南開大學

网络安全技术 课程实验报告

基于 *RSA* 算法自动分配密钥的加密聊天程序

学院　网络空间安全学院

专业　信息安全

姓名　齐明杰

学号　2113997

2024 年 4 月 2 日

目　　录

1 实验目的

在讨论了传统的对称加密算法 DES 原理与实现技术的基础上，本章将以典型的非对称密码体系中 RSA 算法为例，以基于 TCP 协议的聊天程序加密为任务，系统地进行非对称密码体系 RSA 算法原理与应用编程技术的讨论和训练。

通过练习达到以下的训练目的：

① 加深对 RSA 算法基本工作原理的理解。
② 掌握基于 RSA 算法的保密通信系统的基本设计方法。
③ 掌握在 Linux 操作系统实现 RSA 算法的基本编程方法。
④ 了解 Linux 操作系统异步 IO 接口的基本工作原理。

2 实验内容

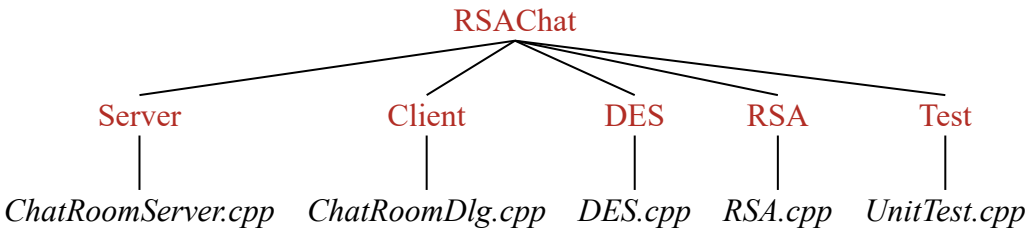
本章编程训练的要求如下：

① 要求在 Linux 操作系统中完成基于 RSA 算法的自动分配密钥加密聊天程序的编写。
② 应用程序保持第三章“基于 DES 加密的 TCP 通信”中示例程序的全部功能，并在此 基础上进行扩展，实现密钥自动生成，并基于 RSA 算法进行密钥共享。
③ 要求程序实现全双工通信，并且加密过程对用户完全透明。
④ 有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。

3 实验步骤及实验结果

3.1 项目结构

本次实验我实现了一个基于RSA算法和DES算法的加密多人聊天室。鉴于Windows平台编程和编译运行更便捷，我在本机 Win10 平台使用 Visual Studio 进行编程。
项目结构如下：



我将项目分为五个模块，分别是**服务端、客户端、DES 模块、RSA 模块和单元测试模块**，各个模块的功能如下：

模块	功能
Server	实现 TCP 通信，生成 RSA 公私钥，分发公钥，与多个客户端交互
Client	实现 TCP 通信，生成 DES 密钥，与服务端交互
DES	实现 DES 加密解密并提供接口
RSA	实现 RSA 加密解密并提供接口
Test	对 DES 和 RSA 模块进行单元测试

表 3.1.1: 各模块功能

3.2 DES 模块

DES 模块的加解密流程如下图：

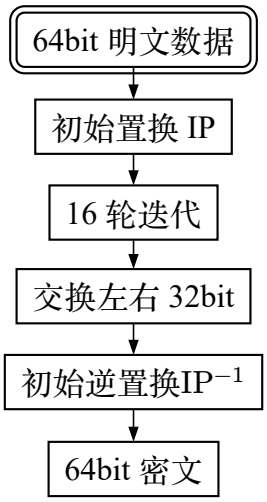


图 3.2.2: DES 加解密流程

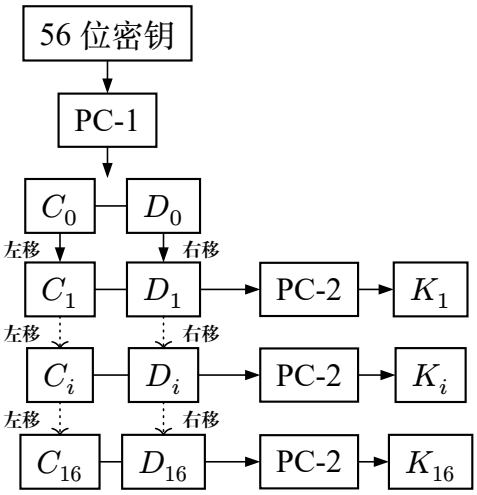


图 3.2.3: 子密钥生成

DES 算法的具体实现在上一次实验已经详细描述，不再赘述，仅给出使用的接口 (DES.h):

```cpp
#pragma once

#include <vector>
#include <bitset>
#include <cstdint>
#include <string>

class DES {
  static const uint8_t IP[64];              // 第一轮置换
  static const uint8_t FP[64];              // 最后一轮置换
  static const uint8_t E_box[48];           // 拓展运算 E 盒
  static const uint8_t P_box[32];           // 置换运算 P 盒
  static const uint8_t S_box[8][4][16];     // 8 个 S 盒
  static const uint8_t PC_1[56];            // PC-1 置换
  static const uint8_t PC_2[48];            // 密钥压缩置换表
  static const uint8_t shift[16];           // 每轮左移的位数
```

```
17
18  public:
19    static std::string generateKey();                    // 生成随机密钥
20    DES(const std::string& key);                         // 构造函数，需要 64 位密钥作为参数
21    std::vector<uint8_t> encrypt(const std::vector<uint8_t>& plaintext);  // 加密和解密接口
22    std::vector<uint8_t> decrypt(const std::vector<uint8_t>& ciphertext);
23    static std::vector<uint8_t> strToVec(const std::string& input) {
24      return std::vector<uint8_t>(input.begin(), input.end());
25    }
26    static std::string vecToStr(const std::vector<uint8_t>& input) {
27      return std::string(input.begin(), input.end());
28    }
29
30  private:
31    enum MODE { ENCRYPT, DECRYPT };
32    std::bitset<48> subKeys[16];                         // 存储生成的 16 轮子密钥
33    std::bitset<64> execute(const std::bitset<64>& data, int mode);      // 加密/解密
34    void genSubKeys(const std::bitset<64>& key);                 // 生成 16 轮子密钥
35    std::bitset<32> f(const std::bitset<32>& R, const std::bitset<48>& K);  // f 函数
36    template<size_t N>
37    std::bitset<N> leftRotate(const std::bitset<N>& bits, int shift);    // 循环左移
38    template<size_t N>
39    uint8_t get(const std::bitset<N>& b, size_t pos) { return b[N - 1 - pos]; }
40    template<size_t N>
41    void set(std::bitset<N>& b, size_t pos, size_t value) { b[N - 1 - pos] = value; }
42      std::vector<uint8_t> pad(const std::vector<uint8_t>& data, size_t blockSize); // PKCS#7 填充
43      std::vector<uint8_t> unpad(const std::vector<uint8_t>& data, size_t blockSize); // PKCS#7 去填充
44  };
45
```

其中，相较于上一次实验，增加了生成随机密钥的接口：

```cpp
1  std::string DES::generateKey() {
2    std::random_device rd;
```

```
3     std::mt19937_64 gen(rd());
4     std::uniform_int_distribution<uint64_t> dis(0, 15);
5     std::string key;
6     for (int i = 0; i < 16; i++) {
7       key.push_back("0123456789ABCDEF"[dis(gen)]);
8     }
9     return key;
10  }
```

使用了 C++11 的随机数库生成 16 位 16 进制密钥，至于为什么不使用 rand()，请参照 **problem_2**。

3.3 RSA 模块

RSA 加密算法是一种典型的公钥加密算法。RSA 算法的可靠性建立在分解大整数的困难性上。假如找到一种快速分解大整数算法的话，那么用 RSA 算法的安全性会极度下降。但是存在此类算法的可能性很小。目前只有使用短密钥进行加密的 RSA 加密结果才可能被穷举解破。只要其钥匙的长度足够长，用 RSA 加密的信息的安全性就可以保证。作为非对称加密算法，RSA 提供了一种更加安全的加密方式，即公钥加密，私钥解密。相较于 DES，RSA 的密钥长度更长，安全性更高，但是速度更慢。因此，RSA 通常用于密钥交换，而不是加密数据。

RSA 模块比 DES 略显简单，我们只需要实现 RSA 的加解密和密钥生成即可，RSA 的加解密和生成密钥流程如下图：

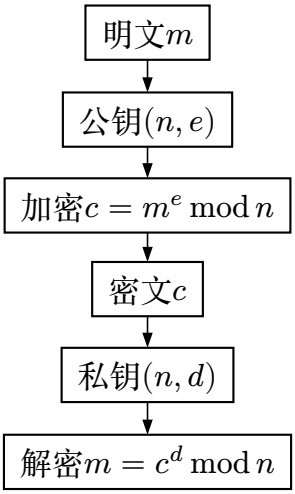


图 3.3.4: RSA 加解密流程

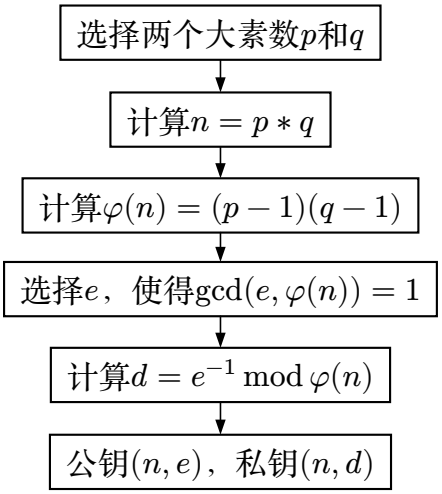


图 3.3.5: RSA 公私钥生成

在日常使用中，RSA 的安全性必须得到保证，这是由 n 的长度决定的，正常来说选择 1024 位的 RSA，但由于大长度的 RSA 实现比较麻烦，需要**涉及高精度运算**等知识，但这些并不会改变 RSA 的本质核心内容，因此为了提高效率，我选择了 16 位的 p 和 q，生成的 n 为 32 位，e 为 65537, RSA 的具体实现如下(RSA.h)： 首先定义了一个公私钥结构体：

```cpp
struct PublicKey {
  uint64_t n, e;
};

struct PrivateKey {
  uint64_t n, d;
};
```

分别用 uint64_t 类型存储 n, e, d。

然后定义了 RSA 类：

```cpp
class RSA {
public:
  static const int BLOCK_SIZE = 2;
  static void generateKeys(PublicKey& pubkey, PrivateKey& prikey);     // 生成公私钥对
      static   std::vector<uint8_t>   encrypt(const   std::vector<uint8_t>& plaintext, PublicKey pubkey); // 使用公钥加密
      static   std::vector<uint8_t>   decrypt(const   std::vector<uint8_t>& ciphertext, PrivateKey prikey); // 使用私钥解密
  static uint64_t encrypt(uint64_t plaintext, PublicKey pubkey);
  static uint64_t decrypt(uint64_t ciphertext, PrivateKey prikey);
  static uint64_t modMul(uint64_t a, uint64_t b, uint64_t mod);
  static uint64_t modPow(uint64_t base, uint64_t exponent, uint64_t mod);
  static uint64_t modInverse(uint64_t a, uint64_t m);          // 求模逆元
  static void exgcd(int64_t a, int64_t b, int64_t& x, int64_t& y);     // 扩展欧几里得算法
  static uint64_t gcd(uint64_t a, uint64_t b);
  static bool millerRabin(uint64_t n, int iter);  // Miller-Rabin 素性测试
  static uint64_t genPrime(int bits);       // 生成一个大素数
  static std::vector<uint8_t> strToVec(const std::string& hex_str);
  static std::string vecToStr(const std::vector<uint8_t>& data);
```

下面是对其中各个重要方法的解释：

3.3.1 generateKeys

```cpp
void RSA::generateKeys(PublicKey& pubkey, PrivateKey& prikey) {
  uint64_t p = RSA::genPrime(16);
  uint64_t q = RSA::genPrime(16);
  uint64_t n = p * q;
  pubkey.n = prikey.n = n;
  uint64_t phi = (p - 1) * (q - 1);
  uint64_t e = 65537;
  while (RSA::gcd(e, phi) != 1) e += 2;
```

```
9     pubkey.e = e;
10    prikey.d = RSA::modInverse(e, phi);
11  }
```

generateKeys 函数用于生成 RSA 公私钥对，首先生成两个 16 位大素数 p 和 q，然后计算$n = p * q$，$\varphi(n) = (p-1)(q-1)$，选择$e = 65537$，然后计算$d = e^{-1} \bmod \varphi(n)$，最后得到公钥$(n, e)$和私钥$(n, d)$。

3.3.2 modInverse

```cpp
1   uint64_t RSA::modInverse(uint64_t a, uint64_t m) {
2     int64_t x, y;
3     RSA::exgcd(a, m, x, y);
4     x = (x + m) % m;
5     return x;
6   }
7
8   void RSA::exgcd(int64_t a, int64_t b, int64_t& x, int64_t& y) {
9     if (b == 0) x = 1, y = 0;
10    else exgcd(b, a % b, y, x), y -= (a / b) * x;
11  }
```

modInverse 函数用于求模逆元，即对于给定的a和m，求出一个x，使得$a * x \equiv 1 (\bmod m)$。这里使用了**扩展欧几里得算法**，即 exgcd，求出 x 和 y，然后 x 为所求。

3.3.3 modMul & modPow

```cpp
1   uint64_t RSA::modMul(uint64_t a, uint64_t b, uint64_t mod) {
2     uint64_t result = 0;
3     a %= mod;
4     while (b > 0) {
5       if (b & 1) {
6         result = (result + a) % mod;
7       }
8       a = (2 * a) % mod;
9       b >>= 1;
10    }
11    return result;
12  }
13
14  uint64_t RSA::modPow(uint64_t base, uint64_t exp, uint64_t mod) {
15    uint64_t result = 1;
16    base = base % mod;
17    while (exp > 0) {
18      if (exp & 1) {
```

```
19        result = RSA::modMul(result, base, mod);
20      }
21      base = RSA::modMul(base, base, mod);
22      exp >>= 1;
23    }
24    return result;
25  }
```

modMul 函数用于求$a * b \bmod m$，*modPow* 函数用于求$\text{base}^{\text{exp}} \bmod m$。采用了**快速幂**的方法，减少了计算量。

3.3.4 millerRabin

```cpp
1   bool RSA::millerRabin(uint64_t n, int iter) {
2     if (n < 4) return n == 2 || n == 3;
3     if (n % 2 == 0) return false;
4     // 写 n - 1 为 2^s * d 的形式
5     uint64_t s = 0;
6     uint64_t d = n - 1;
7     while ((d & 1) == 0) {
8       d >>= 1;
9       ++s;
10    }
11    std::uniform_int_distribution<uint64_t> distribution(2, n - 2);
12    std::random_device rd;
13    std::mt19937_64 gen(rd());
14    for (int i = 0; i < iter; i++) {
15      uint64_t a = distribution(gen);
16      uint64_t x = RSA::modPow(a, d, n);
17      if (x == 1 || x == n - 1) continue;
18      for (uint64_t j = 1; j < s; j++) {
19        x = RSA::modMul(x, x, n);
20        if (x == n - 1) break;
21      }
22      if (x != n - 1) return false;
23    }
24    return true;
25  }
```

millerRabin 函数用于进行 **miller-rabin 素性测试**，判断一个数是否为素数，其中 iter 为迭代次数，迭代次数越多，判断越准确，但是耗时也越长。

3.3.5 genPrime

```cpp
1   uint64_t RSA::genPrime(int bits) {
```

```
2     static std::random_device rd;
3     static std::mt19937_64 gen(rd());
4     uint64_t hbit = 1ULL << (bits - 1);
5     uint64_t lbit = 1ULL << bits;
6     std::uniform_int_distribution<uint64_t> dis(hbit, lbit - 1);
7     uint64_t n = 0;
8     do {
9       n = dis(gen);
10    } while (!RSA::millerRabin(n, 100));
11    return n;
12  }
```

genPrime 函数用于生成一个大素数，首先生成一个 bits 位的随机数，然后进行 miller-rabin 素性测试，直到生成一个素数。

值得注意的是，这里的 miller-rabin 素性测试迭代次数为 100，这是一个比较保守的选择，迭代次数越多，判断越准确，但是耗时也越长。

另外，我使用了 *std::mt19937_64* 和 *std::uniform_int_distribution*，这是 C++11 的随机数库，比 rand()更加安全，**更加随机**。

3.3.6 strToVec & vecToStr

```cpp
1   std::vector<uint8_t> RSA::strToVec(const std::string& hex_str) {
2     if (hex_str.length() % 2 != 0) {
3       throw std::invalid_argument("Hex string must have an even length");
4     }
5     std::vector<uint8_t> bytes;
6     for (size_t i = 0; i < hex_str.length(); i += 2) {
7       std::string byteString = hex_str.substr(i, 2);
8      uint8_t byte = static_cast<uint8_t>(std::stoi(byteString, nullptr, 16));
9       bytes.push_back(byte);
10    }
11    return bytes;
12  }
13
14  std::string RSA::vecToStr(const std::vector<uint8_t>& data) {
15    std::string hex_str;
16    hex_str.reserve(data.size() * 2);
17    for (uint8_t byte : data) {
18      char high = "0123456789ABCDEF"[byte >> 4];
19      char low = "0123456789ABCDEF"[byte & 0x0F];
20      hex_str.push_back(high);
21      hex_str.push_back(low);
22    }
23    return hex_str;
```

```
24 }
```

strToVec 函数用于将 16 进制字符串转换为字节流，*vecToStr* 函数用于将字节流转换为 16 进制字符串。 转换方法：

- *strToVec*：将 16 进制字符串每两个字符(即 1 字节)转化为字节单位，然后存入。
- *vecToStr*：将每个 uint8_t 即 1 字节分成两个 16 进制字符存储输出。

3.3.7 encrypt

```cpp
std::vector<uint8_t> RSA::encrypt(const std::vector<uint8_t>& plaintext, PublicKey pubkey) {
  std::vector<uint8_t> ciphertext;
  uint64_t block = 0;
  int byte_count = 0;

  for (size_t i = 0; i < plaintext.size(); i++) {
    block = (block << 8) | plaintext[i];
    byte_count++;

    if (byte_count == RSA::BLOCK_SIZE || i == plaintext.size() - 1) {
      uint64_t cipher = RSA::modPow(block, pubkey.e, pubkey.n);
      for (int j = 7; j >= 0; j--) {
        ciphertext.push_back((cipher >> (j * 8)) & 0xff);
      }
      block = 0;
      byte_count = 0;
    }
  }

  return ciphertext;
}
```

这个函数将明文数据分成固定大小的块，每个块依次通过左移和或操作进行数字化，然后使用公钥 e 和 n 进行模幂运算加密。这一过程在达到块大小或处理完所有明文数据时进行，每次处理都将加密后的数据块转换为字节序列，最终形成完整的密文。

需要注意的是，每个明文块加密成的密文块长度为 8 字节，即 64 位。

3.3.8 decrypt

```cpp
std::vector<uint8_t> RSA::decrypt(const std::vector<uint8_t>& ciphertext, PrivateKey prikey) {
  std::vector<uint8_t> plaintext;
  uint64_t block = 0;
  int byte_count = 0;
```

```
5
6     for (size_t i = 0; i < ciphertext.size(); i++) {
7       block = (block << 8) | ciphertext[i];
8       byte_count++;
9
10      if (byte_count == 8 || i == ciphertext.size() - 1) {
11        uint64_t plain = RSA::modPow(block, prikey.d, prikey.n);
12        for (int j = RSA::BLOCK_SIZE - 1; j >= 0; j--) {
13          plaintext.push_back((plain >> (j * 8)) & 0xff);
14        }
15        block = 0;
16        byte_count = 0;
17      }
18    }
19
20    return plaintext;
21  }
```

解密函数遍历加密的密文，将其分成固定大小的块(即 8 字节)，并对每个块使用私钥 d 和 n 进行模幂运算解密。每个密文块解密后，通过右移和与操作转换回明文字节(**对应明文块的大小**)，直到所有的密文块被处理完毕，最终拼接成完整的明文数据。

至于为什么使用 **std::vector<uint8_t>**作为接口，请参照 **problem_1**。

3.4 TCP 通信

在 RSA 与 DES 结合的加密通信流程中，首先使用 RSA 算法在客户端和服务端之间安全地传递 DES 的密钥。客户端生成 DES 密钥，并利用服务端的公钥对其进行 RSA 加密，然后通过 TCP 传送给服务端。服务端收到加密的 DES 密钥后，使用其私钥进行解密，从而获得原始的 DES 密钥。此后，双方便使用该 DES 密钥进行对称加密通信，确保数据传输的安全性。这种方法结合了 RSA 的安全密钥交换机制和 DES 的高效数据加密能力，适用于需要安全通信的网络应用。 结合 DES 和 RSA 的使用，TCP 通信的复杂流程如下图所示:

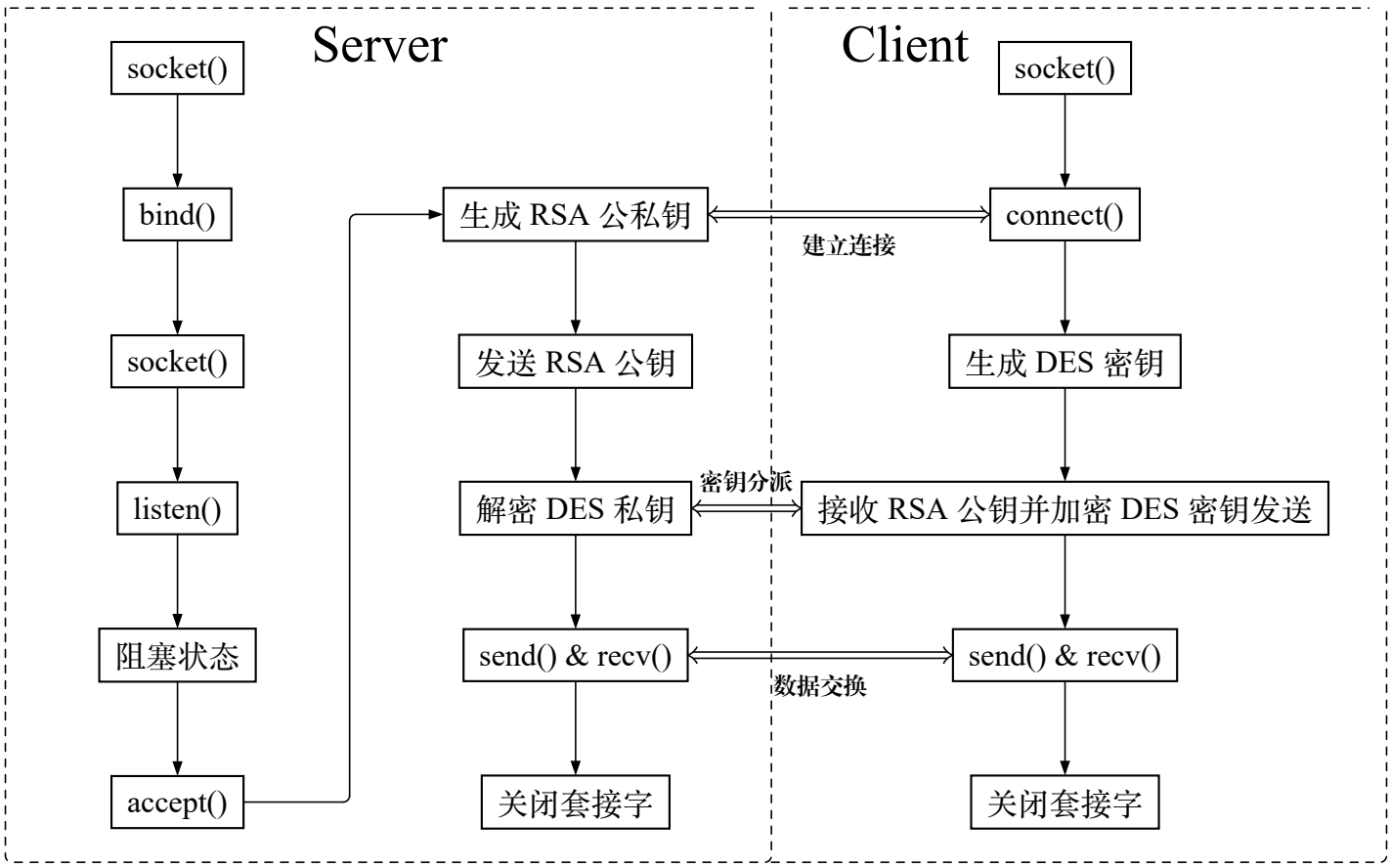


图 3.4.6: 带 RSA 的 TCP 通信

另外，我实现了一个多人聊天室，因此每个独立的**客户端均会生成自己的 DES 密钥**，并通过 RSA 加密后发送给服务端，服务端收到后解密，然后将其加入到一个密钥列表中，以便后续的通信。

3.4.1 服务端

定义了服务器类。由于我实现了多人聊天，故定义了客户结构体，由服务器来进行维护：

```cpp
// 定义客户结构体
struct Client {
    SOCKET sock;            // 套接字
    string username;        // 用户名
    Client(SOCKET sock = INVALID_SOCKET, string username = "$") : sock(sock), username(username) {}
};

// 聊天室服务器类
class ChatRoomServer {
public:
    ChatRoomServer(UINT port, UINT client);         // 构造函数
    ~ChatRoomServer();                              // 析构函数
```

```
13    void Start();                                          // 启动服务器
14    void Stop();                                                  // 关闭服务器
15    void PrintInfo(const string& info);                          // 输出日志
16
17  private:
18    // 定义服务器相关常量
19    UINT MAX_CLIENTS;                              // 最大客户端数量
20    UINT PORT;                        // 服务器端口
21    constexpr static UINT BUFFER_SIZE = 1024; // 缓冲区大小
22
23    // 定义服务器相关变量
24    SOCKET SockServer = INVALID_SOCKET;       // 服务器套接字
25    Client* clients;                          // 客户端数组
26    HANDLE* hThreads;                         // 线程句柄，每个客户端均有一个线程来处理
27    HANDLE hCommandThread;                  // 服务器命令线程句柄
28    UINT hpointer = 0;                          // 线程句柄数组的指针
29    sockaddr_in addrServer;                 // 服务器地址
30    bool shouldRun = true;                              // 用于标记服务器是否应继续运行
31    DES** des;                        // DES 加密解密对象数组
32    PublicKey pub_key;                  // 服务器公钥
33    PrivateKey pri_key;                 // 服务器私钥
34
35    // 定义服务器相关函数
36    void InitWinSock();                 // 初始化 WinSock
37    int find_pos();                           // 查找空闲的客户端存放位置
38    UINT Online_Count();                // 获取在线人数
39    static DWORD WINAPI ClientHandler(LPVOID pParam);    // 每个客户的线程函数
40    void BroadcastMessage(const string& msg);    // 将消息广播给所有客户端
41    string GetCurrTime();                               // 获取当前时间
42    static DWORD WINAPI ListenForCommand(LPVOID pParam);     // 监听服务器命令
43  };
```

相较于客户端，服务端的实现更加复杂，需要维护多个客户端的信息，因此定义了一个客户结构体，包含了客户端的套接字和用户名。服务端类中定义了多个私有成员变量，包括服务器套接字、客户端数组、线程句柄数组、服务器地址、DES 加密解密对象数组、服务器公私钥等。服务端类中定义了多个私有成员函数，包括初始化 WinSock、查找空闲的客户端存放位置、获取在线人数、客户端线程函数、将消息广播给所有客户端、获取当前时间、监听服务器命令等。

其中一些重要函数及其功能：

- ChatRoomServer::InitWinSock()

功能：这个函数用于启动 WinSock 2.2 版本。初始化过程中，如果发生错误，程序会输出错误信息并退出。

- ChatRoomServer::Online_Count()

功能：遍历客户端数组，统计并返回有效套接字的数量，从而得知当前在线的客户端数量。

- ChatRoomServer::Start()

功能：启动服务器的主循环，并准备接收来自客户端的连接。

- **创建服务器套接字**：首先使用 `socket()` 函数创建一个新的套接字。
- **绑定套接字**：使用 `bind()` 函数将新创建的套接字绑定到指定的 IP 地址和端口上。
- **开始监听**：通过 `listen()` 函数使服务器开始监听客户端的连接请求。
- **接受客户端连接**：使用 `accept()` 函数接受来自客户端的连接。对于每一个成功的连接，都会在服务器中为该客户端分配一个新的套接字。
- **创建客户端线程**：每当有新的客户端连接时，都会为这个客户端创建一个新的线程来处理它的消息。这确保了服务器能够并发地处理多个客户端。

- ChatRoomServer::ClientHandler(LPVOID pParam)

功能：为每一个连接的客户端独立执行，处理来自客户端的消息，并与其他客户端进行交互。

- **获取用户名**：首先，这个函数从客户端接收其用户名。这是客户端首次与服务器交互的部分。**获取的是加密后的用户名，需要调用 DES 模块来解密**，
- **发送欢迎消息**：为新连接的客户端发送一个欢迎消息，并广播给所有其他在线的客户端。
- **消息循环**：函数接着进入一个循环，不断地接收来自客户端的消息并**解密**，并将其广播给其他客户端。
- **断开连接处理**：如果客户端发送了退出消息或者由于某种原因与服务器断开了连接，该函数会广播这个客户端的退出消息，然后关闭与该客户端的连接。

代码如下：[

```cpp
// 每个客户的线程函数
DWORD WINAPI ChatRoomServer::ClientHandler(LPVOID pParam) {
    ChatRoomServer* pThis = reinterpret_cast<ChatRoomServer*>(pParam);
    std::vector<uint8_t> buffer(BUFFER_SIZE);
    int bytes;

    // 获取当前客户端
```

```
8     UINT num = pThis->hpointer;
9     Client* client = &pThis->clients[num];
10
11    // 发送 RSA 公钥
12    std::vector<uint8_t> pubkey(2 * sizeof(uint64_t));
13    *reinterpret_cast<uint64_t*>(pubkey.data()) = pThis->pub_key.n;
14    *reinterpret_cast<uint64_t*>(pubkey.data() + sizeof(uint64_t)) = pThis->pub_key.e;
15        send(client->sock,  reinterpret_cast<const  char*>(pubkey.data()), pubkey.size(), 0);
16    pThis->PrintInfo("已向客户[" + std::to_string(num) + "]发送 RSA 公钥.");
17
18    // 接收客户端的 DES 密钥并用 RSA 私钥解密
19    buffer.clear();
20    buffer.resize(BUFFER_SIZE);
21      bytes  =  recv(client->sock,  reinterpret_cast<char*>(buffer.data()), BUFFER_SIZE, 0);
22    if (bytes <= 0) {
23      pThis->PrintInfo("客户端异常断开.");
24      closesocket(client->sock);
25      return 0;
26    }
27    buffer.resize(bytes);
28    std::vector<uint8_t> decKey = RSA::decrypt(buffer, pThis->pri_key);
29    std::string des_key = RSA::vecToStr(decKey);
30    pThis->des[num] = new DES(des_key);
31    DES* des = pThis->des[num];
32     pThis->PrintInfo("已接收客户端[" + std::to_string(num) + "]DES 密钥: " + des_key);
33
34    // 读取客户端的用户名
35    buffer.clear();
36    buffer.resize(BUFFER_SIZE);
37      bytes  =  recv(client->sock,  reinterpret_cast<char*>(buffer.data()), BUFFER_SIZE, 0);
38    if (bytes <= 0) {
39      pThis->PrintInfo("客户端[" + std::to_string(num) + "]异常断开.");
40      closesocket(client->sock);
41      return 0;
42    }
43    // 使用 std::vector<uint8_t> 截取实际接收的数据长度
44    buffer.resize(bytes);
45
```

```
46    // 解密
47    client->username = DES::vecToStr(des->decrypt(buffer));
48
49    // 发送欢迎消息
50    string welcomeMsg = "欢迎 " + client->username + " 加入聊天室!";
51    pThis->PrintInfo(client->username + " 加入聊天室.");
52    pThis->BroadcastMessage("系统消息:" + welcomeMsg);
53
54    // 循环接收客户端消息
55    while (true) {
56      buffer.clear();
57      buffer.resize(BUFFER_SIZE);
58      // 接收客户端信息, 无需用户名
59        bytes = recv(client->sock, reinterpret_cast<char*>(buffer.data()),
pThis->BUFFER_SIZE, 0);
60      // 解密
61      buffer.resize(bytes);
62      string decmsg = DES::vecToStr(des->decrypt(buffer));
63
64      // 客户端发送退出消息或异常断开连接
65      if (bytes <= 0 || decmsg == "exit") {
66        // 广播客户端退出消息
67        string exitMsg = client->username + " 已退出聊天室." + "(当前在线人数: "
+ to_string(pThis->Online_Count() - 1) + ")";
68        pThis->PrintInfo(exitMsg);
69        pThis->BroadcastMessage("系统消息:" + exitMsg);
70
71        // 客户端断开连接
72        closesocket(client->sock);
73        client->sock = INVALID_SOCKET;
74        break;
75      }
76
77      // 正常广播消息 约定消息格式为 "用户名:消息内容"
78      string message = client->username + ":" + decmsg;
79     pThis->PrintInfo("正在广播来自 " + client->username + " 的消息: " + decmsg);
80      pThis->BroadcastMessage(message);
81    }
82    return 0;
83  }
```

该函数相较于上次 DES 实验，**加入了 RSA 相关代码**，因此长度略长，解析如下：

- **客户端识别：**函数开始时，通过传入的参数获取 ChatRoomServer 实例和特定客户端的信息，确保可以对特定客户端进行操作。
- **发送 RSA 公钥：**服务器生成 RSA 公私钥对后，将公钥发送给客户端。这允许客户端使用公钥加密其 DES 密钥，确保只有持有私钥的服务器能解密并获取这个 DES 密钥。
- **接收和解密 DES 密钥：**服务器接收客户端发送的经过 RSA 加密的 DES 密钥。使用服务器的 RSA 私钥对这些数据进行解密，获得客户端的原始 DES 密钥。这保证了 DES 密钥的安全传输，只有服务器能解密并获取客户端的 DES 密钥。
- **建立 DES 加密通道：**一旦获取 DES 密钥，服务器为该客户端实例化一个 DES 对象，并使用此密钥进行后续通信的加密和解密。这样，客户端与服务器之间的通信就通过 DES 加密通道保护起来，确保了通信内容的安全性。
- **接收和处理客户信息：**服务器接着接收客户端发送的加密信息，如用户名。使用先前获得的 DES 密钥对信息进行解密，获得明文信息，如客户端的用户名。
- **广播欢迎消息：**新客户加入后，服务器会广播一条欢迎消息给所有客户端，提示有新成员加入。
- **消息循环处理：**服务器进入循环，持续接收来自该客户端的消息。对于每条接收的消息，服务器先用 DES 解密，再进行处理。如果客户端发送退出指令或异常断开连接，服务器会处理相应的退出逻辑并广播通知其他客户端。
- **资源管理和错误处理：**函数中适时进行资源管理，如关闭套接字和清理分配的资源，确保即使在异常情况下也能正常释放资源。
- ChatRoomServer::BroadcastMessage(const string& msg)
- **遍历客户端**：遍历所有已连接的客户端。
- **发送消息**：**DES 加密后**使用 `send()` 函数将指定的消息发送给每一个在线的客户端。

代码如下：

```cpp
// 广播消息给所有客户端
void ChatRoomServer::BroadcastMessage(const string& msg) {
  for (UINT i = 0; i <= hpointer; i++) {
    // 将 string 转换为 vector<uint8_t>加密
    std::vector<uint8_t> encmsg = des[i]->encrypt(DES::strToVec(msg));
    if (clients[i].sock != INVALID_SOCKET) {
      // 发送数据
      send(clients[i].sock, reinterpret_cast<const char*>(encmsg.data()), encmsg.size(), 0);
    }
  }
}
```

- ChatRoomServer::ListenForCommand(LPVOID pParam)

功能：这个函数在一个单独的线程上运行，监听并处理来自服务器管理员在控制台上输入的命令。

- **命令循环**：函数在一个循环中不断地监听控制台的输入。
- **处理 `exit` 命令**：如果输入的命令是 `exit`，这个函数将修改服务器的状态变量使其停止运行，并关闭服务器套接字以使 `accept()` 函数返回并退出主循环。
- **处理 `count` 命令**：如果输入的命令是 `count`，这个函数将计算并输出当前在线的客户端数量。

3.4.2 客户端

客户端则负责与用户交互，发送和接收消息。本次客户端我沿用上次采用 **C++ MFC 可视化编程**实现。

> **功能概述**
>
> 1. 在上方输入**服务端 IP**，**服务端口**，然后输入自己的**用户名**，点击连接服务器即可连接服务端。
> 2. 聊天区会显示服务端发送来的信息，解析出用户名和消息后，拼接上当前时间进行**显示**。
> 3. 点击退出，即可**离开聊天区**，此时服务器和还在聊天区的客户会收到离开信息。
> 4. 在下面编辑框输入后，点击发送即可发送信息到服务器(不会直接显示在聊天区)。
>
> **支持:**
>
> 1. 多开客户端，在断连后重新连接服务器(之前的聊天内容不会清空)。
> 2. **多线程**管理，主线程负责控制用户和界面的交互，按钮点击事件等，子线程负责接受服务器的消息并转交给主线程打印处理。

界面设计：

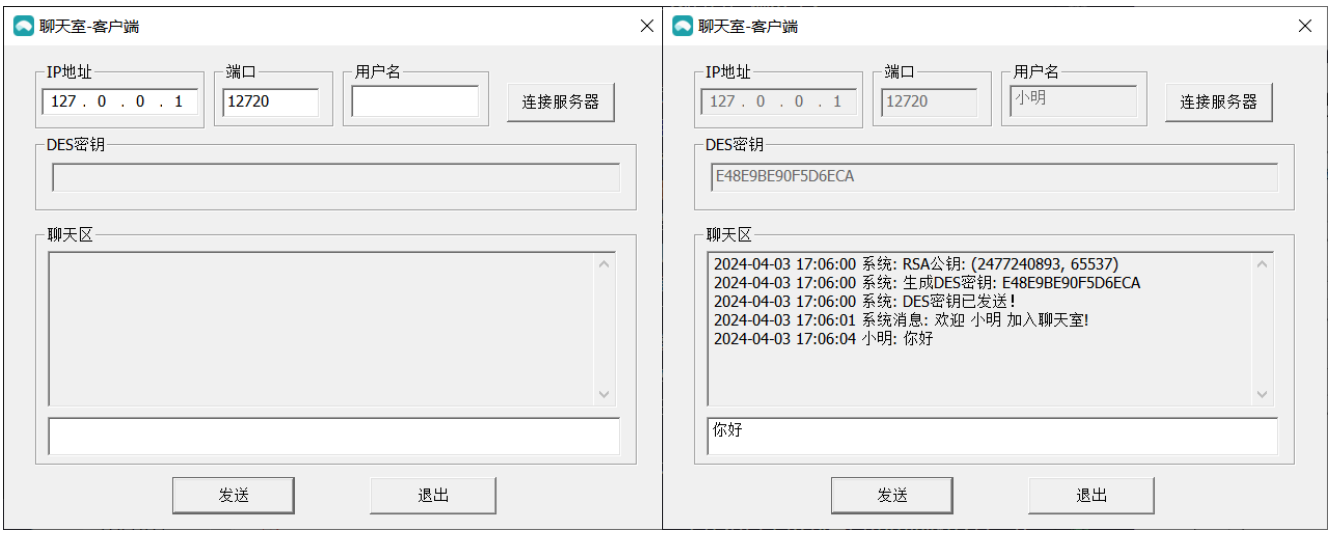


图 3.4.7: 客户端界面

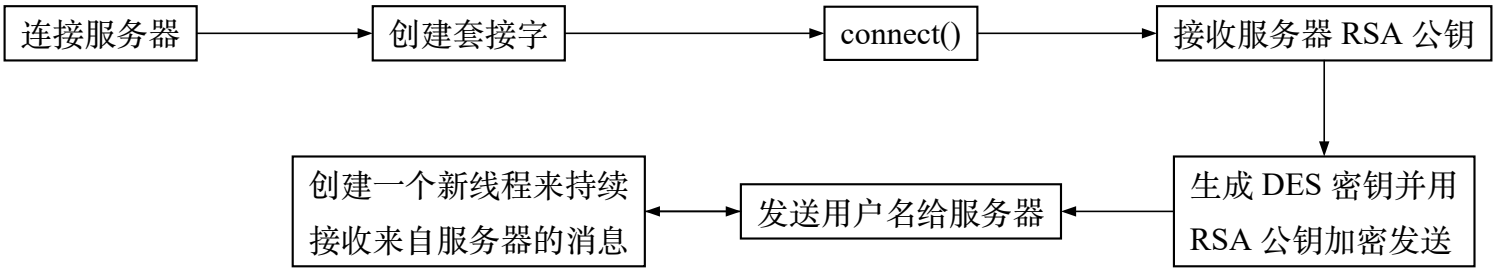


图 3.4.8: 连接服务器流程

，

部分重要函数变量声明如下：

```cpp
afx_msg void OnBnClickedButtonExit(); // 退出按钮
afx_msg void OnBnClickedButtonSend(); // 发送按钮
afx_msg void OnBnClickedButtonConnect(); // 连接按钮
static constexpr UINT BufferSize = 1024; // 缓冲区大小
virtual void OnClose(); // 重写关闭窗口函数
SOCKET SockClient = INVALID_SOCKET; // 客户端套接字
HANDLE hThread = NULL; // 线程句柄
CString UserName; // 用户名
CString key;                            // DES 密钥
DES* des;                              // DES 加密解密对象
PublicKey pub_key;                             // 公钥
void PrintMsg(const CString& Name, const CString& strMsg);//打印消息
static DWORD WINAPI ReceiveMessages(LPVOID pParam);//接收消息线程函数
LRESULT OnUpdateChatMsg(WPARAM wParam, LPARAM lParam); // 更新聊天消息
```

关键函数描述如下：

- OnBnClickedButtonExit()：退出客户端按钮点击事件

功能：当用户点击退出按钮时，此函数会被触发。它会首先确认用户真的想要退出，然后**关闭与服务器的套接字连接、终止消息接收线程**，释放 Winsock 资源，并关闭聊天窗口。

- OnBnClickedButtonSend()：发送消息按钮点击事件

功能：此函数处理用户的消息发送请求。它首先检查消息内容的有效性，然后调用 **DES 加密消息**并发送消息到服务器。如果发送失败，它会提醒用户，并允许用户重新设置连接的参数。

- OnBnClickedButtonConnect()：连接服务器按钮点击事件

功能：此函数处理用户的连接请求。它首先初始化 Winsock、获取 IP、端口和用户名，然后 尝试与服务器建立连接。一旦连接成功，它会发送用户名给服务器并启动一个新线程来接收服务器的消息。

- ReceiveMessages(LPVOID pParam)：接收消息线程函数

功能：此函数在单独的线程中运行，不断地从服务器接收消息。一旦接收到消息，进行 **DES 解密**，使用 OnUpdateChatMsg 函数将消息发送到主线程进行显示。

- OnUpdateChatMsg(WPARAM wParam, LPARAM lParam)：更新聊天消息函数

功能：这是一个消息处理函数，负责处理从 ReceiveMessages 线程发送来的消息，并在聊天窗口中显示它们。

其中相较于上次实验，修改最大的函数是 OnBnClickedButtonConnect()，增加了 RSA 相关代码，如下：

```cpp
// 连接服务器
void CChatRoomDlg::OnBnClickedButtonConnect() {
  // 判断是否已经连接
  if (SockClient != INVALID_SOCKET) {
    MessageBox("已经连接到服务器！");
    return;
  }

  // 初始化 Winsock
  WSADATA wsaData = { 0 };
  if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    MessageBox("初始化 Winsock 失败！");
    return;
  }

  // 获取 IP 和端口
  CString strIP, strPort;
```

```
18    GetDlgItemText(IDC_EDIT_PORT, strPort);
19    CIPAddressCtrl* pIP = (CIPAddressCtrl*)GetDlgItem(IDC_IPADDRESS);
20    {
21      BYTE nf1, nf2, nf3, nf4;
22      pIP->GetAddress(nf1, nf2, nf3, nf4);
23      strIP.Format("%d.%d.%d.%d", nf1, nf2, nf3, nf4);
24    }
25
26    // 获取用户名
27    GetDlgItemText(IDC_EDIT_NAME, UserName);
28
29    // 判断上述信息合法
30    if (strIP.IsEmpty() || strPort.IsEmpty() || UserName.IsEmpty()) {
31      MessageBox("请填写完整信息！");
32      WSACleanup();
33      return;
34    }
35
36    // 创建套接字
37    SockClient = socket(AF_INET, SOCK_STREAM, 0);
38    if (SockClient == INVALID_SOCKET) {
39      MessageBox("创建套接字失败！");
40      WSACleanup();
41      return;
42    }
43
44    // 设置服务器地址
45    sockaddr_in serverAddr;
46    serverAddr.sin_family = AF_INET;
47    serverAddr.sin_port = htons(_ttoi(strPort));
48                if       (inet_pton(AF_INET,       CT2A(strIP.GetBuffer()),
  &(serverAddr.sin_addr)) != 1) {
49      MessageBox("无效的 IP 地址！");
50      WSACleanup();
51      return;
52    }
53
54    // 连接服务器
55    if (connect(SockClient, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) ==
  SOCKET_ERROR) {
56      MessageBox("连接服务器失败！");
57      closesocket(SockClient);
58      SockClient = INVALID_SOCKET;
```

```
59      WSACleanup();
60      return;
61    }
62
63    // 接收服务器 RSA 公钥
64    {
65      std::vector<uint8_t> buffer(BUFFER_SIZE);
66      int ret = recv(SockClient, reinterpret_cast<char*>(buffer.data()), BUFFER_SIZE, 0);
67      if (ret <= 0) {
68        MessageBox("接收 RSA 公钥失败！");
69        closesocket(SockClient);
70        SockClient = INVALID_SOCKET;
71        WSACleanup();
72        return;
73      }
74      buffer.resize(ret);
75      this->pub_key.n = *reinterpret_cast<uint64_t*>(buffer.data());
76      this->pub_key.e = *reinterpret_cast<uint64_t*>(buffer.data() + sizeof(uint64_t));
77      PrintMsg("系统", CString(("RSA 公钥: (" + std::to_string(pub_key.n) + ", " + std::to_string(pub_key.e) + ")").c_str()));
78    }
79
80    // 生成 DES 密钥并用 RSA 公钥加密发送
81    {
82      CString tmp(DES::generateKey().c_str());
83      key = tmp;
84      PrintMsg("系统", "生成 DES 密钥: " + key);
85    }
86    des = new DES(key.GetBuffer());
87    GetDlgItem(IDC_DES_KEY)->SetWindowText(key);
88      std::vector<uint8_t> encKey = RSA::encrypt(RSA::strToVec(key.GetBuffer()), pub_key);
89      send(SockClient, reinterpret_cast<const char*>(encKey.data()), encKey.size(), 0);
90    PrintMsg("系统", "DES 密钥已发送！");
91
92    // 发送用户名给服务器
93      std::vector<uint8_t> encName = des->encrypt(DES::strToVec(UserName.GetBuffer()));
94      send(SockClient, reinterpret_cast<const char*>(encName.data()), encName.size(), 0);
```

```
95      MessageBox("连接成功!");
96
97      // IP, 端口, 用户名, 不再可编辑
98      GetDlgItem(IDC_IPADDRESS)->EnableWindow(FALSE);
99      GetDlgItem(IDC_EDIT_PORT)->EnableWindow(FALSE);
100     GetDlgItem(IDC_EDIT_NAME)->EnableWindow(FALSE);
101
102     // 创建一个新线程来持续接收来自服务器的消息
103     hThread = CreateThread(NULL, 0, ReceiveMessages, this, 0, NULL);
104 }
```

在 OnBnClickedButtonConnect 函数中，客户端首先初始化 Winsock 库以准备网络通讯，并从界面获取服务器的 IP 地址和端口号。之后，创建一个 TCP 套接字并尝试连接到服务器。连接成功后，客户端接收服务器发送的 RSA 公钥，用于安全传输 DES 密钥。客户端生成一个 DES 密钥，用接收到的 RSA 公钥加密此密钥，并将加密后的 DES 密钥发送给服务器，确保只有服务器能够解密并使用它进行后续的加密通信。

此过程中，客户端还将自己的用户名使用新生成的 DES 密钥进行加密，然后发送给服务器，完成身份声明。一旦这些步骤完成，客户端禁用界面上的连接配置控件，防止在已建立的连接中更改设置。最后，客户端在新线程中持续监听服务器发送的消息，这些消息通过 DES 密钥解密后显示在聊天界面上。这个流程确保了客户端与服务器之间的通信安全，有效防止了密钥和数据在传输过程中被窃听或篡改。

3.5 单元测试

为了确保 DES 和 RSA 的正确性，我编写了一些单元测试用例，对 DES 和 RSA 的加密解密功能进行了测试。测试代码如下：

```cpp
1   namespace test {
2     TEST_CLASS(test) {
3   public:
4     // 测试 DES 加密解密
5     TEST_METHOD(TestDES) {
6       DES des("133457799BBCDFF1");
7       std::string plaintextStr = "The homework is awesome!";
8       Logger::WriteMessage(("Plaintext: " + plaintextStr + "\n").c_str());
9       // 转换明文字符串为 vector<uint8_t>
10                    std::vector<uint8_t>     plaintext(plaintextStr.begin(),
    plaintextStr.end());
11      // 加密
12      auto ciphertext = des.encrypt(plaintext);
13      // 将密文转换为十六进制字符串以便输出
14      std::stringstream hexstream;
```

```
15      for (unsigned char c : ciphertext) {
16          hexstream << std::hex << std::setfill('0') << std::setw(2) << static_cast<int>(c) << " ";
17      }
18       Logger::WriteMessage(("Ciphertext (as hex): " + hexstream.str() + "\n").c_str());
19      // 解密
20      auto decryptedVector = des.decrypt(ciphertext);
21      // 将解密后的 vector<uint8_t>转换回 std::string
22      std::string decText(decryptedVector.begin(), decryptedVector.end());
23      Logger::WriteMessage(("Decrypted Text: " + decText + "\n").c_str());
24     Assert::AreEqual(plaintextStr, decText, L"Decrypted text does not match the original plaintext.");
25    }
26    // 测试 RSA 加密解密 接口 1
27    TEST_METHOD(TestRSA_UINT64) {
28      PublicKey pubkey;
29      PrivateKey prikey;
30      RSA::generateKeys(pubkey, prikey);
31      Logger::WriteMessage(("Public Key: (" + std::to_string(pubkey.n) + ", " + std::to_string(pubkey.e) + ")" + "\n").c_str());
32     Logger::WriteMessage(("Private Key: (" + std::to_string(prikey.n) + ", " + std::to_string(prikey.d) + ")" + "\n").c_str());
33      uint64_t text = 123456789;
34          Logger::WriteMessage(("Plaintext:  " + std::to_string(text) + "\n").c_str());
35      uint64_t ciphertext = RSA::encrypt(text, pubkey);
36       Logger::WriteMessage(("Ciphertext: " + std::to_string(ciphertext) + "\n").c_str());
37      uint64_t decrypttext = RSA::decrypt(ciphertext, prikey);
38       Logger::WriteMessage(("Decrypted: " + std::to_string(decrypttext) + "\n").c_str());
39       Assert::AreEqual(text, decrypttext, L"Decrypted text does not match the original plaintext.");
40    }
41    // 测试 RSA 加密解密 接口 2
42    TEST_METHOD(TestRSA_VEC) {
43      PublicKey pubkey;
44      PrivateKey prikey;
45      RSA::generateKeys(pubkey, prikey);
46      std::stringstream hexstream;
47      Logger::WriteMessage(("Public Key: (" + std::to_string(pubkey.n) + ", " + std::to_string(pubkey.e) + ")" + "\n").c_str());
```

```
48      Logger::WriteMessage(("Private Key: (" + std::to_string(prikey.n) + ", " + std::to_string(prikey.d) + ")" + "\n").c_str());
49      std::string text = "133457799BBCDFF1";
50      Logger::WriteMessage(("Plaintext: " + text + "\n").c_str());
51
52      std::vector<uint8_t> plaintext = RSA::strToVec(text);
53      for (unsigned char c : plaintext) {
54          hexstream << std::hex << std::setfill('0') << std::setw(2) << static_cast<int>(c) << " ";
55      }
56        Logger::WriteMessage(("Plaintext (as hex): " + hexstream.str() + "\n").c_str());
57
58      std::vector<uint8_t> ciphertext = RSA::encrypt(plaintext, pubkey);
59      hexstream.str("");
60      for (unsigned char c : ciphertext) {
61          hexstream << std::hex << std::setfill('0') << std::setw(2) << static_cast<int>(c) << " ";
62      }
63        Logger::WriteMessage(("Ciphertext (as hex): " + hexstream.str() + "\n").c_str());
64
65      std::vector<uint8_t> decrypted = RSA::decrypt(ciphertext, prikey);
66      hexstream.str("");
67      for (unsigned char c : decrypted) {
68          hexstream << std::hex << std::setfill('0') << std::setw(2) << static_cast<int>(c) << " ";
69      }
70        Logger::WriteMessage(("Decrypted (as hex): " + hexstream.str() + "\n").c_str());
71      std::string dtext = RSA::vecToStr(decrypted);
72      Logger::WriteMessage(("Decrypted: " + dtext + "\n").c_str());
73        Assert::AreEqual(text, dtext, L"Decrypted text does not match the original plaintext.");
74    }
75    };
76  }
```

在这段单元测试代码中，通过对 DES 和 RSA 加密算法的功能性测试，验证了加密和解密流程的正确性和可靠性。对于 DES 测试，使用了静态密钥和明文字符串进行加密，并验证解密后的文本与原始明文是否一致。在 RSA 测试中，首先生成公钥和私钥，然后对一个整数和一个字符串进行加密和解密操作，确保解密后的数据与原始数据相同。通过输出加密和解密过程中的中间值，单元测试不仅证明了算法的正确实现，还提供了对

加密过程的深入理解和可视化。这些测试结果强调了实现的加密系统的有效性，显示了其在实际应用中的可行性和安全性。

3.6 运行示例

首先运行 ChatRoomServer.exe ， 显示端口以及 DES 密钥(ip 为 127.0.0.1)：

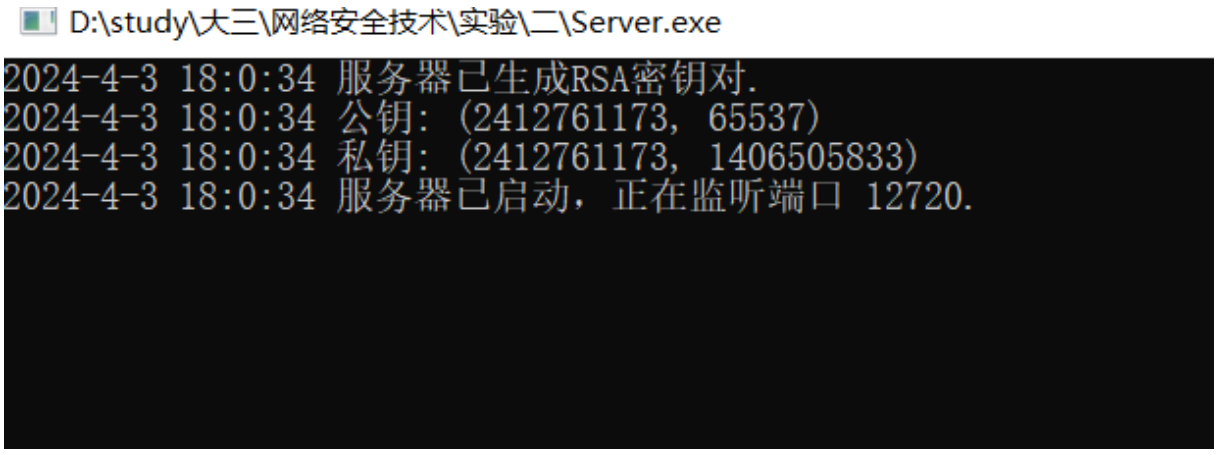


图 3.6.9: 启动服务器

启动 ChatRoomClient.exe ，输入任意用户名，点击 连接服务器，即可**接收 RSA 公钥，同时生成 DES 密钥并加密发送**：

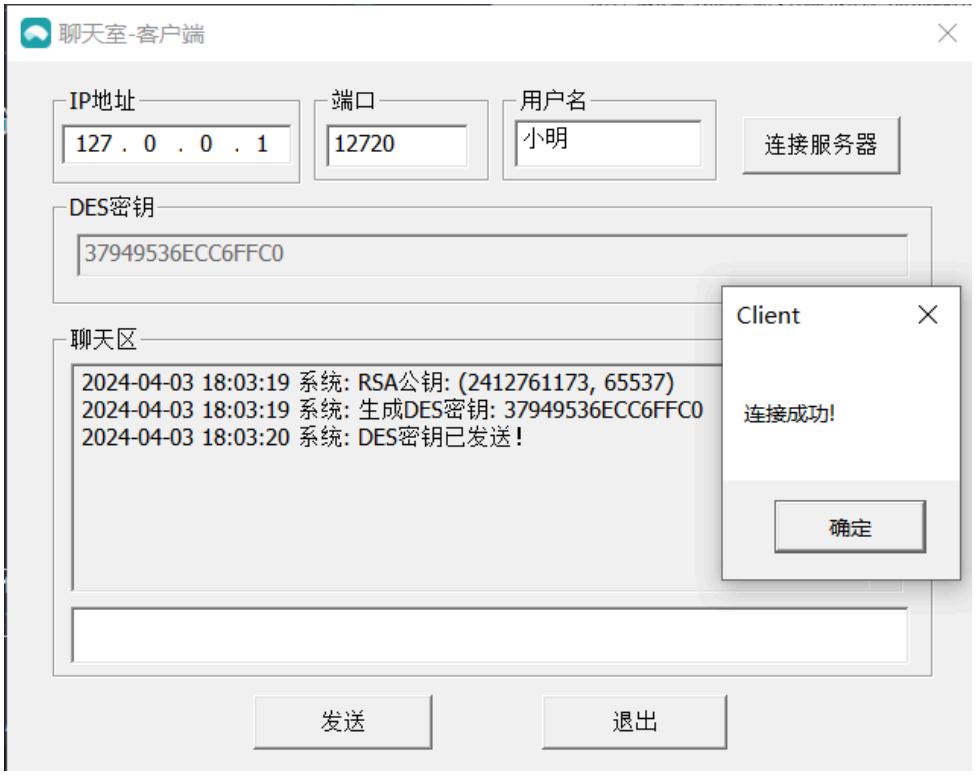


图 3.6.10: 启动客户端

此时客户端可发送任意消息，同时**支持多人聊天**，打开另一个客户端，进行连接后，两个人(或更多)接下来可以任意进行实时聊天了：

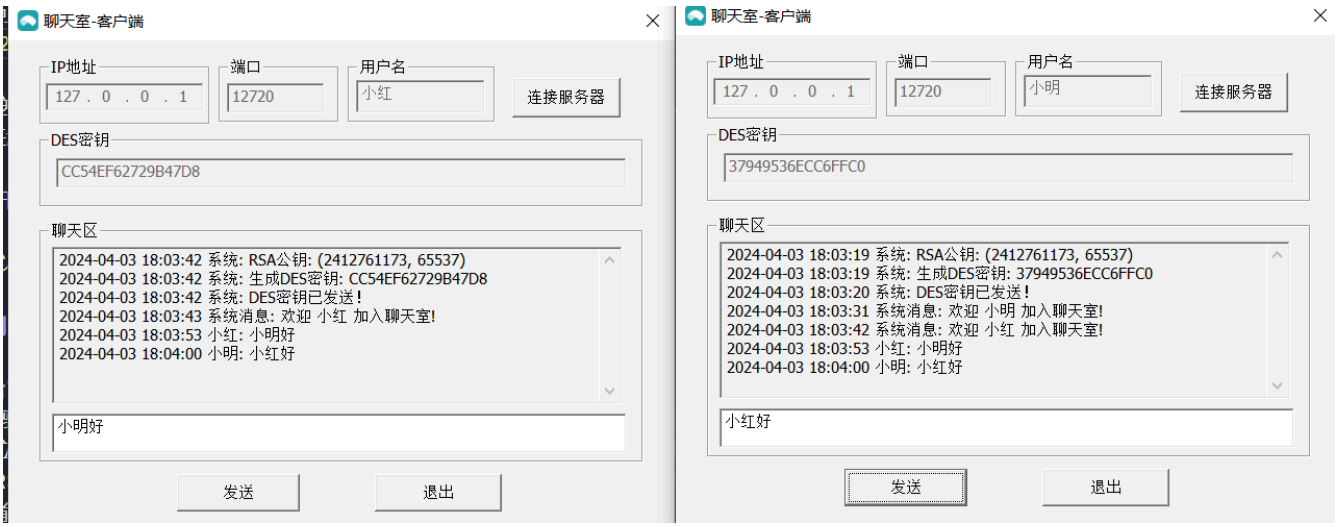


图 3.6.11: 聊天

途中若有一人离开聊天，服务器将对其他所有人进行提示：

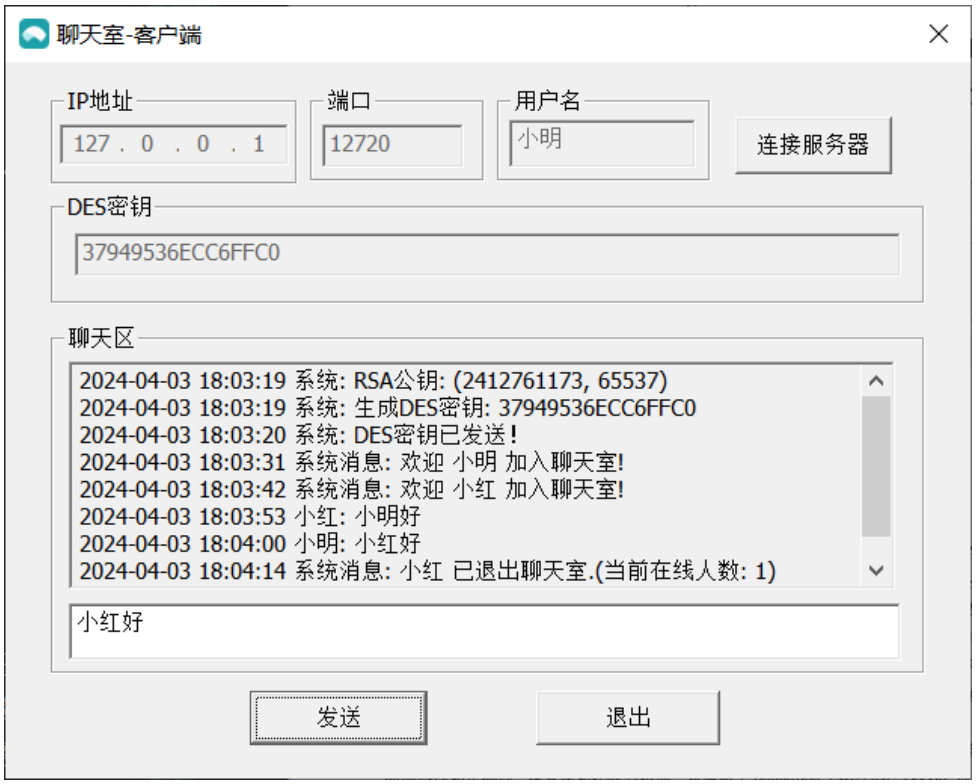


图 3.6.12: 离开

另外，服务器可以通过输入 count 来查询当前在线的客户数量：

图 3.6.13: 查看在线人数

4 实验遇到的问题及其解决方法

4.1 RSA 接口配合 TCP 通信问题

在实验过程中，我发现 RSA 加密解密的接口与 TCP 通信的配合存在一些问题。在客户端与服务器之间建立连接后，客户端需要将自己的 DES 密钥使用服务器的 RSA 公钥加密后发送给服务器。然而，由于 RSA 加密的数据是一个大整数，而 TCP 通信只能发送字节流，这就需要将大整数转换为字节流进行传输。在接收端，服务器需要将接收到的字节流转换为大整数，然后使用私钥解密得到 DES 密钥。这个过程中，数据的转换和传输需要保证数据的完整性和正确性，否则会导致加密密钥的错误，从而影响后续的通信安全。

因此，我在实验中使用了 std::vector 来存储加密后的 DES 密钥，这样可以确保数据的完整性和正确性。在发送和接收数据时，我使用了 reinterpret_cast<char*>将 std::vector 转换为 char*，这样可以将数据转换为字节流进行传输。在接收端，我也使用 reinterpret_cast<uint8_t*>将接收到的数据转换为 std::vector，这样可以确保数据的完整性和正确性。这种方法可以有效解决RSA接口与TCP通信的配合问题，确保了加密密钥的正确传输和使用。 对比如下：

- **原接口：**

```cpp
uint64_t RSA::encrypt(uint64_t plaintext, PublicKey pubkey);

uint64_t RSA::decrypt(uint64_t ciphertext, PrivateKey prikey);
```

- **现接口：**

```cpp
static std::vector<uint8_t> encrypt(const std::vector<uint8_t>& plaintext, PublicKey pubkey);
static std::vector<uint8_t> decrypt(const std::vector<uint8_t>& ciphertext, PrivateKey prikey);
```

4.2 伪随机数发生器问题

一开始，我使用 rand()函数生成伪随机数作为 DES 密钥，但后来发现这种方法并不安全，因为 rand()函数生成的随机数并不是真正的随机数，而是伪随机数。这种伪随机数生成方法容易被破解，从而导致密钥的泄露和通信的不安全。

因此，我改用 C++标准库中的 std::random_device 和 std::uniform_int_distribution 来生成真正的随机数。std::random_device 是一个真正的随机数生成器，它使用硬件和操作系统的随机源来生成随机数，因此生成的随机数更加安全和随机。std::uniform_int_distribution 是一个均匀分布的随机数分布器，它可以生成指定范围内的随机数。通过这种方法，我可以生成更加安全和随机的 DES 密钥，确保通信的安全性和可靠性。

```cpp
uint64_t RSA::genPrime(int bits) {
    static std::random_device rd;
    static std::mt19937_64 gen(rd());
    uint64_t hbit = 1ULL << (bits - 1);
    uint64_t lbit = 1ULL << bits;
    std::uniform_int_distribution<uint64_t> dis(hbit, lbit - 1);
    uint64_t n = 0;
    do {
      n = dis(gen);
    } while (!RSA::millerRabin(n, 100));
    return n;
}
```

4.3 DEBUG 问题

在实验过程中，我发现在调试过程中，由于加密和解密的数据是二进制数据，直接输出到控制台并不直观，很难看出加密和解密的效果。这给调试和测试带来了一定的困难。因此，在单元测试的代码中，我是这样输出的：

```cpp
std::vector<uint8_t> plaintext = RSA::strToVec(text);
for (unsigned char c : plaintext) {
    hexstream << std::hex << std::setfill('0') << std::setw(2) << static_cast<int>(c) << " ";
}
Logger::WriteMessage(("Plaintext (as hex): " + hexstream.str() + "\n").c_str());
```

测试显示

测试以十六进制形式输出明文和密文，便于查看加密和解密效果。

5 实验结论

通过本次实验，成功实现了基于 TCP 协议的聊天室应用，其中集成了 DES 和 RSA 加密算法，保障了通信过程的安全性。在实验过程中，深入理解了对称加密（DES）和非对称加密（RSA）的工作原理及其在实际网络通信中的应用。单元测试验证了加密和解密功能的正确性，确保了加密系统的可靠性。此外，实验强化了在网络编程和安全领域的知识，特别是在实现安全密钥交换和加密通信方面的实践经验。这些技能和知识对于构建安全的网络应用至关重要，也为日后在信息安全领域的深入研究和工作奠定了坚实的基础。