密

RSA 加密函数 `encry()` 和解密函数 `decry()` 分别定义如下：

```

1  /**
2   * 对明文进行一次加密
3   * @param src 要加密的明文
4   * @param k 加密使用的公钥
5   * @return 加密得到的密文的十六进制表示
6   */
7  public static String encry(String src, RSAPublicKey k) {
8      BigInteger integer = BigIntegerUtil.string2BigInteger(src);
9      if (integer.compareTo(k.getN()) >= 0)
10         throw new RSAException("Data must not be larger than n");
11     integer = modPow(integer, k.getE(), k.getN());

```

```

12     return integer.toString(16);
13 }
14
15 /**
16  * 对密文进行一次解密
17  * @param src 要解密的密文的十六进制表示
18  * @param k 解密使用的私钥
19  * @return 解密得到的明文
20  */
21 public static String decry(String src, RSAPrivateKey k) {
22     BigInteger integer = new BigInteger(src, 16);
23     integer = modPow(integer, k.getD(), k.getN());
24     return BigIntegerUtil.bigInteger2String(integer,
        integer.bitLength());
25 }

```

其中，在 `encry()` 函数中要检查明文代表的数的大小是否小于 `n`，否则将抛出异常。由于在本次实验中要使用 RSA 加密的仅仅只是 64 位的 DES 密钥，如果怕不够可以通过增加构造函数的参数 `bitLen` 来增加密钥长度，这样就可以一次加密 64 为 DES 密钥了；而且 `javax.crypto.Cipher` 的 RSA 加密解密的实现也是没有进行分段加密的。因此我没有考虑将原文进行分段加密和解密的问题。

在包 `com.deschatv3.alg` 中又在 `BigIntegerUtil` 类，其中定义的 `string2BigInteger()` 和 `bigInteger2String()` 起到了 `String` 字符串值和 `BigInteger` 之间的桥梁作用，这两个函数和相关的辅助函数定义如下：

```

1  public class BigIntegerUtil {
2
3      /**
4       * 将Byte 数组转换成byte 数组
5       * @param B 要转换的Byte 数组
6       * @return 转换得到的byte 数组
7       */
8      private static byte[] Byte2byte(Byte[] B) {
9          byte[] b=new byte[B.length];
10         for(int i=0;i<b.length;i++) {
11             b[i]=B[i];
12         }
13         return b;
14     }
15
16     /**
17     * 二进制字符串转普通字符串
18     * @param src 要转换的字符串

```

```

19     * @return 转换结果
20     */
21     public static String binary2String(String src) {
22         List<Byte> list=new ArrayList<Byte>();
23         for(int i=0;i<src.length();i+=8) {
24             list.add((byte)(Integer.parseInt(src.substring(i,i+8),2)));
25         }
26         byte[] b=Byte2byte(list.toArray(new Byte[list.size()]));
27         String s=null;
28         try {
29             s = new String(b,"utf-8");
30         } catch (UnsupportedEncodingException e) {
31             // TODO 自动生成的 catch 块
32             e.printStackTrace();
33         }
34         return s;
35     }
36
37     /**
38      * 将一个int 数字转换为二进制的字符串形式。
39      * @param num 需要转换的int 类型数据
40      * @param digits 要转换的二进制位数，位数不足则在前面补0
41      * @return 二进制的字符串形式
42      */
43     public static String toBinary(int num, int digits) {
44         String s=Integer.toBinaryString(num);
45         if(s.length()<digits) {
46             String cover = Integer.toBinaryString(1 <<
digits).substring(1);
47             return cover.substring(s.length()) + s;
48         }
49         else if(s.length()>digits) {
50             return s.substring(s.length() - digits);
51         }
52         else {
53             return s;
54         }
55     }
56
57     /**
58      * 将String 的字符串值转为BigInteger 值
59      * @param src 输入的字符串
60      * @return 对应的BigInteger 值（使用utf-8 编码）
61      */

```

```

62     public static BigInteger string2BigInteger(String src) {
63         byte[] b=null;
64         try {
65             b=src.getBytes("utf-8");
66         } catch (UnsupportedEncodingException e) {
67             // TODO 自动生成的 catch 块
68             e.printStackTrace();
69         }
70         StringBuilder s = new StringBuilder();
71         for (int i = 0; i < b.length; i++) {
72             s.append(toBinary(b[i], 8));
73         }
74         return new BigInteger(s.toString(), 2);
75     }
76
77     /**
78      * 将BigInteger 的值转为对应的字符串
79      * @param integer
80      * @param bitLen
81      * @return
82      */
83     public static String bigInteger2String(BigInteger integer, int bitLen)
84     {
85         bitLen = bitLen % 8 == 0 ? bitLen : (bitLen / 8 + 1) * 8;
86         String binary = integer.toString(2);
87         String zero = "";
88         // 如果二进制位长度小于bitLen 则在前面补0, 否则取最后的bitLen 位
89         if (binary.length() < bitLen) {
90             for (int i = binary.length(); i < bitLen; i++) {
91                 zero += "0";
92             }
93             binary = zero + binary;
94         }
95         else if (binary.length() > bitLen) {
96             binary = binary.substring(binary.length() - bitLen);
97         }
98         return binary2String(binary);
99     }
100
101     public static void main(String[] args) {
102         // TODO 自动生成的方法存根
103
104         System.out.println(BigIntegerUtil.string2BigInteger("ABCDEFGH").toString(2));

```

```
103     }  
104  
105 }
```

这里面的 `string2BigInteger()` 和 `bigInteger2String()` 使用到的辅助函数来自上次在 `DES` 类中定义的辅助函数，在这次直接将 `DES` 类中的这些辅助函数定义都移动到了这个类中，使得辅助函数尽可能与算法的主体实现解耦。

3.2 TCP 通信模块的改写

这次我选择将 TCP 通信模块改写成异步 I/O 通信，并且实现出和上次实验类似的效果。

由于这次将同步 I/O 通信改写成了异步 I/O 通信，因此与 TCP 通信相关的功能都可以集中在客户端或服务端的工作线程中。服务器也可以只使用一个线程管理多个网络连接，不用像上次那样单独为一个网络连接创建一个单独的线程，这样就减少了服务器端的线程开销（因为回调的 `completed()` 函数或者 `failed()` 函数的参数会携带相应的客户端信息）。由于我使用了可视化界面，如果是发送消息也不用像上次一样复用主界面线程来发送消息了，可以从界面中通过 `sendMsg()` 函数的参数传递给工作线程。

客户端的相关实现在包 `com.deschatv3.client` 中，其中 `public` 类 `DESCClient` 是客户端的主界面，`private` 内部类 `AsyncClientHandler` 是客户端与服务器通信的线程类（也包含发起与服务器的连接之后的回调函数，是下面所说的客户端线程），`private` 内部类 `ClientReadHandler` 含有客户端读操作的回调函数，`private` 内部类 `ClientWriteHandler` 含有客户端进行写操作的回调函数；服务器的相关实现在 `com.deschatv3.server` 中，其中 `public` 类 `DESServer` 是服务器的主界面，`private` 内部类 `AsyncServerHandler` 是服务器监听以及服务器与客户端通信的线程类，`private` 内部类 `AcceptHandler` 含有服务器接受来自客户端连接的回调函数，`private` 内部类 `ServerReadHandler` 含有服务器进行读操作的回调函数，`private` 内部类 `ServerWriteHandler` 含有服务器进行写操作的回调函数。建立客户端或服务端线程的代码在 `com.deschatv3.mainprg` 中的类 `Main`（只展示和上次实验不同的地方）。

实现的大体效果也是客户端能和服务器进行多对一通信（不同客户端的端口号不

同且随机)，客户端和服务端任何一方都可以随时给对方发送消息而不用等待对方回复（我上次实验忘记说明并演示这个了），服务器可以选择任何一个与其连接的客户端发送消息，这和上次实验的效果是一致的。按照实验要求，每个客户端首先需要随机生成 DES 密钥，服务器首先要随机生成 RSA 密钥对，服务器将生成号的 RSA 公钥发送给客户端，客户端用 RSA 公钥将 DES 密钥加密发给服务器，服务器使用 RSA 私钥解密这个 DES 密钥，双方使用这个 DES 密钥进行通信。服务器为每个客户端产生的 RSA 密钥对都不同。

本次实验中使用的 RSA 类的实例，传入的 bitLen 参数为 512，产生的密钥长度在 1024 bit 附近。

按照实验要求，这次把上次在服务器端随机产生 DES 密钥的函数移到了客户端（在 DESCClient 的内部类 AsyncClientHandler 中）：

```
1  /**
2   * 生成随机 DES 密钥
3   */
4  private void genRandomDESKey() {
5      StringBuilder s=new StringBuilder();
6      String alpha="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7      Random r = new Random();
8      for(int i=0;i<8;i++) {
9          s.append(alpha.charAt(r.nextInt(26)));
10     }
11     key=s.toString();
12 }
```

3.2.1 通信线程的建立和消息通信的实现

客户端线程建立的代码在 Main 类中和上次不同的地方是，这次将连接的代码集成到了 DESCClient 类中，不需要在 Main 类中使用额外的代码建立，如下图所示：

```
System.out.println(serverIP);
if(serverIP!=null) {
    //建立套接字并和服务器连接
    DESCClient clientDlg=new DESCClient(serverIP,2000); //连接成功后建立客户端线程并打开对话框
    /*EventQueue.invokeLater(new Runnable() {
        @Override
        public void run() {
            clientDlg.setVisible(true);
        }
    });*/
}
```

在 `AsyncClientHandler` 类的构造函数中打开客户端与服务器的信道：

```
1 public AsyncClientHandler(String serverIP, int port) {
2     this.serverIP = serverIP;
3     this.port = port;
4     genRandomDESKey();
5     try {
6         // 创建异步的客户端通道
7         clientChannel = AsynchronousSocketChannel.open();
8     } catch (IOException e) {
9         e.printStackTrace();
10    }
11 }
```

在 `AsyncClientHandler` 类中的客户端线程的函数中，发起与服务器的连接。这里选择使用随机数产生端口号，由于端口号可能被系统的其他程序占用，因此要不断产生新的端口号直到绑定成功为止：（使用 `CountDownLatch` 的原因是阻塞客户端线程，让客户端线程的函数不再继续往下执行直到线程退出，线程退出会导致程序执行时程序意外退出。但是读写等操作的处理也在这个线程中，因为都是由 `connect()` 引起的，起到和主界面线程同步的作用）

```
1 @Override
2 public void run() {
3     // 创建 CountDownLatch 等待
4     latch = new CountDownLatch(1);
5     Random r = new Random();
6     int newPort = -1;
7     // 发起异步连接操作，回调参数就是这个类本身，如果连接成功会回调 completed
    方法
8     while(true) {
9         try {
10             newPort = r.nextInt(65536);
11             clientChannel.bind(new InetSocketAddress("127.0.0.1",
12                 newPort));
13         } catch (IOException e2) {
14             // TODO 自动生成的 catch 块
15             newPort = -1;
16         }
17         if (newPort != -1) {
18             break;
19         }
20     }
21     clientChannel.connect(new InetSocketAddress(serverIP, port), this,
22         this);
```



```

21     try {
22         latch.await();
23     } catch (InterruptedException e1) {
24         e1.printStackTrace();
25     }
26     try {
27         clientChannel.close();
28     } catch (IOException e) {
29         e.printStackTrace();
30     } finally {
31         if (exit)
32             System.exit(0);
33     }
34 }

```

如果连接成功，将回调 `AsyncClientHandler` 类的 `completed()` 函数。该函数将打开客户端的可视化界面，然后分配缓冲区，进行读操作，准备接收来自服务器端的 RSA 公钥：

```

1     @Override
2     public void completed(Void result, AsyncClientHandler attachment) {
3         isconn = true;
4         try {
5             address = clientChannel.getLocalAddress();
6         } catch (IOException e) {
7             // TODO 自动生成的 catch 块
8             e.printStackTrace();
9         }
10        updateInfo();
11        EventQueue.invokeLater(new Runnable() {
12            @Override
13            public void run() {
14                DESClient.this.setVisible(true);
15            }
16        });
17        ByteBuffer readBuffer = ByteBuffer.allocate(1024);
18        clientChannel.read(readBuffer, readBuffer, new
19            ClientReadHandler(clientChannel, latch));

```

如果连接失败，将回调 `AsyncClientHandler` 类的 `failed()` 函数。这里先截取一部分代码，其他未展示的代码稍后再解释。这部分代码会打印错误堆栈信息到文件中，然后弹出失败对话框，关闭信道，终止程序。

```

1      @Override
2      public void failed(Throwable exc, AsyncClientHandler attachment) {
3          exc.printStackTrace();
4          if (!reconn) {
5              System.out.println("Connect failed: "+exc.getMessage());
6              JOptionPane.showMessageDialog(null, "Connect failed:
"+exc.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
7              try {
8                  clientChannel.close();
9                  latch.countDown();
10             } catch (IOException e) {
11                 e.printStackTrace();
12             } finally {
13                 System.exit(0);
14             }
15         }

```

服务器线程的建立在 **Main** 中的代码和上次一样。直接打开对话框并建立监听线程即可，不再展示出来。

AsyncServerHandler 类的构造函数和服务器线程的函数如下。在构造函数中建立服务器端的信道，将默认 IP 地址和端口号绑定。在服务器线程的函数中，接收来自客户端的第一个连接：

```

1      public AsyncServerHandler(int port) {
2          this.port = port;
3          try {
4              //创建服务端通道
5              channel = AsynchronousServerSocketChannel.open();
6              //绑定端口
7              channel.bind(new InetSocketAddress(port));
8          } catch (IOException e) {
9              e.printStackTrace();
10         }
11     }
12     @Override
13     public void run() {
14         //CountDownLatch 可以允许当前的现场一直阻塞，防止在执行过程中意外退出，
15         //起到线程同步的作用
16         System.out.println("Server Started");
17         latch = new CountDownLatch(1);
18         updateListenState();
19         //用于接收客户端的连接
20         channel.accept(this,new AcceptorHandler());
21         try {

```

```

22         latch.await();
23     } catch (InterruptedException e) {
24         e.printStackTrace();
25     } finally {
26         if(exit)
27             System.exit(0);
28     }
29 }

```

如果有来自客户端的连接，就回调 **AcceptHandler** 的 **completed()**函数，加入新的客户端通道到服务器维护的客户端列表中，继续接受新的连接，并对这一连接的客户端生成 **RSA** 密钥对并发送公钥的参数 **e**。如果有新的连接就再回调这个 **completed()**函数。

```

1     @Override
2     public void completed(AsynchronousSocketChannel channel,
3         AsyncServerHandler serverHandler) {
4         if (count > 50) {
5             JOptionPane.showMessageDialog(null, "The count of clients
6             cannot be more than 50!", "Error", JOptionPane.ERROR_MESSAGE);
7             return;
8         }
9         count++;
10        index++;
11        String name = "client_"+index;
12        Client client = new Client(name, channel);
13        clients.put("client_"+index, client);
14        addClient(channel);
15        serverHandler.channel.accept(serverHandler, this); //继续接受其他
16        客户端的请求
17        //聊天前首先向客户端发送 RSA 公钥
18        client.prepare();
19        if(!client.makeRSAKey) {
20            client.genRSAPublicKey();
21            client.rsa = new RSA(512);
22            RSAPublicKey publicKey = client.rsa.getRSAPublicKey();
23            String e = publicKey.getE().toString();
24            serverHandle.sendMsg(e, client);
25        }
26        else {
27            ByteBuffer buffer = ByteBuffer.allocate(1024);
28            client.channel.read(buffer, buffer, new
29            ServerReadHandler(client));
30        }
31    }

```

```
27     }
```

客户端将等待（不是真正的等待）服务器端发送 RSA 公钥的参数 **e**，如果客户端接收到 RSA 公钥的参数 **e**，将回调 **ClientReadHandler** 类的 **completed()** 函数。之后将再次进行读操作，准备接收 RSA 公钥的参数 **n**。

```
1     @Override
2     public void completed(Integer result, ByteBuffer buffer) {
3         buffer.flip();
4         byte[] bytes = new byte[buffer.remaining()];
5         buffer.get(bytes);
6         String body;
7         try {
8             body = new String(bytes, "UTF-8");
9             if (!clientHandle.geteofRSAPublicKey) {
10                clientHandle.rcvingRSAPublicKey();
11                clientHandle.geteofRSAPublicKey = true;
12                clientHandle.e_ = body;
13                ByteBuffer readBuffer = ByteBuffer.allocate(1024);
14                clientChannel.read(readBuffer, readBuffer, new
15                ClientReadHandler(clientChannel, latch));
16            }
17            else if (!clientHandle.getnofRSAPublicKey) {
18                .....
19            }
20        }
```

服务器端发送的 RSA 公钥的参数 **e**，客户端成功接收之后，将回调 **ServerWriteHandler** 的 **completed()** 函数，继续使用写操作发送 RSA 公钥的参数 **n**：

```
1     @Override
2     public void completed(Integer result, ByteBuffer buffer) {
3         // 如果没有发送完，就继续发送直到完成
4         if (buffer.hasRemaining())
5             client.channel.write(buffer, buffer, this);
6         else{
7             if(!client.makeRSAKey) {
8                 RSAPublicKey publicKey = client.rsa.getRSAPublicKey();
9                 String n = publicKey.getN().toString();
10                client.sendRSAPublicKey();
11                client.makeRSAKey = true;
12                serverHandle.sendMsg(n, client);
13            }
14            else if(!client.getDESKey){
```

```

15         .....
16     }
17 }
18 }

```

如果客户端接收到 RSA 公钥的参数 `n`，将再次回调 `ClientReadHandler` 类的 `completed()` 函数。之后将收到的参数 `e` 和 `n` 组装成完整的 RSA 公钥，使用这个 RSA 公钥加密随机生成的 DES 密钥，使用写操作发送给服务器端。如果服务器端收到了，`ClientWriteHandler` 的 `completed()` 函数被回调。之后，就进行读操作，准备接收从服务器发来的消息。

```

1  @Override
2  public void completed(Integer result, ByteBuffer buffer) {
3      buffer.flip();
4      byte[] bytes = new byte[buffer.remaining()];
5      buffer.get(bytes);
6      String body;
7      try {
8          body = new String(bytes, "UTF-8");
9          if (!clientHandle.geteofRSAPublicKey) {
10             .....
11         }
12         else if (!clientHandle.getnofRSAPublicKey) {
13             clientHandle.n_ = body;
14             RSAPublicKey publicKey = new RSAPublicKey(new
15 BigInteger(clientHandle.e_), new BigInteger(clientHandle.n_));
16             String encode = RSA.encrypt(clientHandle.key, publicKey);
17             System.out.println("e of RSA public key:
18 "+clientHandle.e_);
19             System.out.println("n of RSA public key:
20 "+clientHandle.n_);
21             System.out.println("RSA encrypt result: "+encode);
22             clientHandle.getnofRSAPublicKey = true;
23             clientHandle.sendMsg(encode);
24             clientHandle.gotRSAPublicKey();
25             clientHandle.beginToChat();
26             ByteBuffer readBuffer = ByteBuffer.allocate(1024);
27             clientChannel.read(readBuffer, readBuffer, new
28 ClientReadHandler(clientChannel, latch));
29         }
30     }
31 }

```

```

1  private class ClientWriteHandler implements CompletionHandler<Integer,
2  ByteBuffer>{

```

```

2     private AsynchronousSocketChannel clientChannel;
3     private CountDownLatch latch;
4     public ClientWriteHandler(AsynchronousSocketChannel
    clientChannel,CountDownLatch latch) {
5         this.clientChannel = clientChannel;
6         this.latch = latch;
7     }
8     @Override
9     public void completed(Integer result, ByteBuffer buffer) {
10        //完成全部数据的写入
11        if (buffer.hasRemaining()) {
12            clientChannel.write(buffer, buffer, this);
13        }
14    }

```

如果服务器端发送的 RSA 公钥的参数 n 被成功接收，将再次回调 `ServerWriteHandler` 的 `completed()` 函数，使用读操作准备接收来自客户端发来的被 RSA 公钥加密的 DES 密钥：

```

1     @Override
2     public void completed(Integer result, ByteBuffer buffer) {
3         //如果没有发送完，就继续发送直到完成
4         if (buffer.hasRemaining())
5             client.channel.write(buffer, buffer, this);
6         else{
7             if(!client.makeRSAKey) {
8                 .....
9             }
10            else if(!client.getDESKey){
11                ByteBuffer readBuffer = ByteBuffer.allocate(1024);
12                client.channel.read(readBuffer, readBuffer, new
    ServerReadHandler(client));
13            }
14        }
15    }

```

如果服务器端成功收到了客户端发来的使用 RSA 公钥加密的 DES 密钥，则回调 `ServerReadHandler` 的 `completed()` 函数，继续使用写操作，准备接收来自客户端发来的消息。

```

1     @Override
2     public void completed(Integer result, ByteBuffer attachment) {
3         attachment.flip();
4         byte[] message = new byte[attachment.remaining()];

```

```

5      attachment.get(message);
6      try {
7          String msg = new String(message, "UTF-8");
8          if(!client.getDESKey) {
9              client.key = RSA.decrypt(msg,
client.rsa.getRSAPrivateKey());
10         System.out.println("RSA decrypt result: "+client.key);
11         client.rcvDESKey();
12         client.prepareDone();
13         client.getDESKey = true;
14         client.des=new DES(client.key);
15         ByteBuffer readBuffer = ByteBuffer.allocate(1024);
16         client.channel.read(readBuffer, readBuffer, new
ServerReadHandler(client));
17     }

```

这时，如果客户端点击“发送”按钮给服务器发送消息，“发送”按钮的监听事件会响应。首先获取输入框中文本（不能为空），然后将获取的文本使用 DES 密钥加密，使用写操作发送加密后的文本，最后将原来的文本显示在对话框的工作日志中，清空输入框。

```

1      sendMsg.addActionListener(new ActionListener() {
2          public void actionPerformed(ActionEvent arg0) {
3              if(editMsg.getText().length()==0) {
4                  JOptionPane.showMessageDialog(null, "消息内容不能为空！
", "Warning", JOptionPane.WARNING_MESSAGE);
5                  return;
6              }
7              String temp,response;
8              DES des=new DES(clientHandle.getDESKey());
9              response=editMsg.getText().trim();//从输入框获取要发送的原文
10             System.out.println("DES encry source: "+response);
11             temp=des.getResult(response, true);//DES 加密
12             System.out.println("DES encry binary result: "+temp);
13             clientHandle.sendMsg(temp);
14             workLog.append("client -> server: \n"+response+"\n");
15             editMsg.setText("");//发送后清空输入框
16         }
17     });

```

如果服务器收到客户端发来的消息，ServerReadHandler 的 completed() 函数被回调。先使用 DES 密钥将消息解密，然后将解密后的消息显示到对话框的工

作日志中。最后继续使用读操作接收来自客户端发来的消息，之后 `ServerReadHandler` 的 `completed()` 函数被回调，执行同样的操作。

```
1  @Override
2  public void completed(Integer result, ByteBuffer attachment) {
3      attachment.flip();
4      byte[] message = new byte[attachment.remaining()];
5      attachment.get(message);
6      try {
7          String msg = new String(message, "UTF-8");
8          if(!client.getDESKey) {
9              .....
10         }
11         else if(!msg.equals("END")) {
12             System.out.println("DES decry binary source: "+msg);
13             String res=client.des.getResult(msg, false);//DES 解密
14             System.out.println("DES decry result: "+res);
15             client.rcvLog(res);//将解密后的消息输出到对话框
16             ByteBuffer readBuffer = ByteBuffer.allocate(1024);
17             client.channel.read(readBuffer, readBuffer, new
ServerReadHandler(client));
18         }
```

如果服务器端要点击“发送”按钮给 `ComboBox` 上显示的客户端发送消息，“发送”按钮的监听事件会响应。与客户端不同的是，服务器要从客户端列表中查找对应的客户端信息，然后再调用对应客户端的写操作。

```
1  sendMsg.addActionListener(new ActionListener() {
2      public void actionPerformed(ActionEvent arg0) {
3          if(editMsg.getText().length()==0) {
4              JOptionPane.showMessageDialog(null, "消息内容不能为空！
", "Warning", JOptionPane.WARNING_MESSAGE);
5              return;
6          }
7          String
select=clientListComboBox.getSelectedItem().toString();//选择要发送的客户
端
8          Client client=clients.get(select);//获取与该客户端通信的线程和
套接字
9          if(client==null) {
10             JOptionPane.showMessageDialog(null, "此客户端不存在！
", "Error", JOptionPane.WARNING_MESSAGE);
11             return;
12         }
13         String temp = null, response = null;
```



```

14         response=editMsg.getText().trim();//从输入框获取要发送的原文
15         System.out.println("DES encry source: "+response);
16         temp=client.des.getResult(response, true);//DES 加密
17         System.out.println("DES encry binary result: "+temp);
18         serverHandle.sendMsg(temp, client);
19         workLog.append("server -> "+select+": \n"+response+"\n");
20         editMsg.setText("");//发送后清空输入框
21     }
22 });

```

客户端收到服务器发来的消息，ClientReadHandler 类的 completed()函数被回调，操作和服务端端的类似。

```

1     @Override
2     public void completed(Integer result,ByteBuffer buffer) {
3         buffer.flip();
4         byte[] bytes = new byte[buffer.remaining()];
5         buffer.get(bytes);
6         String body;
7         try {
8             body = new String(bytes,"UTF-8");
9             if (!clientHandle.geteofRSAPublicKey) {
10                 .....
11             }
12             else if (!clientHandle.getnofRSAPublicKey) {
13                 .....
14             }
15             else {
16                 if (exit) {
17                     latch.countDown();
18                     System.exit(0);
19                 }
20                 DES des=new DES(clientHandle.key);
21                 String receiveMsg=body;
22                 if(!receiveMsg.equals("END")) {
23                     //接收消息并解密
24                     System.out.println("DES decry binary source:
25 "+receiveMsg);
26                     String res=des.getResult(receiveMsg, false);//DES 解密
27                     System.out.println("DES decry result: "+res);
28                     clientHandle.rcvLog(res);//将解密后的消息输出到对话框
29                     ByteBuffer readBuffer = ByteBuffer.allocate(1024);
30                     clientChannel.read(readBuffer,readBuffer,new
ClientReadHandler(clientChannel, latch));
31                 }

```

```

31         else {
32
33         }
34     }
35     } catch (UnsupportedEncodingException e) {
36         e.printStackTrace();
37     } catch (IOException e) {
38         // TODO 自动生成的 catch 块
39         e.printStackTrace();
40     }
41 }

```

3.2.2 线程退出的实现

如果客户端要点击窗口上的×，则客户端线程退出。这里分了一些情况来讨论：如果客户端和服务端已连接并且“发送”按钮可点击（表明客户端的准备工作已经做好，即已经完成密钥的交互），则向服务器发送明文“END”消息，服务器端响应之后即可退出程序；如果客户端和服务端已连接并且“发送”按钮可点击，则不能退出线程；如果客户端和服务端未连接，则结束客户端线程的阻塞状态，退出程序。

```

1     addWindowListener(new WindowAdapter() {
2         @Override
3         public void windowOpened(WindowEvent e) {
4             editMsg.requestFocus();
5         }
6
7         @Override
8         public void windowClosing(WindowEvent e) {
9             if (clientHandle.isconn) {
10                 if (sendMsg.isEnabled()) {
11                     exit=true;
12                     clientHandle.sendMsg("END");
13                 }
14             }
15             else {
16                 exit=true;
17                 clientHandle.latch.countDown();
18             }
19         }
20     });

```

如果客户端发送“END”消息，服务器端的 `ServerReadHandler` 的 `completed()` 函数被回调，服务器端会关闭和客户端的 TCP 连接，并且从客户端列表中删除此客户端的信息。

```
1 private class ServerReadHandler implements CompletionHandler<Integer,
  ByteBuffer> {
2     private Client client;
3     public ServerReadHandler(Client client) {
4         this.client = client;
5     }
6     // 读取到消息后的处理
7     @Override
8     public void completed(Integer result, ByteBuffer attachment) {
9         attachment.flip();
10        byte[] message = new byte[attachment.remaining()];
11        attachment.get(message);
12        try {
13            String msg = new String(message, "UTF-8");
14            if(!client.getDESKey) {
15                .....
16            }
17            else if(!msg.equals("END")) {
18                .....
19            }
20            else {
21                client.channel.shutdownInput();
22                client.channel.shutdownOutput();
23                client.channel.close();
24                serverHandle.deleteClient(client.name);
25            }
26        } catch (UnsupportedEncodingException e) {
27            e.printStackTrace();
28        } catch (IOException e) {
29            // TODO 自动生成的 catch 块
30            e.printStackTrace();
31        }
32    }
```

客户端发送“END”消息，还会收到空消息，因此 `ClientReadHandler` 的 `completed()` 函数被回调。由于 `exit` 标记为 `true`，因此会从这个 `if` 语句中解除客户端线程的阻塞状态（如果客户端和服务器未连接，则结束客户端线程的阻塞状态，这段代码在此时也会被执行），退出程序。代码已经在上面展示过。

如果服务器要点击窗口上的×，则服务器线程退出。分两种情况讨论：如果没有连接的客户端，则直接退出；如果有，则要遍历客户端列表，如果有客户端没有准备好（即已经完成密钥的交互），则不能退出程序。

```
1      addWindowListener(new WindowAdapter() {
2          @Override
3          public void windowOpened(WindowEvent e) {
4              editMsg.requestFocus();
5          }
6
7          @Override
8          public void windowClosing(WindowEvent e) {
9              String temp = (String)clientListComboBox.getSelectedItem();
10             if (temp == null) {
11                 exit=true;
12                 serverHandle.latch.countDown();
13             }
14             else {
15                 boolean find=false;
16                 for (String client : clients.keySet()) {
17                     if (clients.get(client).prepare) {
18                         find=true;
19                         break;
20                     }
21                 }
22                 if (!find) {
23                     exit=true;
24                     serverHandle.latch.countDown();
25                 }
26             }
27         }
28     });
```

此时在服务器线程的函数中，线程会解除阻塞状态，退出程序，代码已经在上面展示过。

3.2.3 客户端断线重连的实现

如果服务器端要点击窗口上的×退出程序，则客户端和服务器的连接状态丢失，首先 `ClientReadHandler` 的 `failed()` 函数被回调，关闭当前的信道并重新打开信道，客户端保持原来的 IP 地址和端口号，重新尝试和服务器发起连接。

```
1      @Override
```

```

2      public void failed(Throwable exc,ByteBuffer attachment) {
3          exc.printStackTrace();
4          System.out.println("Data receive failed: "+exc.getMessage());
5          clientHandle.isconn = false;
6          //断线重连, 隔一秒尝试连接一次
7          //重新连接成功以后会重新获得密钥
8          clientHandle.reconnecting();
9          try {
10             Thread.sleep(1000);
11         } catch (InterruptedException e) {
12             // TODO 自动生成的 catch 块
13             e.printStackTrace();
14             System.out.println("InterruptedException: "+e.getMessage());
15         }
16         try {
17             clientHandle.reconn = true;
18             clientHandle.geteofRSAPublicKey=false;
19             clientHandle.getnofRSAPublicKey=false;
20             //重新建立套接字并和服务端连接, 保持客户端原端口不变
21             clientChannel.close();
22             clientHandle.clientChannel =
AsynchronousSocketChannel.open();
23             clientHandle.clientChannel.bind(clientHandle.address);
24             clientHandle.clientChannel.connect(new
InetSocketAddress(clientHandle.serverIP, clientHandle.port),
clientHandle, clientHandle);
25         } catch (IOException e) {
26             // TODO 自动生成的 catch 块
27             e.printStackTrace();
28             System.out.println("IOException: "+e.getMessage());
29         }
30     }
31 }

```

如果连接成功, 将回调 `AsyncClientHandler` 类的 `completed()` 函数, 其他操作则保持原来的正常操作, 不再介绍。如果连接失败, 则回调 `AsyncClientHandler` 类的 `failed()` 函数, 重新尝试连接, 每隔 1s 重新尝试一次。如果重新尝试连接仍失败, 则在此回调 `AsyncClientHandler` 类的 `failed()`; 如果连接成功, 将回调 `AsyncClientHandler` 类的 `completed()` 函数, 执行正常的操作。如果连接恢复, DES 密钥仍采用原来的, 不再重新生成。

```

1      @Override

```

```

2     public void failed(Throwable exc, AsyncClientHandler attachment) {
3         exc.printStackTrace();
4         if (!reconn) {
5             System.out.println("Connect failed: "+exc.getMessage());
6             JOptionPane.showMessageDialog(null, "Connect failed:
"+exc.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
7             try {
8                 clientChannel.close();
9                 latch.countDown();
10            } catch (IOException e) {
11                e.printStackTrace();
12            } finally {
13                System.exit(0);
14            }
15        }
16        else {
17            //断线重连，隔一秒尝试连接一次
18            //重新连接成功以后会重新获得RSA 密钥
19            try {
20                Thread.sleep(1000);
21            } catch (InterruptedException e) {
22                // TODO 自动生成的 catch 块
23                e.printStackTrace();
24                System.out.println("InterruptedException:
"+e.getMessage());
25            }
26            try {
27                clientHandle.reconn = true;
28                clientHandle.geteofRSAPublicKey=false;
29                clientHandle.getnofRSAPublicKey=false;
30                //重新打开通道并和服务器连接，保持客户端原端口不变
31                clientChannel.close();
32                clientChannel = AsynchronousSocketChannel.open();
33                clientChannel.bind(clientHandle.address);
34                clientChannel.connect(new InetSocketAddress(serverIP,
port), this, this);
35            } catch (IOException e) {
36                // TODO 自动生成的 catch 块
37                e.printStackTrace();
38                System.out.println("IOException: "+e.getMessage());
39            }
40        }
41    }

```

这次实验中断线重连的功能对上次实验的实现进行了优化，上次实验中，由于每隔 1s 就要尝试重连一次，但也是每隔 1s 就在日志和控制台输出重连的信息。因此这次尝试重连的信息只需要输出一次，就在第一次尝试重连的时候输出即可。而且在客户端失去连接之后，显示服务器 IP 的位置还是显示了连接到的服务器的 IP，这是一个 bug，在这次实验中改成显示“未连接到服务器”，连接到服务器之后再显示服务器 IP。

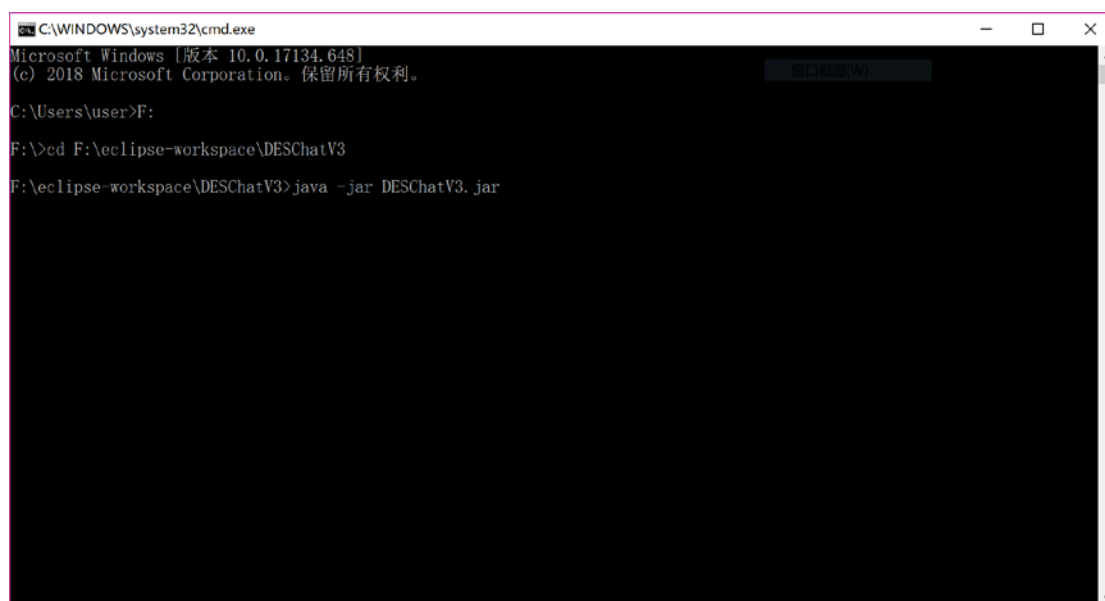
3.3 实验结果展示

3.3.1 运行程序的方法

和上次实验中运行程序的方式一致，这里就简单将上次实验报告中的这部分内容粘贴下来。

以下的图片仅为演示建立线程的方法，不作为实验结果展示。

打开终端（Windows 或 Linux 终端均可，这里使用 Windows 的 cmd 演示），定位到 DESChatV3.jar 所在的文件夹，输入命令 `java -jar DESChatV3.jar` 启动程序：

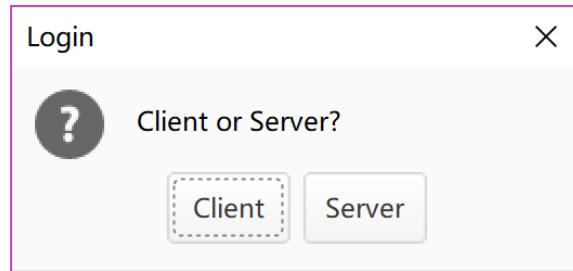


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.17134.648]
(c) 2018 Microsoft Corporation. 保留所有权利。

C:\Users\user>F:

F:\>cd F:\eclipse-workspace\DESChatV3
F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
```

程序启动后，看到以下对话框，点“Client”建立客户端线程，点“Server”则建立服务器线程。一定要先建立服务器线程再建立客户端线程，因为建立客户端线程前会尝试连接服务器线程，连接成功了才能建立客户端线程：



点击“Server”看到以下对话框则服务器线程建立成功（如果不成功则先关闭占用端口号 2000 的线程再尝试，点击窗口的×结束此线程及退出程序）：

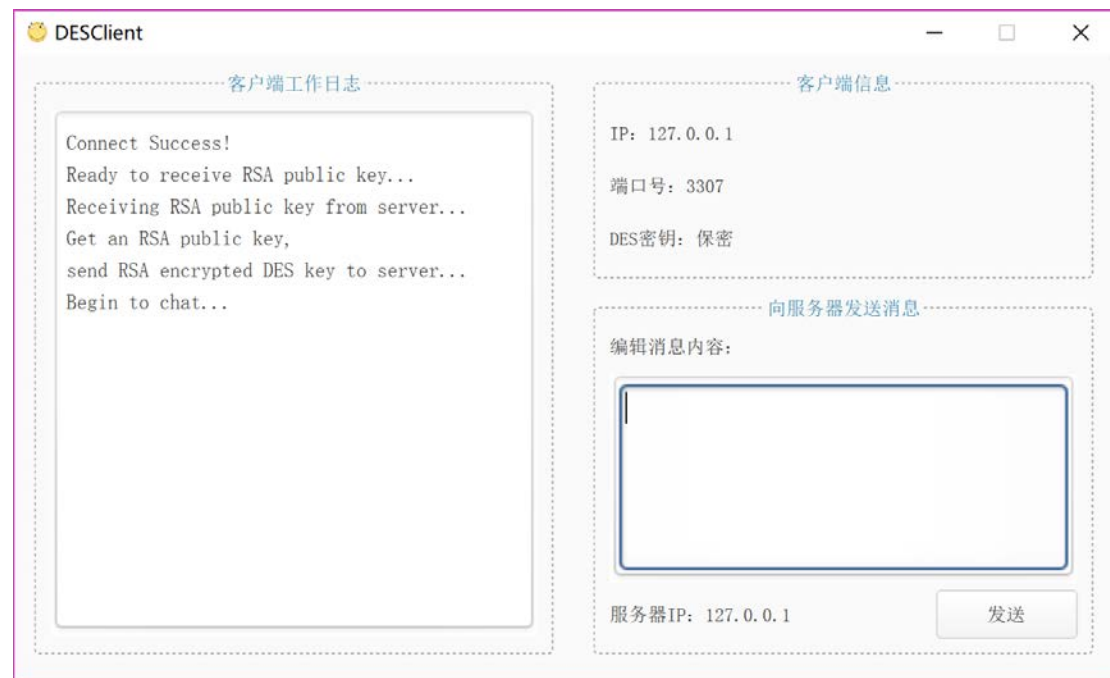


点击“Client”之后要先输入服务器的 IP 地址（127.0.0.1），一定要输入正确（如果 IP 地址格式正确但不是服务器的 IP 地址则无法连接，也无法建立客户端线程）：



看到以下的对话框则成功建立客户端线程（点击窗口的×结束此线程及退出程

序):



3.3.2 多线程聊天测试

以下测试按照上述的方法和下面的演示流程建立和退出线程，篇幅所限，仅展示该线程的最终输出结果。（***RSA 公钥参数以十进制显示，RSA 加密结果以十六进制显示***）

程序演示的流程如下：

建立服务器线程

建立客户端线程 1

客户端 1 向服务器发送：你好

客户端 1 向服务器发送：在吗

服务器向客户端 1 发送：我在

服务器向客户端 1 发送：有事吗

客户端 1 向服务器发送：没有

建立客户端线程 2

建立客户端线程 3

客户端 2 向服务器发送：我喜欢唱、跳、rap、篮球

客户端线程 1 退出

服务器向客户端 2 发送：鸡你太美
客户端线程 2 退出
建立客户端线程 4
服务器向客户端 3 发送：今天在好好学习吗？
客户端 3 向服务器发送：没有！
客户端线程 3 退出
客户端 4 向服务器发送：今天吃什么了？
服务器向客户端 4 发送：脆皮鸡!!!!
服务器线程退出
重新启动服务器线程
客户端 4 向服务器发送：今天睡了几个小时？
服务器向客户端 4 发送：24 小时！
客户端线程 4 退出
服务器线程退出

客户端 1 最终的可视化界面如下：



客户端 1 最终的终端输出如下：

```
C:\WINDOWS\system32\cmd.exe
C:\Users\user>F:
F:\>cd F:\eclipse-workspace\DESChatV3
F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
Choose: 0
127.0.0.1
DES key: KSFZLKWJ
e of RSA public key: 239715192152703078186360863727115656537398872902854626886492980336945982214042066211949812574835407
447409209895741030009062808615084852388205177974925109509215692875943861499622216611219356698206594043430844316002700806
23928058504150360448965275682585819312397077046697255348679925631993890368308138426472979
n of RSA public key: 354074183925675046977904957590643003435704010017585081994018314505801264508536569358931899931604252
249910760904980791885680085156139395568295875696819442112445826013172391478953987672926140105969455019349521701369428745
52752275785394113993181949043191689237215667722183320351310957814050728224233489260675361
RSA encrypt result: 2e55bb47a65fd212c3a7dd451e25e74875717d22030761cf802da7a131f3f25bccf94c18c167d1bb7c9d15f616185430f2a
59262c2987e859d370af40297c18d9743efe24a70ef4051c0ab9ec0ae768abd1de166b2e5e8c4b40710cddf232186969fad7442af5c2dbb8837ad9e9
f5692d0a19d49eddaa634a25e49c42cc8a8
DES encry source: 你好
DES encry binary result: 000100001010111100101100011100010111000110000101000010001011110
DES encry source: 在吗
DES encry binary result: 1011010111110110101100101111110111001001010111001011001111001011
DES decry source: 00001001000111101101001011011000111101011101010111001100110000
DES decry result: 我在
DES decry binary source: 0101001011000101001100101101101100111001000001001110011101010000101110100110010001110010111
00001011000010101001111011011011
DES decry result: 有事吗
DES encry source: 没有
DES encry binary result: 1011011010100010000111111011101111110001101101011000001111011
F:\eclipse-workspace\DESChatV3>
```

客户端 2 最终的可视化界面如下：



客户端 2 最终的终端输出如下：

```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
Microsoft Windows [版本 10.0.17134.648]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\user>F:

F:\>cd F:\eclipse-workspace\DESChatV3

F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
Choose: 0
127.0.0.1
DES key: JHARGVQB
e of RSA public key: 315974381110482288227657339679494233788413140255380473905681924701622981517026427623834376136616890
130217751230154606484069806359384618600542676137241070522558856195968067594672672413364508382698448145768060127219695892
25700129447251809901573081876607602779973404233534582215483357373296416396578433182819099
n of RSA public key: 569063856599707921025529192605623121388697760058839625964999613984510364939942526887665510987948532
641325822513488296145741283933971149219570287508408010302559775635313169649237283227526656565721190614263771777593218260
74117828778163334098842716712520410812057134325033745415244934008548767159414583783529713
RSA encrypt result: 3814d8cc5402715430eeb159836b3f9d4c471f5bc4565a3b6f403feceb22c4fdbcd8166916f63a3c4cb592c8785fc25ebd5f
22286f04ec3ea892d0aed47ed165dfec1e992b4f15fb80e9cf040093234d7a41aa5c43fa3096f73f47e432d335820eb6a548803f00f4987fd98e259f
a34f531dfb3eaa514f58dda967f2a234e513
DES encry source: 我喜欢唱、跳、rap、篮球
DES encry binary result: 010110010011010110010101111001111010001111010000001001000001000000011101101100011011001010111
00100110010000111111100010000011010000101101010110101111100000100111100100100111100001100101101110001101011111101
11010101100100001010100101100010001110111111010001010101101000110000100000101101010010011101101010010
DES decry binary source: 00101111011000011000000110001001101111110111111101110001001011000101011101001011011100111
110100011001101011000101100011000
DES decry result: 鸡你太美
```

客户端 3 最终的可视化界面如下：



客户端 3 最终的终端输出如下：

```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
Microsoft Windows [版本 10.0.17134.648]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\user>F:

F:\>cd F:\eclipse-workspace\DESChatV3

F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
Choose: 0
127.0.0.1
DES key: JUGAYAOM
e of RSA public key: 19682860263565763865075674494953193562811260466865635421254235096480817990824571881416856123919695
156475074610974714502176719828056627466116047699969993374080174606601110412785337613916081391627934828487166020081657438
6566026878756145679976022075519551677684313113194789999542276916087138793732472907656939
n of RSA public key: 792233949456554124057890389013657939014224810979011299316080844700081763536435069385995063882898733
910254893302692022168927832063982230432175463129792273970875715630997206917987946455322717911312236497111278984057424795
267190239902460990495061559052518499269050274633386909740557406372577228651658319499313
RSA encrypt result: 24e2d42cbb6e8093ed50887a37af32c81830b3176904d6b926fc3aafbe44522c24fcdf2d70ae9ce7c72e1684e6f88c5ab98d
601eacda8f5827c7367b171e104329e83e0c7f93eb98fc6ad61821fae122515622c1390997abc56630a4ee63f63912449942ebca9ee6d10b25c831f1
88f6896ed07ef0ca81e8f729e23e5dde82b
DES decry binary source: 101010101111001011100101101000011010000101100000000011000110000100110001010010100110001001110
1000010100000010100010011001100101101110110011100111001100000001010111111111111011011001011011110011000001000111110
0010001110011100100001001010100101000111
DES decry result: 今天在好好学习吗?
DES encry source: 没有!
DES encry binary result: 11001110000001110000001100110100101001100000011101001110001110010001110100011101110
111110000010001100000101001010110
```

客户端 4 最终的可视化界面如下：





客户端 4 最终的终端输出如下:


```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
Choose: 0
127.0.0.1
DES key: PVRQCMGH
e of RSA public key: 724397502539183693927335158052832618473728582296057014585262226491753291882969472699449481342105005
11396961825452445024759261551177538629904537589290812229642665484208189536732370547826127397202913706995323817525321512
221041626637517491199147579011816841957388534570591583315646220758582741965791292465673
n of RSA public key: 42886883554675105852580782691447075885507769227708532579809855509613819393683803715994442797932424
293026463305532932649580033136435624659121720155102735033647265945688411937158589533481513226233805860984304955969736276
8404062495577087960611035914654369718816455397272025822249811297418583014482504251458833
RSA encrypt result: 529953f846fceb6fb4ef54d55838200941178ca14e7198246ef6e107a565181f986fa1e5842c139805a2cfff1185e1a9a630e
601b4836f32e12ebdde4db776ffc03fdc4f98ed98ee76a5c8b6e132e9c1b6e5ab705f19f6eca9af1d34ac4d80751c3aa312ab2a95a6eb4a2a06a303
74bd0a757f405185d2aa46b1e2da0f1f4f6
DES encry source: 今天吃什么了?
DES encry binary result: 100101001010010011001011101011010011100100011101111001001010111000011000000111011100101001110
011010101001011011101011000110101001000000111100011110011100010011110101100000110101010100
DES decry binary source: 101001001001100100011000110001010101110100011011001001100110110001001110110100011011011011011
011110001101111010000101000010000010111010000010001110101110011001110000100010101000000010110
DES decry result: 脆皮鸡!!!
Data receive failed: 指定的网络名不再可用。

DES key: PVRQCMGH
e of RSA public key: 273033058869271609636076179784950672751215437286080925505061559483440151640772369313810215236144074
728091090260417346516621179164939344268569911573429369488891916337421988759917398504795829693445695305215517642676207766
7948606845857108822364325726467063164695003821698114184888703835458545857418859777085147
n of RSA public key: 10542403052798163266330434310946414254901857819953789457955519389189862428281211441309614251506646
67177405882886404411685381524450835206701116894166014250312551762661985872361183468626795075219874460985542858317067492
16389924076719663015331586206698703730171037579848186069216204970124720764495390629633117
RSA encrypt result: 852b90c9999c596ff429d3491138c5e1dbab3d083a67a9694bb656b97e52eeb4e01e2504fff29f4bd389a7ac4ba72bd31214
d6060efc050b87343d05906321d616ea8fbd84f8d3b8d1dbd1a74a9e6f2930c38ef5821582ef85901573900b683b915d1e7877c3df55063ed43879e8
d6060efc050b87343d05906321d616ea8fbd84f8d3b8d1dbd1a74a9e6f2930c38ef5821582ef85901573900b683b915d1e7877c3df55063ed43879e8
```

```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
8404062495577087960611035914654369718816455397272025822249811297418583014482504251458833
RSA encrypt result: 529953f846fceb6fb4ef54d55838200941178ca14e7198246ef6e107a565181f986fa1e5842c139805a2cfff1185e1a9a630e
601b4836f32e12ebdde4db776ffc03fdc4f98ed98ee76a5c8b6e132e9c1b6e5ab705f19f6eca9af1d34ac4d80751c3aa312ab2a95a6eb4a2a06a303
74bd0a757f405185d2aa46b1e2da0f1f4f6
DES encry source: 今天吃什么了?
DES encry binary result: 100101001010010011001011101011010011100100011101111001001010111000011000000111011100101001110
011010101001011011101011000110101001000000111100011110011100010011110101100000110101010100
DES decry binary source: 101001001001100100011000110001010101110100011011001001100110110001001110110100011011011011011
011110001101111010000101000010000010111010000010001110101110011001110000100010101000000010110
DES decry result: 脆皮鸡!!!
Data receive failed: 指定的网络名不再可用。

DES key: PVRQCMGH
e of RSA public key: 273033058869271609636076179784950672751215437286080925505061559483440151640772369313810215236144074
728091090260417346516621179164939344268569911573429369488891916337421988759917398504795829693445695305215517642676207766
7948606845857108822364325726467063164695003821698114184888703835458545857418859777085147
n of RSA public key: 10542403052798163266330434310946414254901857819953789457955519389189862428281211441309614251506646
67177405882886404411685381524450835206701116894166014250312551762661985872361183468626795075219874460985542858317067492
16389924076719663015331586206698703730171037579848186069216204970124720764495390629633117
RSA encrypt result: 852b90c9999c596ff429d3491138c5e1dbab3d083a67a9694bb656b97e52eeb4e01e2504fff29f4bd389a7ac4ba72bd31214
d6060efc050b87343d05906321d616ea8fbd84f8d3b8d1dbd1a74a9e6f2930c38ef5821582ef85901573900b683b915d1e7877c3df55063ed43879e8
8e8b554e37a787c8eeeca424b9a6fad0cf2
DES encry source: 今天睡了几个小时?
DES encry binary result: 0000010010110011010100000101111010001101101101100010100101010001010011110101010011110100
101110100110010011001001110011001111100001010101010111101100011000110001011111111000101000111101110000011110
101101001010111100010110000101001101100
DES decry binary source: 0101101000101111100011100011110011101001001011100101100010101100010001101001111001
00001110001000100101010100010010
DES decry result: 24小时!
```

另外，打开客户端 4 对应的错误日志可以看到服务器断线期间的错误堆栈输出（不管是服务器线程还是客户端线程创建的时候都会有日志输出，这些日志可以删掉）：

```
error-client-20200520113546.log - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
java.io.IOException: 指定的网络名不再可用。

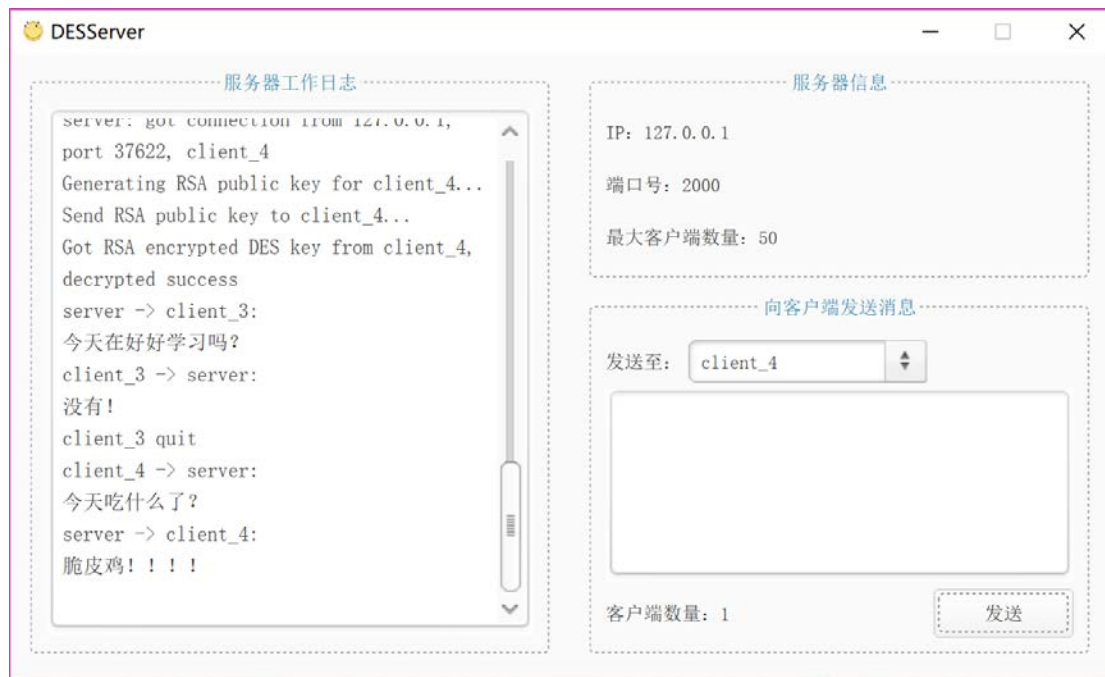
    at java.base/sun.nio.ch.io.cp.translateErrorToIOException(Unknown Source)
    at java.base/sun.nio.ch.io.cp.access$700(Unknown Source)
    at java.base/sun.nio.ch.io.cp$EventHandlerTask.run(Unknown Source)
    at java.base/sun.nio.ch.AsynchronousChannelGroupImpl$1.run(Unknown Source)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
    at java.base/java.lang.Thread.run(Unknown Source)
java.io.IOException: 远程计算机拒绝网络连接。

    at java.base/sun.nio.ch.io.cp.translateErrorToIOException(Unknown Source)
    at java.base/sun.nio.ch.io.cp.access$700(Unknown Source)
    at java.base/sun.nio.ch.io.cp$EventHandlerTask.run(Unknown Source)
    at java.base/sun.nio.ch.AsynchronousChannelGroupImpl$1.run(Unknown Source)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
    at java.base/java.lang.Thread.run(Unknown Source)
java.io.IOException: 远程计算机拒绝网络连接。
```

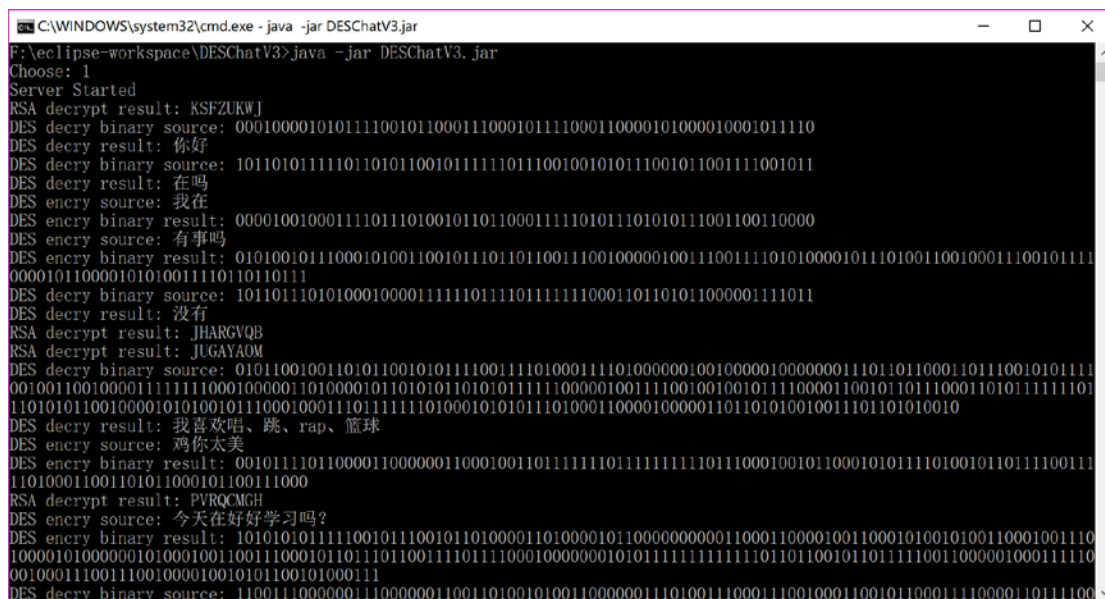
第一次开启服务器的最终可视化界面如下：





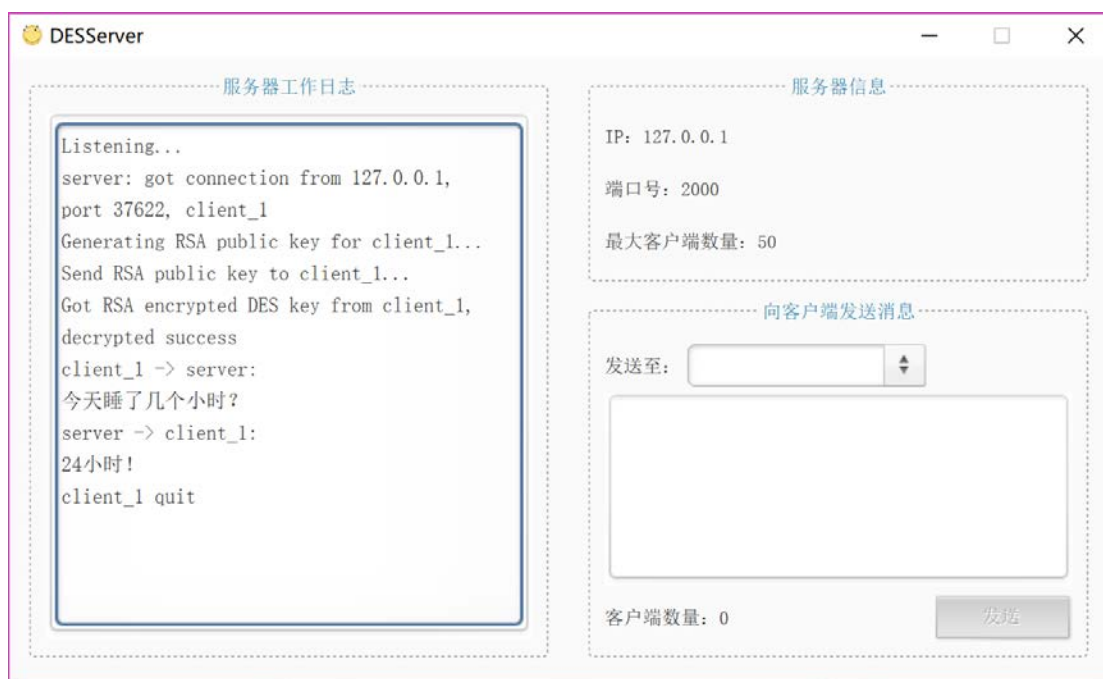


第一次开启服务器的最终的终端输出如下：



```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
DES encry source: 有事吗
DES encry binary result: 010100101110001010011001011101011001110010000010011100111101010000101110100110010001110010111
00001011000010101001111011011011
DES decry binary source: 10110111010100010000111111011101111110001101101011000001111011
DES decry result: 没有
RSA decrypt result: JHARGVQB
RSA decrypt result: JUGAYAOM
DES decry binary source: 01011001001101011001010111100111101000111101000001001000001000000111011011000110111001010111
00100110010000111111100010000011010000101101010110101011111000001001110010010011110000110010111000010101111101
1101010110010000101000101110001000111011111101000101011101000110000100111010100100111011010010
DES decry result: 我喜欢唱、跳、rap、篮球
DES encry source: 鸡你太美
DES encry binary result: 00101111011000011000000110001001101111110111111111101110001001011000101011110100101011100111
110100011001101011000101100111000
RSA decrypt result: PVRCMGH
DES encry source: 今天在好好学习吗?
DES encry binary result: 10101010111110010111001011010000110100001011000000000110001100001001100010100110001001110
10000101000000101000100110011000101101110110011100010000000101011111111111011011001011011110011000001000111110
0010001110011100100001001010110010100011
DES decry binary source: 110011100000011100000011001101001010011000000110100111000111001000110010111000011011100
111110000010001100000101001010110
DES decry result: 没有!
DES decry binary source: 100101001010010011001011101011010011100100011101111001001010111000011000000111011100101001110
01101010100101011111010110001101011001000000111100011111001110001001111101011000001101010101100
DES decry result: 今天吃什么了?
DES encry source: 脆皮鸡!!!
DES encry binary result: 10100100100110010001100011000101010111010001101100100110011011000100111011010001101110111011
0111100011011110100001010000110000010111101000000100011101011100110011100001000101000000010110
```

第二次开启服务器的最终可视化界面如下（由于是第二次开启服务器，客户端 4 在这里的编号又重新开始计算了，所以是 **client_1**）:



第二次开启服务器的最终的终端输出如下:

```
C:\WINDOWS\system32\cmd.exe - java -jar DESChatV3.jar
Microsoft Windows [版本 10.0.17134.648]
(c) 2018 Microsoft Corporation. 保留所有权利。

C:\Users\user>F:

F:\>cd F:\eclipse-workspace\DESChatV3

F:\eclipse-workspace\DESChatV3>java -jar DESChatV3.jar
Choose: 1
Server Started
RSA decrypt result: PVRQCMGH
DES decrypt binary source: 00000100101100110101000001011111010001101101101101100010100101010001010011110101010011110100
1011101001100100110010011001110011001111100001010101011111011000110001100010111111110001011000111101110000011110
1011010010101111000101100001101001101100
DES decrypt result: 今天睡了几个小时?
DES encrypt source: 24小时!
DES encrypt binary result: 0101101000101111100011100101100001111001111010010010111100101110001010110010010001101001111001
000011100010001001010110100010010
```

四、实验遇到的问题及其解决方法

4.1 如果使用 `BigInteger` 类产生指定范围的随机数

可以发现,Java 的 `BigInteger` 类并没有定义可以产生指定范围随机数的普通函数或者构造函数,而只有产生范围为 $[0, 2^{\text{bitLen}} - 1]$ 的随机数。于是参考 JDK 1.8 中 `BigInteger` 类中 `passesMillerRabin()` 函数的实现,在 `RSA` 类中实现了产生指定范围随机数的函数 `genRandom()`,见 3.1.2 节。

```
private boolean passesMillerRabin(int iterations, Random rnd) {
    // Find a and m such that m is odd and this == 1 + 2**a * m
    BigInteger thisMinusOne = this.subtract(ONE);
    BigInteger m = thisMinusOne;
    int a = m.getLowestSetBit();
    m = m.shiftRight(a);

    // Do the tests
    if (rnd == null) {
        rnd = ThreadLocalRandom.current();
    }
    for (int i=0; i < iterations; i++) {
        // Generate a uniform random on (1, this)
        BigInteger b;
        do {
            b = new BigInteger(this.bitLength(), rnd);
        } while (b.compareTo(ONE) <= 0 || b.compareTo(this) >= 0);
    }
}
```

4.2 如何提升实现的 RSA 算法的速度

本次实验最初实现的 RSA 算法的速度还是比较慢的，如果密钥长度在 400 bits 以上就会慢到难以忍受。因此，我考虑了以下优化的角度：

1. 将乘以、除以 2 或者 2 的倍数的实现改为移位操作
2. 求出参数 d 真的需要一个个去尝试吗？
3. 需要调用多少次 Miller-Rabin 算法？

对于第一个和第二个问题，最初就考虑到了。

对于第三个问题，我最初选择调用 100 次，但是参考了 JDK 1.8 中 `BigInteger` 类的 `primeToCertainty()` 实现，可以发现，在这个源代码当中如果要测试的那个数二进制位越长，要进行测试的次数就越少。当然，根据 ANSI x9.80 标准，这个是基于进行 Miller-Rabin 测试之后还要再进行 Lucas-Lehmer 测试（判断是不是梅森素数）的假设的，由于 Lucas-Lehmer 测试的效率和精度都较高，因此可以结合 Lucas-Lehmer 测试来减少 Miller-Rabin 测试的次数，进而提高素性检测的效率，否则进行 50 次测试是比较合适的。由于进行 50 次测试确实比较慢，所以我引入了这部分代码，为了简单起见也没有考虑 Lucas-Lehmer 测试的实现，这就包含了程序精确度和安全性的折中，即使是我设置的 `bitLen` 参数是 512，出错概率也小于万分之一，基本上满足本次实验要少量运行程序而非大量运行程序的精度需求。这对于需要大量运行该程序的企业级场合来说并不是好的方案。此外还有一个改进的点就是将较小的素数可以直接保存在一个表中，这样判断较小的素数可以直接查表。当然这次实验没有实现。

五、实验结论

1. 客户端发送的 DES 密钥使用服务器端传来的 RSA 公钥加密，服务器收到之后使用私钥解密出 DES 密钥。最终双方的通信内容都使用这个 DES 密钥加密，一方使用 DES 加密前的原文和另一方通过 DES 解密得到的原文是相同的，可以说明传输的密钥是对的并且本次实验的 RSA 算法实现正确。
2. 通过这次编程，更加深入地理解了 RSA 公钥和私钥产生的原理和 RSA 加密解密的原理。
3. 熟悉了使用 Java NIO 框架实现异步非阻塞网络通信的方法。这个框架的使用相比 Java 的标准 I/O 框架要困难一些，而且相关的资料也比较少。但是学习了

一个别人实现的例子之后，而且以前使用过 MFC 的 `CAsyncSocket` 类也是异步非阻塞 socket，就很快上手并改写了本次实验的 TCP 通信模块并保持上次实验中的 TCP 模块功能，因此也提高了重构代码的能力。由于时间关系，本次实验并没有将 TCP 通信模块改写为 select 机制的实现，但是有时间也要学习。

4. 本次实验实现的 RSA 算法的性能瓶颈在于大素数的产生，我也尝试使用了 `BigInteger` 类的 `probablePrime()` 方法去产生素数，即使是让 p 和 q 的长度都是 1024 位都是秒出结果，但是在我这里最少也要十几秒。尽管数字的长度越长，计算的时间就会越多，但这也并不是主要矛盾。因此本次实验实现的 RSA 算法还需要继续优化。