



南開大學
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《网络安全技术》课程作业

实验二：基于 RSA 算法自动分配密钥的加密聊天程序

学院：密码与网络空间安全学院

年级：2022 级

班级：信息安全

姓名：陆皓喆

学号：2211044

手机号：15058298819

2025 年 5 月 4 日

目录

1 实验目标	2
2 github 仓库	2
3 实验内容	2
4 实验原理	2
4.1 RSA 算法	2
4.2 MillerRabin 算法	3
5 实验步骤	4
5.1 实验环境	4
5.2 基于 DES 加解密的修改	4
5.2.1 GetKey 函数	4
5.2.2 RandomGenKey 函数	5
5.3 RSA 加解密算法实现	5
5.3.1 RSA_Operation.h	5
5.3.2 RSA_Operation.cpp	7
5.4 基于 TCP 协议的进一步优化	12
6 实验遇到的问题及其解决方法	21
6.1 对于 select 模型掌握不熟练	21
6.2 对于 RSA 密钥生成错误的处理错误	22
7 实验结论	22
7.1 利用 cmake 编译项目	22
7.2 新建服务器端和客户端	22
7.3 功能测试	23
7.3.1 英文输入测试	23
7.3.2 中文输入测试	23
7.3.3 断开连接测试	23
8 代码结构说明	24
9 参考文献	24

1 实验目标

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。

2 github 仓库

本次实验的有关代码和文件，都已经上传至我的个人 github 中。
您可以通过访问[此链接](#)来查阅我的代码文件。

3 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。

4 实验原理

本次实验涉及到 RSA 加解密、DES 加解密、TCP 协议通信、MillerRabin 算法等实验原理，由于在第一次实验中，我们已经详细介绍了 DES 算法和 TCP 通信协议，所以此处我们不再详细介绍，仅选择 RSA 算法和 MillerRabin 算法进行介绍。

4.1 RSA 算法

以下是 RSA 算法的对应参数的含义。

参数	含义
p, q	两个大质数（加密安全性的基础，需保密）
$n = p \times q$	公钥和私钥的一部分，加密/解密的模数（公开）
$\phi(n)$	欧拉函数，值为 $(p-1)(q-1)$ （计算私钥时使用，需保密）
e	公钥指数，与 $\phi(n)$ 互质（公开，用于加密）
d	私钥指数， e 的模 $\phi(n)$ 逆元（保密，用于解密）

下面简单介绍一下 RSA 算法的加解密的主要步骤。

1. 密钥生成

- 选择两个大质数 p 和 q (如 $p = 3, q = 7$)。
- 计算乘积 $n = p \times q$ (如 $n = 21$)。
- 计算欧拉函数 $\phi(n) = (p - 1)(q - 1)$ (如 $\phi(21) = 12$)。
- 选择公钥指数 e , 满足 $1 < e < \phi(n)$ 且 $\gcd(e, \phi(n)) = 1$ (如 $e = 5$)。
- 计算私钥指数 d , 满足 $e \times d \equiv 1 \pmod{\phi(n)}$ (如 $d = 5$, 因 $5 \times 5 \equiv 1 \pmod{12}$)。
- 结果:
 - 公钥: $(e, n) = (5, 21)$
 - 私钥: $(d, n) = (5, 21)$

2. 加密过程

- 输入明文 m (需满足 $1 \leq m < n$, 如 $m = 2$)。
- 计算公式:

$$c = m^e \bmod n$$

- 计算示例:

$$c = 2^5 \bmod 21 = 32 \bmod 21 = 11$$

3. 解密过程

- 输入密文 c (如 $c = 11$)。
- 计算公式:

$$m = c^d \bmod n$$

- 计算示例:

$$m = 11^5 \bmod 21 = 2$$

4.2 MillerRabin 算法

Miller-Rabin 算法是一种概率性素性检验方法, 用于判断一个数是否为素数, 其核心原理基于费马小定理和二次探测定理, 具体如下:

1. 费马小定理

若 p 是素数且 $a \not\equiv 0 \pmod{p}$, 则 $a^{p-1} \equiv 1 \pmod{p}$.

- 若存在 a 使得 $a^{n-1} \not\equiv 1 \pmod{n}$, 则 n 一定是合数 (强伪证)。

2. 二次探测定理

若 p 是素数且 $x^2 \equiv 1 \pmod{p}$, 则 $x \equiv 1 \pmod{p}$ 或 $x \equiv -1 \pmod{p}$.

- 若 n 是合数且存在 $x^2 \equiv 1 \pmod{n}$ 但 $x \not\equiv \pm 1 \pmod{n}$, 则 n 为合数 (二次伪证)。

实验流程:

- 将 $n - 1$ 分解为 $d \times 2^r$ (d 为奇数)。
- 随机选取基 a ，计算 $a^d \pmod n$ ，若结果为 1 或 $n - 1$ ，则通过一次检验；否则对结果连续平方 $r - 1$ 次，若中途出现 $n - 1$ ，则通过检验，否则 n 为合数。
- 重复多次随机选取不同的 a ，若均通过检验，则 n 大概率为素数（错误概率随检验次数指数降低）。

5 实验步骤

5.1 实验环境

本次实验与 Lab01 相同，仍然选择在服务器上进行实验，服务器的环境为 ubuntu22.04，整体的代码框架与上次也基本相同，生成的可执行文件我们存放在 bin 文件夹下。

代码的具体架构如下所示：

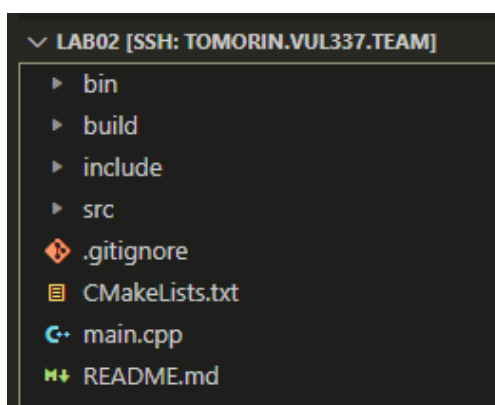


图 5.1: 代码架构

下面，我们就可以开始我们的实验了。

5.2 基于 DES 加解密的修改

首先，我们需要进一步修改我们上次实验中编写的 DES 算法的相关代码。由于本次实验需要实现密钥的分配工作，所以我们需要补充一些相关的函数。

5.2.1 GetKey 函数

代码如下所示：

```
1 uint8_t* DesOp::GetKey() {  
2     auto* key_copy = new uint8_t[8];  
3     Copy(key_copy, key, 8);  
4     return key_copy;  
5 }
```

GetKey 函数解释

该函数的作用是获取 DES 加密的 8 字节密钥副本。函数通过动态分配内存创建一个长度为 8 的 `uint8_t` 数组，调用 `Copy` 函数将类成员 `key` 中的原始密钥数据逐字节复制到新数组中，最后返回新数组的指针。这种方式避免了直接返回原始密钥的引用或指针，通过副本机制保证了密钥数据的封装性和安全性。

5.2.2 RandomGenKey 函数

代码如下所示：

```
1 void DesOp::RandomGenKey() {
2     std::random_device rd;
3     std::mt19937 engine(rd());
4     std::uniform_int_distribution<unsigned int> dist(0, 255);
5
6     for (int i = 0; i < 8; i++) {
7         key[i] = (uint8_t)dist(engine);
8     }
9     GenerateSubKeys();
10 }
```

RandomGenKey 函数解释

`DesOp::RandomGenKey()` 函数用于生成随机 DES 密钥并预处理子密钥。其通过 `std::random_device` 结合硬件熵源和 `std::mt19937` 强随机数生成器，逐字节填充 8 字节密钥数组，确保 64 位原始密钥的随机性。生成后调用 `GenerateSubKeys()` 执行密钥编排，将原始密钥转换为 16 轮加密所需的 48 位子密钥序列。该实现利用密码学安全的随机数生成机制，满足 DES 对密钥随机性的要求，但需注意硬件熵源的环境依赖及密钥存储安全，适用于需要动态生成 DES 密钥的加密场景。

5.3 RSA 加解密算法实现

5.3.1 RSA_Operation.h

我们编写的有关 RSA 算法的头文件代码如下所示：

```
1 // RSA 操作，用于生成密钥、加解密
2 #ifndef ENCCHAT_RSA_H
3 #define ENCCHAT_RSA_H
4
5 #include <stdint>
6 #include <random>
7
8 // 定义 128 位整型（注意：不是所有编译器都支持）
```

```
9  #define uint128_t __uint128_t
10 #define int128_t __int128_t
11
12 class RSA {
13 private:
14     uint64_t p;    // 素数 p
15     uint64_t q;    // 素数 q
16     uint64_t n;    // 模数 n = p * q
17     uint64_t phi;  // 欧拉函数 (n) = (p-1)*(q-1)
18     uint64_t e;    // 公钥指数 e
19     uint64_t d;    // 私钥指数 d
20
21     // 模幂运算: 计算 (base^exp) mod mod
22     static uint64_t ModExp(uint64_t base, uint64_t exp, uint64_t mod);
23     // 模逆运算: 计算 a 关于模 m 的逆元
24     static uint64_t ModInv(uint64_t a, uint64_t m);
25     // Miller-Rabin 素性测试: 检测 n 是否为质数, 默认测试轮数为 50 次
26     static bool MillerRabin(uint64_t n, int round = 50);
27
28 public:
29     RSA();    // 构造函数, 初始化密钥各项为 0
30     ~RSA();   // 析构函数
31
32     // 生成公钥和私钥, 返回 true 表示生成成功
33     bool GenerateKey();
34
35     // 获取公钥 e
36     inline uint64_t GetPublicKey() { return e; };
37
38     // 获取模数 n
39     inline uint64_t GetModulus() { return n; };
40
41     // 静态加密函数: 使用公钥 e 对明文进行加密
42     static uint64_t Encrypt(uint32_t plainText, uint64_t e, uint64_t n);
43
44     // 使用私钥解密密文, 返回解密后的明文
45     uint32_t Decrypt(uint64_t cipherText);
46
47     // 打印当前配置 (密钥、模数等)
48     void PrintConfig();
49 };
```

50 `#endif`

我们构造了一个 RSA 加解密的头文件，具体的函数和变量和含义如下所示：

- **p, q**: RSA 算法的两个大质数，是加密安全性的基础，需保密。
- **n**: 公钥和私钥的一部分，计算方式为 $n = p \times q$ ，是加密/解密的模数，公开。
- **phi**: 欧拉函数，值为 $\phi(n) = (p - 1)(q - 1)$ ，计算私钥时使用，需保密。
- **e**: 公钥指数，与 $\phi(n)$ 互质，公开，用于加密。
- **d**: 私钥指数，是 e 的模 $\phi(n)$ 逆元，即满足 $e \times d \equiv 1 \pmod{\phi(n)}$ ，需保密，用于解密。
- **ModExp**: 模幂运算函数，计算 $\text{base}^{\text{exp}} \pmod{\text{mod}}$ ，是 RSA 加密和解密的核心运算。
- **ModInv**: 模逆运算函数，计算 a 关于模 m 的逆元，即满足 $a \times x \equiv 1 \pmod{m}$ 的 x 。
- **MillerRabin**: Miller-Rabin 素性测试函数，用于检测一个数是否为质数，测试轮数越多，准确性越高。
- **GenerateKey**: 生成 RSA 密钥对，包括选择大质数 p 和 q ，计算 n 、 $\phi(n)$ ，选择公钥指数 e ，计算私钥指数 d 。
- **GetPublicKey**: 获取公钥指数 e 。
- **GetModulus**: 获取模数 n 。
- **Encrypt**: 静态加密函数，使用公钥 (e, n) 对明文进行加密。
- **Decrypt**: 使用私钥 (d, n) 解密密文，返回解密后的明文。
- **PrintConfig**: 在服务器端打印 RSA 具体的密钥数据。

5.3.2 RSA_Operation.cpp

该部分我们根据前面设计的头文件，对应的实现了 RSA 加密的每一个部分。下面我们对代码进行详细的分析。

ModExp 函数

```
1  uint64_t RSA::ModExp(uint64_t base, uint64_t exp, uint64_t mod) {
2      base = base % mod;
3      uint64_t idx = (1LL << 63);
4      while (!(exp & idx)) {
5          idx >>= 1;
6      }
7
8      uint128_t result = 1;
9      while (idx) {
10         result = (uint128_t)((result * result) % mod);
```



```
11     if (exp & idx) {
12         result = (uint128_t)((result * base) % mod);
13     }
14     idx >>= 1;
15 }
16
17 return (uint64_t)result;
18 }
```

此处我们实现了模幂运算，此处的算法基本与《密码学》中 RSA 算法相似，此处也不再展开介绍。

ModInv 函数

```
1 uint64_t RSA::ModInv(uint64_t a, uint64_t m) {
2
3     assert(a < m);
4
5     int128_t r0 = m, r = a;
6     int128_t q = -1;
7     int128_t s0 = 1, s = 0;
8     int128_t t0 = 0, t = 1;
9
10    while (r0 % r) {
11        int128_t tmp = r0;
12        r0 = r;
13        r = tmp % r0;
14
15        q = tmp / r0;
16
17        tmp = s0;
18        s0 = s;
19        s = tmp - s0 * q;
20
21        tmp = t0;
22        t0 = t;
23        t = tmp - t0 * q;
24    }
25
26    if (r == 1) {
27        if (s < 0)
28            s += a;
29        if (t < 0)
30            t += m;
```

```
31         return t;  
32     }  
33  
34     return 0;  
35 }
```

这个函数，我们主要实现了模逆的运算。此处的算法基本与之前《信息安全数学基础》中的算法一样，此处就不再展开介绍对应的原理了。

MillerRabin 函数

```
1  bool RSA::MillerRabin(uint64_t n, int round) {  
2      if (n == 2) return true;  
3      if (n < 2 || (n & 1) == 0) return false;  
4  
5      uint64_t m = n - 1;  
6      uint64_t k = 0;  
7      while ((m & 1) == 0) {  
8          m >>= 1;  
9          k++;  
10     }  
11  
12     std::random_device rd;  
13     std::mt19937 gen(rd());  
14     std::uniform_int_distribution<uint64_t> dis(2, n - 1);  
15  
16     while (round--> 0) {  
17         uint64_t a = dis(gen);  
18         uint64_t b = ModExp(a, m, n);  
19         if (b == 1 || b == n - 1) continue;  
20  
21         for (int i = 0; i < k; i++) {  
22             b = ModExp(b, 2, n);  
23             if ((b == n - 1) && (i < k - 1)) {  
24                 b = 1;  
25                 break;  
26             }  
27             if (b == 1) return false;  
28         }  
29  
30         if (b != 1) return false;  
31     }  
32     return true;  
}
```

}

这一个函数，我们实现了 Miller-Rabin 素性测试算法，用于判断一个 64 位无符号整数 n 是否为素数。我们先处理边界情况（如 n 为 2、小于 2 或偶数），再将 $n-1$ 分解为 $m * 2^k$ (m 为奇数)，然后进行 round 轮测试。每轮随机选择基 a ，计算 $b = a^m \bmod n$ ，若 b 为 1 或 $n-1$ 则通过；否则迭代平方 b ，若过程中 b 变为 $n-1$ 则标记通过。若所有轮次均通过则返回 `true`，否则返回 `false`。存在的问题是依赖未定义的 `ModExp` 函数，且 `if (b == n - 1) && (i < k - 1)` 中的 $i < k - 1$ 检查可能多余，标准实现中 b 一旦变为 $n-1$ 即可判定通过。此外，对于大素数， round 次测试可能影响性能。

GenerateKey 函数

```
1  bool RSA::GenerateKey() {
2      std::random_device rd;
3      std::mt19937 gen(rd());
4      std::uniform_int_distribution<uint64_t> dis(0x20000000, 0xFFFFFFFF);
5
6      const int MAX_ROUND = 50;
7      do {
8          p = dis(gen);
9      } while (!MillerRabin(p, MAX_ROUND));
10
11     do {
12         q = dis(gen);
13     } while (!MillerRabin(q, MAX_ROUND));
14
15     n = p * q;
16     phi = (p - 1) * (q - 1);
17
18     e = 65537;
19     while (e < phi) {
20
21         if ((phi % e != 0) && (MillerRabin(e, MAX_ROUND))) {
22             break;
23         }
24         e += 2;
25     }
26
27     if (e >= phi) {
28
29         p = 0; q = 0; n = 0; phi = 0; e = 0; d = 0;
30         return false;
31     }
32 }
```

```
33     d = ModInv(e, phi);  
34     return true;  
35 }
```

这个函数我们主要是用于 RSA 密钥的生成。为了便于在 C++ 系统上的计算，我们仅仅实现了 RSA 加密，并没有对其安全性做出保证，因此我们选取的 RSA 配置是比较容易被破解的，但是可以顺利地完成任务。如果我们生成的 p 和 q 不是素数的话，我们就需要重新生成一次对应的配置，如果 e 比 phi 要大的话也是不行的，另外，e 和 phi 是不能互素的。这些都是 RSA 的基础要求。对应的素数判定，我们使用的是前面编写的 MillerRabin 函数来进行伪素数判定。

Encrypt 函数

```
1  uint64_t RSA::Encrypt(uint32_t plainText, uint64_t e, uint64_t n) {  
2      return ModExp((uint64_t)plainText, e, n);  
3  }
```

这个函数主要用于加密。我们根据 RSA 的加密流程，在明文 plaintext 的基础上做 e 次幂，然后同余上 n，就可以得到我们的结果。

Decrypt 函数

```
1  uint32_t RSA::Decrypt(uint64_t cipherText) {  
2      return (uint32_t)ModExp(cipherText, d, n);  
3  }
```

这个函数，我们主要返回我们的解密后的数据。我们使用密文做 d 次幂，然后再同余上 n，就可以获得我们的明文的值 m。

PrintConfig 函数

```
1  // 打印 RSA 的具体配置信息  
2  void RSA::PrintConfig() {  
3      std::cout << "RSA 配置数据: " << std::endl;  
4      std::cout << "p (素数): " << p << std::endl;  
5      std::cout << "q (素数): " << q << std::endl;  
6      std::cout << "n (模数): " << n << std::endl;  
7      std::cout << "phi: " << phi << std::endl;  
8      std::cout << "e (公钥指数): " << e << std::endl;  
9      std::cout << "d (私钥指数): " << d << std::endl;  
10 }
```

编写这个函数的主要目的是，在服务器端显示出我们的 RSA 加密的具体配置，这样可以帮助我们更好地可视化我们的 RSA 配置过程。具体的内容就是，显示出我们的 RSA 的配置数据。

5.4 基于 TCP 协议的进一步优化

此处，我们按照实验的进阶要求，在上一次 DES 通信的基础上，首先完成了 RSA 的密钥传输和认证，另外，还完成了对应的进阶要求，也就是利用 `select` 模式进行传输的优化操作。下面我们就本次实验编写的 `chat` 文件展开进行介绍。与上次实验没有发生变化的部分，我就不再介绍了，此处主要就修改的部分展开进行介绍。

setNonBlocking 函数

```
1 // 设置 socket 为非阻塞模式
2 static void setNonBlocking(int sockfd) {
3     int flags = fcntl(sockfd, F_GETFL, 0);
4     fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
5 }
```

这个函数，我们将指定的套接字设置成了非阻塞模式。首先通过 `fcntl(sockfd, F_GETFL, 0)` 获取当前套接字的文件状态标志（如是否为 `O_NONBLOCK`、`O_ASYNC` 等），然后使用按位或（`|`）操作将 `O_NONBLOCK` 标志添加到现有标志中，最后通过 `fcntl(sockfd, F_SETFL, ...)` 将新的标志设置回套接字，从而启用非阻塞模式。

Connect 函数

```
1 void Chat::Connect() {
2     clientSocket = socket(AF_INET, SOCK_STREAM, 0);
3     if (clientSocket < 0) {
4         std::cerr << "Error: Failed to create socket." << std::endl;
5         return;
6     }
7     // 设置客户端 socket 为非阻塞模式
8     setNonBlocking(clientSocket);
9
10    struct sockaddr_in serverAddr;
11    serverAddr.sin_family = AF_INET;
12    serverAddr.sin_port = htons(serverPort);
13    serverAddr.sin_addr.s_addr = inet_addr(serverIp);
14
15    if (connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) <
16        ↪ 0) {
17        // 非阻塞 connect 返回 -1 是正常情况，
18        // 通过 select() 判断是否连通
19        fd_set writefds;
20        FD_ZERO(&writefds);
21        FD_SET(clientSocket, &writefds);
22        timeval tv;
```

```
22     tv.tv_sec = 10; // 超时 10 秒
23     tv.tv_usec = 0;
24     int ret = select(clientSocket + 1, NULL, &writefds, NULL, &tv);
25     if(ret <= 0) {
26         std::cerr << "Error: Failed to connect to server." << std::endl;
27         return;
28     }
29 }
30 std::cout << "Connected to server." << std::endl;
31 }
```

这个函数主要完成了建立非阻塞 TCP 连接的功能，具体代码解释如下所示：

1. **创建套接字：**调用 `socket(AF_INET, SOCK_STREAM, 0)` 创建 TCP 套接字，并通过 `setNon-Blocking` 将其设为非阻塞模式。
2. **配置服务器地址：**填充 `sockaddr_in` 结构体，指定服务器的 IP 地址 (`serverIp`) 和端口 (`serverPort`)，并转换为网络字节序。
3. **发起非阻塞连接：**
 - 调用 `connect` 立即返回 -1 是预期行为（非阻塞模式下连接建立过程会异步进行）。
 - 使用 `select` 监听套接字是否可写 (`writefds`)：
 - 若 `select` 返回 `> 0` 且套接字在 `writefds` 中，表示连接成功。
 - 若返回 0（超时）或 -1（错误），则连接失败。

Send 函数

```
1 void Chat::Send() {
2     std::cin.getline(message, MAX_MESSAGE_LENGTH);
3     char* cipherText = nullptr;
4     int cipherTextLength = -1;
5
6     if (strcmp(message, EXIT_COMMAND) == 0) {
7         isRunning = false;
8         exited = true;
9     }
10
11     des.Encrypt(message, strlen(message), cipherText, cipherTextLength);
12     if (send(clientSocket, cipherText, cipherTextLength, 0) < 0) {
13         std::cerr << "Error: Failed to send message." << std::endl;
14         delete[] cipherText;
15         return;
16     }
```

```
17     delete[] cipherText;
18 }
```

这段代码实现了 Chat 类的消息发送功能，主要逻辑为读取用户输入、加密后发送。具体的代码分析如下所示：

1. 用户输入处理：

- 使用 `std::cin.getline` 读取一行输入到 `message` 缓冲区,最大长度为 `MAX_MESSAGE_LENGTH`。

2. 退出逻辑：

- 若输入为 `EXIT_COMMAND` (如 “exit”), 设置 `isRunning` 和 `exited` 标志为 `true`, 触发退出流程。

3. 消息加密：

- 调用 `des.Encrypt` 对消息进行 DES 加密, 生成密文 `cipherText` 和长度 `cipherTextLength`。

4. 网络发送：

- 通过 `send` 将加密后的消息发送至服务器。
- 若发送失败, 输出错误信息并释放加密内存。

ReceiveThread 函数

```
1 void Chat::ReceiveThread() {
2     const char* info = isServer ? "Client" : "Server";
3     char* plainText = nullptr;
4     int plainTextLength = -1;
5
6     while (isRunning) {
7         fd_set readfds;
8         FD_ZERO(&readfds);
9         FD_SET(clientSocket, &readfds);
10        timeval tv;
11        tv.tv_sec = 1; // 超时 1 秒, 避免长时间阻塞
12        tv.tv_usec = 0;
13        int ret = select(clientSocket + 1, &readfds, NULL, NULL, &tv);
14        if(ret > 0 && FD_ISSET(clientSocket, &readfds)) {
15            memset(buffer, 0, sizeof(buffer));
16            ssize_t len = recv(clientSocket, buffer, sizeof(buffer), 0);
17            if (len <= 0) {
18                isRunning = false;
19                if (!exited) {
20                    std::cerr << "Error: Failed to receive message or connection
                    ↳ closed." << std::endl;
```

```
21         }
22         break;
23     }
24     plainText = nullptr;
25     plainTextLength = -1;
26     des.Decrypt(buffer, len, plainText, plainTextLength);
27
28     // 检查退出命令
29     if (strcmp(plainText, EXIT_COMMAND) == 0) {
30         isRunning = false;
31         delete[] plainText;
32         std::cout << info << " exited." << std::endl;
33         break;
34     }
35
36     std::cout << info << ": " << plainText << std::endl;
37     delete[] plainText;
38 }
39 else if(ret < 0) {
40     isRunning = false;
41     std::cerr << "Error: select() failed." << std::endl;
42     break;
43 }
44 }
45 }
```

该函数实现了 Chat 类的接收线程函数 ReceiveThread，用于在独立线程中循环接收并处理消息。下面我们分析一下具体的代码：

1. 网络通信管理

- (a) 使用 select() 监听 socket，设置 1 秒超时 (tv_sec = 1)
- (b) 通过 FD_ISSET 检测可读事件后：
 - 调用 recv() 接收数据到 buffer
 - 当 len <= 0 时判定连接异常

2. 数据解密处理

- (a) 解密流程：
 - 调用 des.Decrypt() 解密 buffer
 - 结果存入动态分配的 plainText
- (b) 后续处理：
 - 检测 EXIT_COMMAND 命令终止线程

- 打印普通消息: `std::cout << info << ": " << plainText`

3. 线程控制与错误处理

(a) 控制机制:

- 通过 `isRunning` 标志控制线程运行
- 使用 `exited` 避免重复报错

(b) 错误处理:

- `select()` 失败时立即终止
- 确保调用 `delete[] plainText` 释放内存

RunServer 函数

```
1 void Chat::RunServer() {
2     isServer = true;
3
4     // 创建 serverSocket
5     serverSocket = socket(AF_INET, SOCK_STREAM, 0);
6     if (serverSocket < 0) {
7         std::cerr << "Error: Failed to create socket." << std::endl;
8         return;
9     }
10    // 设置 serverSocket 为非阻塞
11    setNonBlocking(serverSocket);
12
13    struct sockaddr_in serverAddr;
14    serverAddr.sin_family = AF_INET;
15    serverAddr.sin_port = htons(serverPort);
16    serverAddr.sin_addr.s_addr = INADDR_ANY; // 接受任意 IP
17
18    if (bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0)
19        ↪ {
20        std::cerr << "Error: Failed to bind." << std::endl;
21        return;
22    }
23
24    if (listen(serverSocket, 1) < 0) {
25        std::cerr << "Error: Failed to listen." << std::endl;
26        return;
27    }
28
29    // 使用 select() 等待连接到来
30    fd_set readfds;
```

```
30     FD_ZERO(&readfds);
31     FD_SET(serverSocket, &readfds);
32     timeval tv;
33     tv.tv_sec = 10;    // 等待 10 秒
34     tv.tv_usec = 0;
35     int ret = select(serverSocket + 1, &readfds, NULL, NULL, &tv);
36     if(ret <= 0) {
37         std::cerr << "Error: No incoming connection within timeout." << std::endl;
38         return;
39     }
40
41     struct sockaddr_in clientAddr;
42     socklen_t clientAddrLen = sizeof(clientAddr);
43     clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr,
44         ↪ &clientAddrLen);
45     if (clientSocket < 0) {
46         std::cerr << "Error: Failed to accept." << std::endl;
47         return;
48     } else {
49         std::cout << "Client connected." << std::endl;
50     }
51     // 设置 clientSocket 为非阻塞模式
52     setNonBlocking(clientSocket);
53
54     // RSA 密钥生成 (最多重试 3 次)
55     const int N_RETRY = 3;
56     for (int i = 0; i < N_RETRY; i++) {
57         if (rsa.GenerateKey()) {
58             std::cout << "RSA key generated successfully." << std::endl;
59             break;
60         }
61         if (i == N_RETRY - 1) {
62             std::cerr << "Error: Failed to generate RSA key." << std::endl;
63             std::cerr << "Server Exiting..." << std::endl;
64             return;
65         } else {
66             std::cerr << "Warning: Failed to generate RSA key. Retrying..." <<
67                 ↪ std::endl;
68         }
69     }
70
71     // 显示 RSA 详细配置信息
```

```
70     rsa.PrintConfig();
71
72     // 发送公钥和模数给客户端
73     uint64_t e = rsa.GetPublicKey();
74     uint64_t n = rsa.GetModulus();
75     if (send(clientSocket, reinterpret_cast<const char*>(&e), sizeof(e), 0) < 0) {
76         std::cerr << "Error: Failed to send public key." << std::endl;
77         return;
78     }
79     if (send(clientSocket, reinterpret_cast<const char*>(&n), sizeof(n), 0) < 0) {
80         std::cerr << "Error: Failed to send modulus." << std::endl;
81         return;
82     }
83
84     // 使用 select() 等待客户端发送 DES 密钥 (加密后的 DES key)
85     FD_ZERO(&readfds);
86     FD_SET(clientSocket, &readfds);
87     tv.tv_sec = 5; // 等待 5 秒
88     tv.tv_usec = 0;
89     ret = select(clientSocket + 1, &readfds, NULL, NULL, &tv);
90     if (ret <= 0) {
91         std::cerr << "Error: Timeout waiting for DES key." << std::endl;
92         return;
93     }
94
95     uint64_t desKey_enc[8];
96     if (recv(clientSocket, reinterpret_cast<char*>(desKey_enc),
97         ↪ sizeof(desKey_enc), 0) < 0) {
98         std::cerr << "Error: Failed to receive DES key." << std::endl;
99         return;
100     }
101
102     uint8_t desKey[8];
103     for (int i = 0; i < 8; i++) {
104         desKey[i] = rsa.Decrypt(desKey_enc[i]);
105     }
106     des.SetKey((char*)desKey);
107
108     std::cout << "Key exchange completed." << std::endl;
109     std::cout << "You can start chatting now." << std::endl;
110
111     auto chatLoop = [this]() {
```

```
111         isRunning = true;
112         receiveThread = std::thread(&Chat::ReceiveThread, this);
113         while (isRunning) {
114             Send();
115         }
116         Close();
117     };
118
119     chatLoop();
120 }
```

该函数主要完成了服务器的启动和通信的操作，下面我们对其代码进行详细的分析。

1. 服务器初始化

(a) 创建非阻塞式服务器套接字

- 使用 `socket(AF_INET, SOCK_STREAM, 0)` 创建 TCP 套接字
- 调用 `setNonBlocking()` 设置为非阻塞模式

(b) 绑定并监听端口

- 绑定到 `INADDR_ANY` 和指定端口 `serverPort`
- 开始监听 (`listen()`) 并设置最大连接数为 1

2. 客户端连接处理

(a) 使用 `select()` 等待连接 (10 秒超时)

(b) 接受连接 (`accept()`) 并设置非阻塞模式

- 失败时输出错误信息并退出
- 成功时打印 "Client connected"

3. RSA 密钥交换

(a) 生成 RSA 密钥 (最多重试 3 次)

- 调用 `rsa.GenerateKey()` 生成密钥对
- 失败时输出警告并重试

(b) 发送公钥参数

- 发送公钥 `e` 和模数 `n` 给客户端
- 使用 `reinterpret_cast` 转换数据类型

4. DES 密钥交换

(a) 等待接收加密的 DES 密钥 (5 秒超时)

(b) 使用 RSA 解密 DES 密钥

- 接收 8 个加密的 64 位块 (`desKey_enc`)

- 逐块解密得到 8 字节 DES 密钥
- (c) 初始化 DES 加密器
- 调用 `des.SetKey()` 设置解密后的密钥

5. 聊天主循环

- (a) 启动接收线程 (`ReceiveThread`)
- (b) 主线程处理消息发送 (`Send()`)
- (c) 终止时调用 `Close()` 清理资源

RunClient 函数

```
1 void Chat::RunClient() {
2     isServer = false;
3     Connect();
4
5     des.RandomGenKey();
6     uint8_t* desKey = des.GetKey();
7
8     // 等待服务器发来公钥和模数
9     fd_set readfds;
10    FD_ZERO(&readfds);
11    FD_SET(clientSocket, &readfds);
12    timeval tv;
13    tv.tv_sec = 5;    // 等待 5 秒
14    tv.tv_usec = 0;
15    int ret = select(clientSocket + 1, &readfds, NULL, NULL, &tv);
16    if(ret <= 0) {
17        std::cerr << "Error: Timeout waiting for public key and modulus." <<
18        ↵ std::endl;
19        return;
20    }
21
22    uint64_t e, n;
23    if (recv(clientSocket, reinterpret_cast<char*>(&e), sizeof(e), 0) < 0) {
24        std::cerr << "Error: Failed to receive public key." << std::endl;
25        return;
26    }
27    if (recv(clientSocket, reinterpret_cast<char*>(&n), sizeof(n), 0) < 0) {
28        std::cerr << "Error: Failed to receive modulus." << std::endl;
29        return;
30    }
```

```
31     uint64_t desKey_enc[8];
32     for (int i = 0; i < 8; i++) {
33         desKey_enc[i] = RSA::Encrypt((uint32_t)desKey[i], e, n);
34     }
35     delete[] desKey;
36
37     if (send(clientSocket, reinterpret_cast<const char*>(desKey_enc),
38         ↪ sizeof(desKey_enc), 0) < 0) {
39         std::cerr << "Error: Failed to send DES key." << std::endl;
40         return;
41     }
42
43     std::cout << "Key exchange completed." << std::endl;
44
45     auto chatLoop = [this]() {
46         isRunning = true;
47         receiveThread = std::thread(&Chat::ReceiveThread, this);
48         while (isRunning) {
49             Send();
50         }
51         Close();
52     };
53
54     chatLoop();
55 }
```

我们分析一下这个函数。该函数实现了客户端的运行和通信。首先，完成与服务器端的连接操作，然后等待服务器发过来的公钥和模数，等待五秒，进行 select 模式的连接操作。收到对应的公钥 e 和 n 后，客户端需要相应的进行验证操作，如果密钥对上了，就说明我们的密钥是正确的，这样就可以开始正常的通信了。我们还使用了 loop 聊天主循环来接收聊天，高效率地实现了我们所需的功能。

6 实验遇到的问题及其解决方法

6.1 对于 select 模型掌握不熟练

这基本上是我第一次接触 select 模型，在之前的《计算机网络》中也只是了解，并没有进行系统的实现。在本次作业中，通过查阅资料，辅助大模型，我编写出了基于 select 模型的通信系统，并顺利进行服务器和客户端的通信。主要遇到的问题有下面几个：

- 服务器连接后，客户端还未连接。导致连接失败。后续，我设计了延时等待机制，设置了 10 秒的延时，这样就可以保证我们服务器和客户端进行正常的通信。
- 服务器找不到对应的客户端进行密钥的传输和验证。后续，我使用 select() 在接收公钥和模数前等待数据到达，确保客户端能正确读取到公钥和模数数据，解决了问题。

6.2 对于 RSA 密钥生成错误的处理错误

对于 MillerRabin 算法生成的素数，是一个伪随机的素数，在开始时，我没有设计循环生成的操作，导致我们生成的数有一些不是素数。后期，我们修改了对应的素数的生成函数，进行 50 轮循环，确保我们生成的是一个素数。同时，设置三次的生成次数阈值，这样如果生成的 RSA 密钥不符合要求的话，我们可以进行重新生成，确保我们生成的数值是符合要求的。

7 实验结论

7.1 利用 cmake 编译项目

首先，我们使用命令行来对我们的整个项目进行编译。具体命令行如下所示：

```
1 cmake . -B build && cmake --build build
```

运行后，得到下面的结果，我们成功完成了编译，在 bin 文件夹下生成了一个可执行文件 RSA_chat。

```
(base) root@tomorin:/home/Network_Security_Technology/Lab02# cmake . -B build && cmake --build build
-- Configuring done
-- Generating done
-- Build files have been written to: /home/Network_Security_Technology/Lab02/build
Consolidate compiler generated dependencies of target RSA_chat
[ 20%] Building CXX object CMakeFiles/RSA_chat.dir/src/RSA_Operation.cpp.o
[ 40%] Linking CXX executable ../bin/RSA_chat
[100%] Built target RSA_chat
(base) root@tomorin:/home/Network_Security_Technology/Lab02#
```

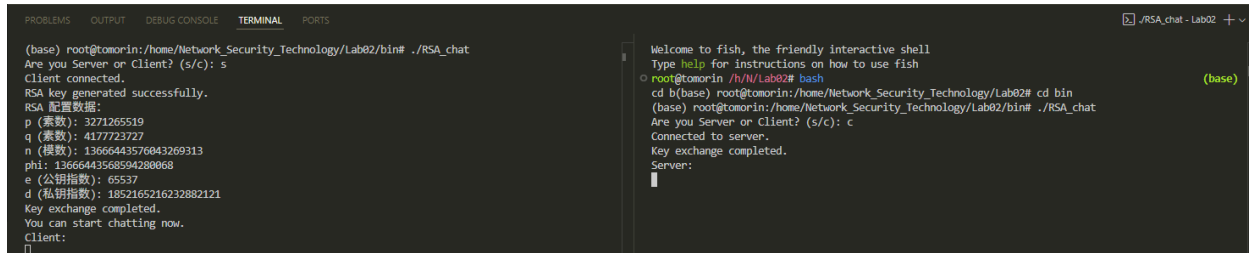
图 7.2: cmake 编译项目

7.2 新建服务器端和客户端

我们分别新开两个进程，进入 bin 目录，运行程序。具体的命令行如下所示：

```
1 cd bin
2 ./RSA_chat
```

得到下面的结果，如图所示：



```
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# ./RSA_chat
Are you Server or Client? (s/c): s
Client connected.
RSA key generated successfully.
RSA 配置数据:
p (素数): 3271265519
q (素数): 4177723727
n (模数): 13666443576043269313
phi: 13666443568594280068
e (公钥指数): 65537
d (私钥指数): 1852165216232882121
Key exchange completed.
You can start chatting now.
Client:
[]

Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/N/Lab02# bash
(base) root@tomorin:/home/Network_Security_Technology/Lab02# cd bin
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# ./RSA_chat
Are you Server or Client? (s/c): c
Connected to server.
Key exchange completed.
Server:
[]
```

图 7.3: 新建服务器和客户端并完成连接

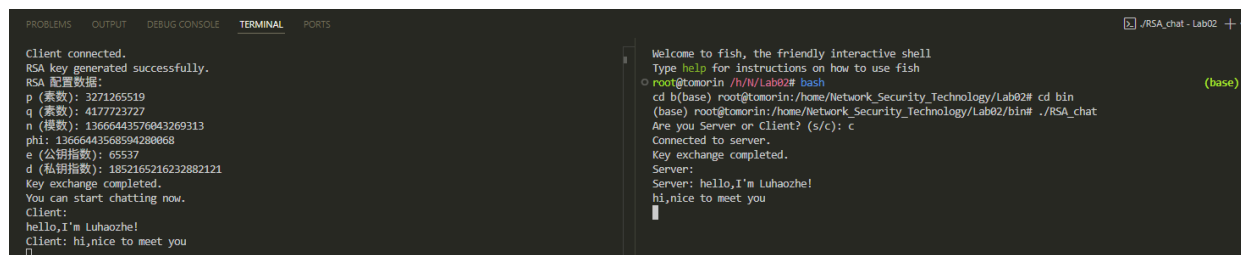
可以看到，我们成功完成了服务器和客户端的连接，并且双方都在等待对方的消息。而且，在建立连接之后，服务器已经完成了 RSA 公钥私钥指数的生成，并将公钥发送给了客户端，客户端也对应完成了验证。

7.3 功能测试

下面进行本次实验的功能测试，主要测试内容包括英文输入测试、中文输入测试和断开连接测试。下面展示我们的测试结果。

7.3.1 英文输入测试

我们分别在服务器端和客户端进行输入英文测试，发现互相的通信都没有问题，如下图所示：



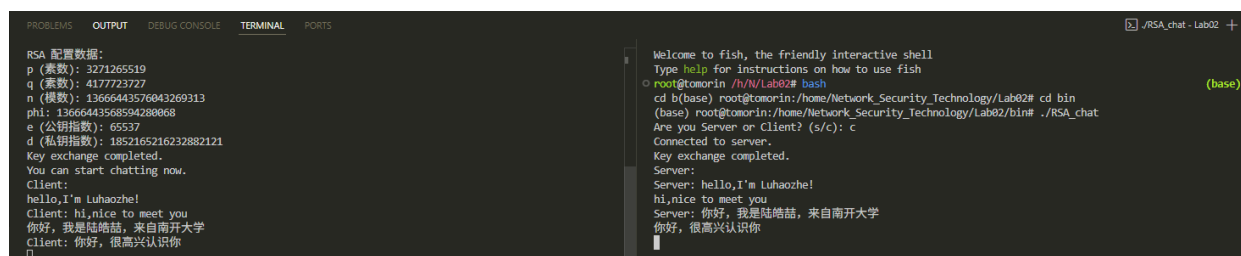
```
Client connected.
RSA key generated successfully.
RSA 配置数据:
p (素数): 3271265519
q (素数): 4177723727
n (模数): 13666443576043269313
phi: 13666443568594280068
e (公钥指数): 65537
d (私钥指数): 1852165216232882121
Key exchange completed.
You can start chatting now.
Client:
hello,I'm Luhaozhe!
Client: hi,nice to meet you
[]

Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/N/Lab02# bash
cd b(base) root@tomorin:/home/Network_Security_Technology/Lab02# cd bin
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# ./RSA_chat
Are you Server or Client? (s/c): c
Connected to server.
Key exchange completed.
Server:
Server: hello,I'm Luhaozhe!
hi,nice to meet you
```

图 7.4: 英文输入测试

7.3.2 中文输入测试

我们接着进行中文的输入测试。我们在服务器端和客户端分别输入一些中文，也发现通信是没有问题的。



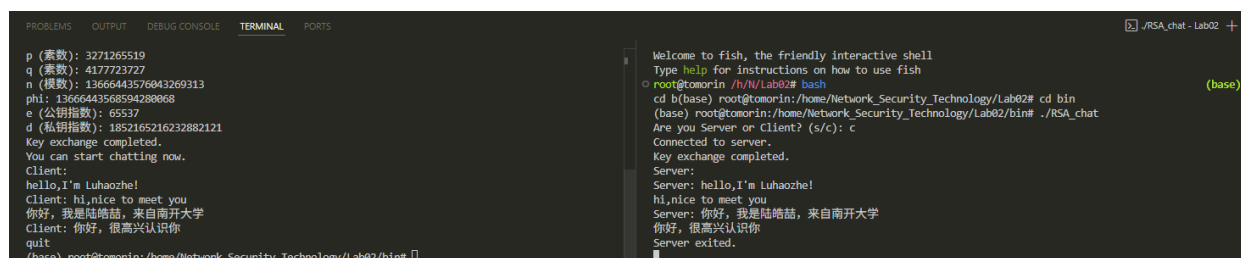
```
RSA 配置数据:
p (素数): 3271265519
q (素数): 4177723727
n (模数): 13666443576043269313
phi: 13666443568594280068
e (公钥指数): 65537
d (私钥指数): 1852165216232882121
Key exchange completed.
You can start chatting now.
Client:
hello,I'm Luhaozhe!
Client: hi,nice to meet you
你好,我是陆皓喆,来自南开大学
Client: 你好,很高兴认识你
[]

Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/N/Lab02# bash
cd b(base) root@tomorin:/home/Network_Security_Technology/Lab02# cd bin
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# ./RSA_chat
Are you Server or Client? (s/c): c
Connected to server.
Key exchange completed.
Server:
Server: hello,I'm Luhaozhe!
hi,nice to meet you
Server: 你好,我是陆皓喆,来自南开大学
你好,很高兴认识你
```

图 7.5: 中文输入测试

7.3.3 断开连接测试

最后,我们进行断开连接测试。我们在服务器端输入 quit,进行断开测试,发现服务器端成功地退出了进程,另外一端的客户端也不能正常通信,实验成功。



```
p (素数): 3271265519
q (素数): 4177723727
n (模数): 13666443576043269313
phi: 13666443568594280068
e (公钥指数): 65537
d (私钥指数): 1852165216232882121
Key exchange completed.
You can start chatting now.
Client:
hello,I'm Luhaozhe!
Client: hi,nice to meet you
你好,我是陆皓喆,来自南开大学
Client: 你好,很高兴认识你
quit
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# []

Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/N/Lab02# bash
cd b(base) root@tomorin:/home/Network_Security_Technology/Lab02# cd bin
(base) root@tomorin:/home/Network_Security_Technology/Lab02/bin# ./RSA_chat
Are you Server or Client? (s/c): c
Connected to server.
Key exchange completed.
Server:
Server: hello,I'm Luhaozhe!
hi,nice to meet you
Server: 你好,我是陆皓喆,来自南开大学
你好,很高兴认识你
Server exited.
```

图 7.6: 断开连接测试

这样,我们全部的测试环节就结束了,顺利的完成了我们的实验!

8 代码结构说明

本次实验，本人采用 makefile 的方法来进行框架的构建，具体的代码请见文件夹。主要结构如下所示：

```
Lab02
├── .gitignore
├── CMakeLists.txt
├── main.cpp
├── README.md
├── src
│   ├── chat.cpp
│   ├── DES_Operation.cpp
│   └── RSA_Operation.cpp
├── include
│   ├── chat.h
│   ├── DES_Operation.h
│   └── RSA_Operation.h
├── bin
│   └── RSA_chat
└── build
```

9 参考文献

- 第四章基于 RSA 算法自动分配密钥的加密聊天程序
- 公开密钥加密之 RSA 算法【概念 + 计算 + 代码实现】
- RSA 加密、解密、签名、验签（验证签名）&RSA 算法原理
- RSA 加密/解密 - 锤子在线工具
- C++ select 模型详解（多路复用 IO）