



南开大学  
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

---

## 实验一：基于 DES 加密的 TCP 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 4 月 13 日

# 目录

<b>1 实验目标</b>	<b>2</b>
<b>2 github 仓库</b>	<b>2</b>
<b>3 实验内容</b>	<b>2</b>
<b>4 实验原理</b>	<b>2</b>
4.1 DES 加密 . . . . .	2
4.2 TCP 通信 . . . . .	4
<b>5 实验步骤</b>	<b>5</b>
5.1 实验环境 . . . . .	5
5.2 DES 加解密算法实现 . . . . .	6
5.2.1 DES_Operation.h . . . . .	7
5.2.2 DES_Operation.cpp . . . . .	9
5.3 基于 TCP 协议的聊天室实现 . . . . .	13
5.3.1 chat.h . . . . .	13
5.3.2 chat.cpp . . . . .	14
<b>6 实验遇到的问题及其解决方法</b>	<b>20</b>
6.1 在 linux 上编译 CPP 项目 . . . . .	20
6.2 DES 相关算法实现 . . . . .	20
6.3 .gitignore 的使用 . . . . .	20
<b>7 实验结论</b>	<b>20</b>
7.1 利用 cmake 编译项目 . . . . .	20
7.2 新建服务器端与客户端 . . . . .	21
7.3 功能测试 . . . . .	22
7.3.1 英文输入测试 . . . . .	22
7.3.2 中文输入测试 . . . . .	22
7.3.3 断开连接测试 . . . . .	22
<b>8 代码结构说明</b>	<b>23</b>
<b>9 参考文献</b>	<b>23</b>

## 1 实验目标

DES (Data Encryption Standard) 算法是一种用 56 位有效密钥来加密 64 位数据的对称分组加密算法, 该算法流程清晰, 已经得到了广泛的应用, 算是应用密码学中较为基础的加密算法。TCP (传输控制协议) 是一种面向链接的、可靠的传输层协议。TCP 协议在网络层 IP 协议的基础上, 向应用层用户进程提供可靠的、全双工的数据流传输。

本次实验, 我们需要在 Linux 平台下, 实现基于 DES 加密的 TCP 通信, 具体要求如下。

- 能够在了解 DES 算法原理的基础上, 编程实现对字符串的 DES 加密解密操作。
- 能够在了解 TCP 和 Linux 平台下的 Socket 运行原理的基础上, 编程实现简单的 TCP 通信, 为简化编程细节, 不要求实现一对多通讯。
- 将上述两部分结合到一起, 编程实现通信内容事先通过 DES 加密的 TCP 聊天程序, 要求双方事先互通密钥, 在发送方通过该密钥加密, 然后由接收方解密, 保证在网络上传输的信息的保密性。

## 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

1. 实现 DES 加解密算法;
2. 实现基于 TCP 协议的一个简易聊天室;
3. 将二者结合, 聊天室发送的内容需经过 DES 加密。

## 4 实验原理

### 4.1 DES 加密

在进行本次实验之前, 我首先阅读了本次实验所提供的**参考资料**, 重新复习了一遍上学期《密码学》中学习的 DES 加密相关的知识。由于在参考资料中, 已经详细介绍了有关 DES 加密的流程, 所以此处我们仅仅做一下概括。

DES 加密的步骤一般如下所示:

1. **初始置换  $IP$** : 64 bit 的明文重新排列, 而后分成左右两块, 每块 32bit, 用  $L_0$  和  $R_0$  表示。 $IP$  置换表是固定的。通过对该表进行观察可以发现其中相邻两列的元素位置号数相差 8, 前 32 个元素均为偶数号码, 后 32 个均为奇数号码, 这样的置换相当于将明文的各字节按列写出, 各列经过偶采样和奇采样置换后, 再对其进行逆序排列, 将阵中元素按行读出以便构成置换的输出。
2. **逆初始置换  $IP^{-1}$** : 在 16 圈迭代之后, 将左右两端合并为 64bit, 进行逆初始置换  $IP^{-1}$ , 得到输出的 64bit 密文。

3. **16 轮迭代**: 16 轮迭代是 DES 算法的核心部分。将经过 IP 置换后的数据分成 32bit 的左右两段, 进行 16 圈迭代, 每轮迭代只对右边的 32bit 进行一系列的加密变换, 在一轮加密变换结束时, 将左边的 32bit 与右边进行异或后得到的 32bit, 作为下一轮迭代时右边的段, 并将这轮迭代中的右边段未经任何加密变换时的初始值直接作为下一轮迭代时左边的段, 这需要在每轮迭代开始时, 先将右边段保存一个副本, 以便在该轮迭代结束时, 将该副本直接赋值给下一轮迭代的左边段。在每轮迭代时, 右边的数据段要经过的加密运算包括选择扩展运算 E、密钥加运算、选择压缩运算 S 和置换 P, 这些变换合称 f 函数。

- **选择扩展运算**: 将输入的右边 32bit 扩展成为 48bit 输出, 置换结果按行输出的结果即为密钥加运算 48bit 的输入。
  - **密钥加运算**: 将选择扩展运算输出的 48bit 作为输入, 与 48bit 的子密钥进行异或运算, 异或结果作为选择压缩运算 (S 盒) 的输入。
  - **选择压缩运算**: 选择压缩运算 (S 盒) 是 DES 算法中唯一的非线性部分, 它是一个查表运算, 共有 8 张非线性的变换表, 每张表的输入为 6bit, 输出为 4bit。在查表之前, 将密钥加运算的输出作为 48bit 输入, 将其分为 8 组, 每组 6bit, 分别进入 8 个 S 盒进行运算, 得出 32bit 的输出结果作为置换运算的输入。
  - **置换运算**: 是一个 32bit 的换位运算, 对选择压缩运算输出的 32bit 数据按表进行换位。将数据  $R_1 R_2 R_3 \dots R_{31} R_{32}$  转换成为  $R_{16} R_7 R_{20} \dots R_4 R_{25}$ 。至此, 最终获得的 32bit 数据, 即为此轮迭代的输出。此输出与左边的 32bit 进行异或作为下一轮的右边段, 进行加密运算前的原始的右边段作为下一轮的左边段。
4. **子密钥生成**: 64bit 初始密钥经过置换选择 PC-1、循环左移运算 LS、置换选择 PC-2, 产生 16 轮迭代所用到的子密钥  $k_i$ 。
- **置换选择 PC-1**: 置换选择 PC-1 只在第一轮子密钥的产生过程中需要使用, 它的目的是从 64bit 初始密钥中选出 56bit 有效位。
  - **循环左移 LS**: 此轮密钥的产生所需要循环左移的位数即为表中的第 N 个元素 (首元素为第一个)。C, D 寄存器中的 28bit 经过循环左移后, 拼接为 56bit 作为此轮置换选择 PC-2 的输入, 同时也作为第 N+1 轮子密钥循环左移的输入。
  - **置换选择 PC-2**: 置换选择 PC-2 将输入的 56bit 中的第 9、18、22、25、35、38、43、54 位删去, 将其余位置按照表 2-16 置换位置, 输出 48bit, 作为第 N 轮的子密钥。

以上就是 DES 加密的主要流程, 我们展示一下对应的 DES 加密流程图以及 DES 密钥生成流程图:

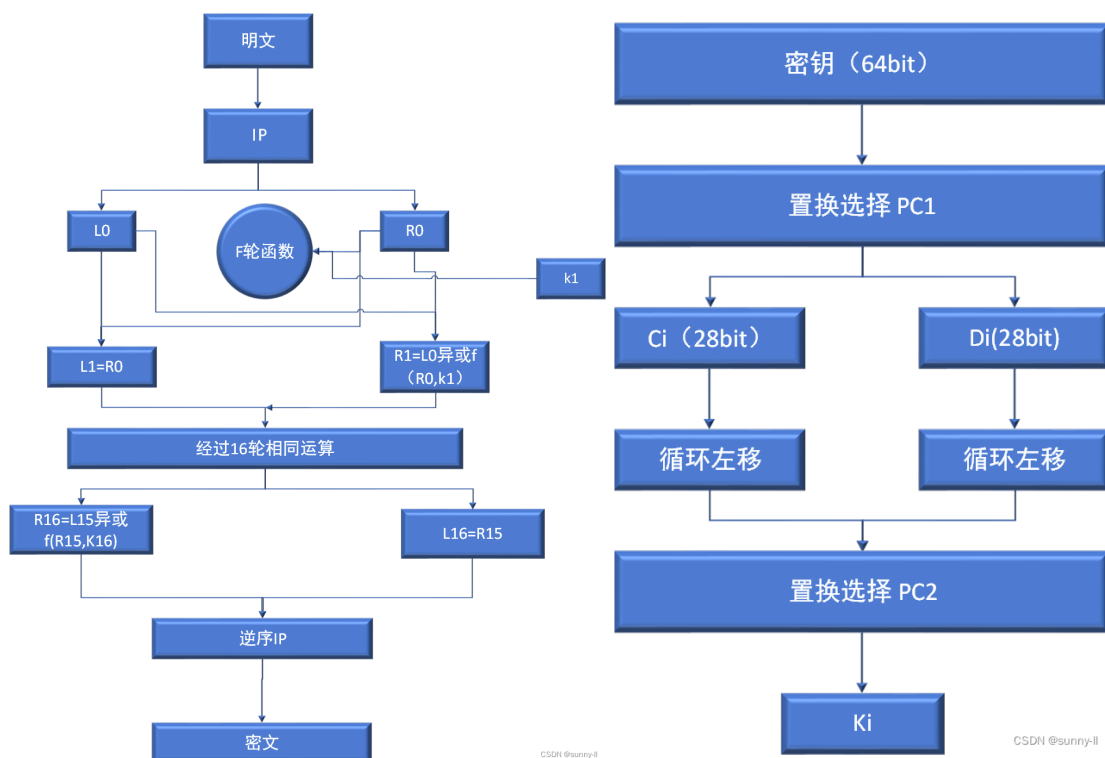


图 4.1: DES 加密流程图以及 DES 密钥生成流程图

## 4.2 TCP 通信

同样，在上学期的《计算机网络》中，我们学习到了 TCP 相关的通信协议，了解其数据包的格式，知道各有效位的含义，因此此处不再详细阐述，仅仅回顾一下 TCP 数据包的组成。

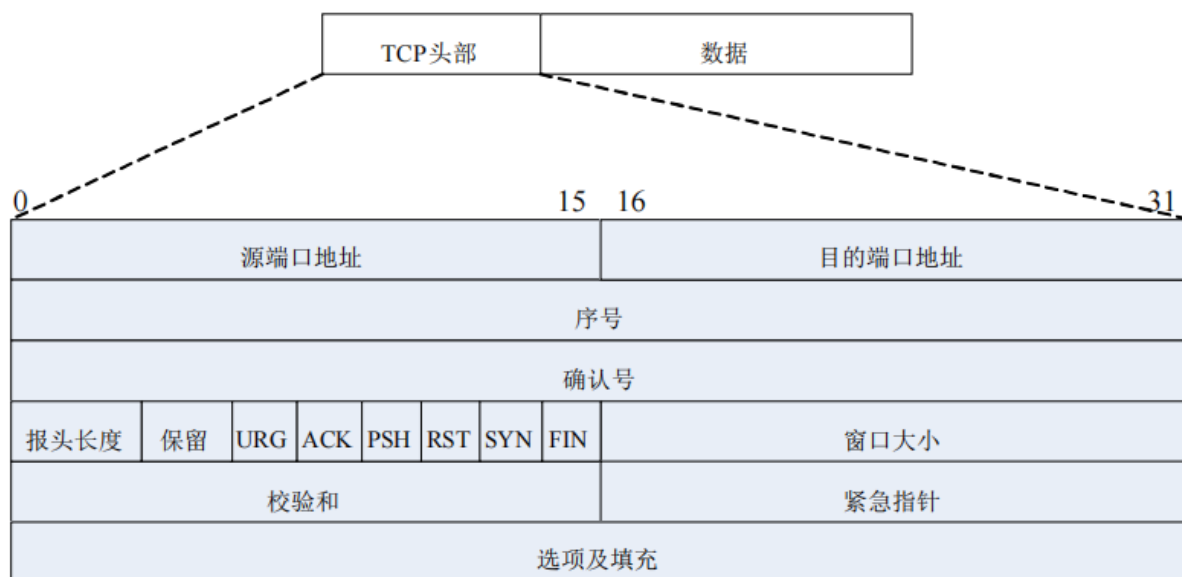


图 4.2: TCP 主要格式

在 TCP 协议中，存在很多的标志位。简单介绍一下各标志位的含义：

- **URG**：紧急字段指针。如果为 1，表示数据包中包含紧急数据。此时 TCP 协议包头结构中的紧急指针有效。
- **ACK**：确认标志位。如果为 1，表示包中的确认号是有效的。否则，包中的确认号是无效的。
- **PSH**：推送功能。如果为 1，表示接收端应尽快将数据传送给应用层。
- **RST**：重置位，用来复位一个连接。RST 标志置位的数据包称为复位包。一般情况下，如果 TCP 收到的一个分段明显不是属于该主机上的任何一个连接，则向远端发送一个复位包。
- **SYN**：用来建立连接，让连接双方同步序列号。如果 SYN=1 而 ACK=0，则表示数据包为连接请求；如果 SYN=1 而 ACK=1，则表示接受连接。
- **FIN**：表示发送端已经没有数据要求传输了，请求释放连接。

为了实现 TCP 协议的编程，我们必须使用系统封装好的一些函数来进行编写。简单介绍一些常见的函数：

- **socket 函数**：该函数用于创建通信的套接字，并返回该套接字的文件描述符。
- **bind 函数**：该函数用于将套接字与指定端口相连。
- **listen 函数**：该函数用于实现服务器等待客户端请求的功能。
- **accept 函数**：该函数用于处于监听状态的服务器，在获得客户机连接请求后，会将其放置在等待队列中，当系统空闲时，服务器用该函数接受客户机连接请求。
- **connect 函数**：该函数用于客户端向服务器发出连接请求。
- **write 函数**：该函数用于服务器和客户端建立连接后，将 buf 中的 nbytes 字节的内容写入文件描述符。
- **read 函数**：该函数用于从文件描述符 fd 中读取内容。
- **send 函数**：该函数的作用基本同 write 函数相同，用于将信息发送到指定的套接字文件描述符中，其功能比 write 函数更为全面。
- **recv 函数**：该函数的作用基本同 read 函数相同，用于从指定的套接字中获取信息。
- **close 函数**：该函数用于关闭套接字。

介绍完 TCP 的主要原理，接下来我们就可以开始实验啦。

## 5 实验步骤

### 5.1 实验环境

本次实验按照作业要求，选用 linux 环境进行编写代码。由于 linux 环境下编译 Cpp 文件需要使用命令行，而本次实验中涉及到多个 Cpp 文件，所以我们选用 cmake 工具来进行编译，生成的可执行文件存放在 bin 文件夹中，然后进行测试即可。

本次实验我们在 linux 服务器上进行搭建，以下是服务器的一些基本信息：

```
root@tomorin:/home/Network_Security_Technology/Lab01# nvidia-smi
Sun Apr 13 05:41:07 2025
```

NVIDIA-SMI 550.135			Driver Version: 550.135			CUDA Version: 12.4		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.	
=====								
0	NVIDIA A100-PCIE-40GB	Off	00000000:1A:00.0	Off		0		
N/A	75C	P0	242W / 250W	38119MiB / 40960MiB	96%	Default	Disabled	
-----								
1	NVIDIA A100-PCIE-40GB	Off	00000000:1B:00.0	Off		0		
N/A	76C	P0	253W / 250W	38101MiB / 40960MiB	95%	Default	Disabled	
-----								
2	NVIDIA A100-PCIE-40GB	Off	00000000:88:00.0	Off		0		
N/A	31C	P0	33W / 250W	1MiB / 40960MiB	0%	Default	Disabled	
-----								
3	NVIDIA A100-PCIE-40GB	Off	00000000:89:00.0	Off		189		
N/A	36C	P0	ERR! / 250W	1MiB / 40960MiB	0%	Default	Disabled	
-----								

图 5.3: 服务器基本信息

我们选用 ubuntu20.04 版本，显卡使用 A100(40GB) 来进行实验。

首先，我们需要进行 cmake 工具的安装，我们使用以下命令行进行安装即可：

```
1 sudo apt update
2 sudo apt install cmake
```

执行上述步骤后，得到以下结果，我们就完成了 cmake 工具的安装！

```
root@tomorin:/home/Network_Security_Technology/Lab01# sudo apt update
Hit:1 https://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy InRelease
Hit:2 https://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-updates InRelease
Hit:3 https://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-backports InRelease
Hit:4 https://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-security InRelease
Hit:5 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64 InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
81 packages can be upgraded. Run 'apt list --upgradable' to see them.
root@tomorin:/home/Network_Security_Technology/Lab01# sudo apt install cmake
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
cmake is already the newest version (3.22.1-1ubuntu1.22.04.2).
0 upgraded, 0 newly installed, 0 to remove and 81 not upgraded.
root@tomorin:/home/Network_Security_Technology/Lab01#
```

图 5.4: 安装 cmake 工具

下面我们就可以开始本次实验了。

## 5.2 DES 加解密算法实现

本次实验中，我们将头文件写在了 include 文件夹中，将主文件写在了 src 文件夹中，统一格式，方便我们进行 cmake 的编译。

下面我们对编写的代码进行分析。

### 5.2.1 DES\_Operation.h

我们构建了一个有关于 DES 操作的头文件，代码如下所示：

```
1  class DesOp {
2  private:
3      uint8_t key[8] = {0};
4      uint8_t subKeys[16][6] = {0};
5
6      static const uint8_t IP[64];
7      static const uint8_t IP_INV[64];
8      static const uint8_t E[48];
9      static const uint8_t S[8][4][16];
10     static const uint8_t P[32];
11     static const uint8_t PC1[2][28];
12     static const uint8_t LS[16];
13     static const uint8_t PC2[48];
14
15     void GenerateSubKeys();
16     void F(uint8_t* R, uint8_t* subKey, uint8_t* result);
17     void DES(uint8_t* plainText_byte, uint8_t* cipherText_byte, bool isEncrypt);
18
19     static void Xor(uint8_t* a, uint8_t* b, int length);
20     static void Copy(uint8_t* a, uint8_t* b, int length);
21     static void ByteToBit(uint8_t* byte, uint8_t* bit, int length);
22     static void BitToByte(uint8_t* bit, uint8_t* byte, int length);
23
24 public:
25     DesOp();
26     ~DesOp() = default;
27     void SetKey(const char* key);
28     void Encrypt(char* plainText, int plainTextLength, char*& cipherText, int&
        ↪ cipherTextLength);
29     void Decrypt(char* cipherText, int cipherTextLength, char*& plainText, int&
        ↪ plainTextLength);
30 };
```

我们构造了一个 DesOp 的类，在里面定义了一些实现 DES 加解密的函数，具体代码解释如下所示。



## DES\_Operation.h 代码解释

- **key[8]**: 这是一个长度为 8 字节 (64 位) 的数组, 用于存储 DES 算法所使用的密钥。
- **subKeys[16][6]**: 这是一个二维数组, 用于存储 16 轮子密钥, 每个子密钥长度为 6 字节 (48 位)。
- **IP[64]**: 初始置换表, 用于在加密和解密开始时对明文或密文进行初始置换。
- **IP\_INV[64]**: 逆初始置换表, 用于在加密和解密结束时对数据进行逆初始置换。
- **E[48]**: 扩展置换表, 用于将 32 位的右半部分数据扩展为 48 位。
- **S[8][4][16]**: S 盒, 是 DES 算法中的非线性部分, 将 48 位的数据压缩为 32 位。
- **P[32]**: P 盒置换表, 用于对 S 盒输出的 32 位数据进行置换。
- **PC1[2][28]**: 密钥初始置换表, 用于对 64 位的密钥进行置换得到 56 位的密钥。
- **LS[16]**: 左移表, 规定了在生成子密钥过程中每一轮密钥的左移位数。
- **PC2[48]**: 密钥压缩置换表, 用于将 56 位的密钥压缩为 48 位的子密钥。
- **GenerateSubKeys()**: 用于生成 16 轮子密钥, 它会根据初始密钥和 PC1、LS、PC2 等置换表生成 16 个 48 位的子密钥。
- **F(uint8\_t\* R, uint8\_t\* subKey, uint8\_t\* result)**: 这是 DES 算法中的 F 函数, 它接收 32 位的右半部分数据 R 和 48 位的子密钥 subKey, 经过扩展置换、异或、S 盒替换和 P 盒置换等操作后, 将结果存储在 result 中。
- **DES(uint8\_t\* plainText\_byte, uint8\_t\* cipherText\_byte, bool isEncrypt)**: 实现了 DES 算法的核心加密和解密逻辑, 根据 isEncrypt 参数的值决定是进行加密还是解密操作。
- **Xor(uint8\_t\* a, uint8\_t\* b, int length)**: 对两个字节数组进行按位异或操作。
- **Copy(uint8\_t\* a, uint8\_t\* b, int length)**: 将一个字节数组的内容复制到另一个字节数组中。
- **ByteToBit(uint8\_t\* byte, uint8\_t\* bit, int length)**: 将字节数组转换为位数组。
- **BitToByte(uint8\_t\* bit, uint8\_t\* byte, int length)**: 将位数组转换为字节数组。
- **SetKey(const char\* key)**: 用于设置 DES 算法的密钥, 设置完密钥后会调用 GenerateSubKeys() 方法生成子密钥。
- **Encrypt(char\* plainText, int plainTextLength, char\*& cipherText, int& cipherTextLength)**: 对明文进行加密操作, 接收明文和明文长度作为输入, 输出加密后的密文和密文长度。
- **Decrypt(char\* cipherText, int cipherTextLength, char\*& plainText, int& plainTextLength)**: 对密文进行解密操作, 接收密文和密文长度作为输入, 输出解密后的明文和明文长度。

### 5.2.2 DES\_Operation.cpp

下面，我们分别对各函数进行分析。

**DES 各部分数据** 此处不再过多介绍，基本上按照参考资料中的代码进行编写，我们分别定义了各部分数据，代码不再贴上来，详见我的代码。

**简单功能函数** 此处也不再过多介绍，我们定义了 Xor 函数、Copy 函数、ByteToBit 函数、BitToByte 函数来辅助我们完成 DES 加解密的设计。具体代码过于简单，不予展示。

#### SetKey 函数

```
1 void DesOp::SetKey(const char* key) {
2     for (int i = 0; i < 8; i++) {
3         this->key[i] = key[i];
4     }
5     GenerateSubKeys();
6 }
```

#### SetKey 函数解释

将输入的字符数组 key 复制到类的成员变量 key 中，并调用 GenerateSubKeys 函数生成子密钥。

#### GenerateSubKeys 函数

```
1 void DesOp::GenerateSubKeys() {
2     uint8_t key_bit[64];
3     ByteToBit(key, key_bit, 8);
4     uint8_t key56L[28], key56R[28];
5     for (int i = 0; i < 28; i++) {
6         key56L[i] = key_bit[PC1[0][i] - 1];
7         key56R[i] = key_bit[PC1[1][i] - 1];
8     }
9
10    uint8_t subKey[48];
11    for (int i = 0; i < 16; i++) {
12        for (int j = 0; j < LS[i]; j++) {
13            uint8_t tempL = key56L[0], tempR = key56R[0];
14            for (int k = 0; k < 27; k++) {
15                key56L[k] = key56L[k + 1];
16                key56R[k] = key56R[k + 1];
17            }
18            key56L[27] = tempL;
```

```
19         key56R[27] = tempR;
20     }
21
22     int index;
23     for (int j = 0; j < 48; j++) {
24         index = PC2[j] - 1;
25         if (index < 28) {
26             subKey[j] = key56L[index];
27         } else {
28             subKey[j] = key56R[index - 28];
29         }
30     }
31     BitToByte(subKey, subKeys[i], 6);
32 }
33 }
```

#### GenerateSubKeys 函数解释

这个函数生成 16 个子密钥。首先将密钥转换为位数组，然后根据 PC1 表将其分为左右两部分。接着对每一轮，根据 LS 表对左右两部分进行左移操作，再根据 PC2 表从左右两部分中选取 48 位组成子密钥，最后将子密钥转换为字节数组并存储到 subKeys 中。

#### F 函数

```
1 void DesOp::F(uint8_t* R, uint8_t* subKey, uint8_t* result) {
2     uint8_t R_exp[48];
3     for (int i = 0; i < 48; i++) {
4         R_exp[i] = R[E[i] - 1];
5     }
6
7     uint8_t subKey_bit[48];
8     ByteToBit(subKey, subKey_bit, 6);
9     Xor(R_exp, subKey_bit, 48);
10
11     uint8_t S_out[32];
12     for (int i = 0; i < 8; i++) {
13         int row = R_exp[i * 6] * 2 + R_exp[i * 6 + 5];
14         int col = (R_exp[i * 6 + 1] << 3) + (R_exp[i * 6 + 2] << 2) + (R_exp[i * 6
15         ↵ + 3] << 1) + R_exp[i * 6 + 4];
16         uint8_t val = S[i][row][col];
17         for (int j = 0; j < 4; j++) {
18             S_out[i * 4 + j] = (val >> (3 - j)) & 0x01;
19         }
20     }
21 }
```

```
19     }
20
21     for (int i = 0; i < 32; i++) {
22         result[i] = S_out[P[i] - 1];
23     }
24 }
```

### F 函数解释

首先将输入的 32 位数据 R 通过扩展置换表 E 扩展为 48 位，然后与子密钥进行异或运算。接着将异或结果通过 S 盒进行替换，得到 32 位的结果，最后通过 P 盒进行置换并将结果存储到 result 中。

### DES 函数

```
1 void DesOp::DES(uint8_t* plainText_byte, uint8_t* cipherText_byte, bool isEncrypt)
   ↪ {
2     uint8_t plainText[64], cipherText[64];
3     ByteToBit(plainText_byte, plainText, 8);
4     ByteToBit(cipherText_byte, cipherText, 8);
5
6     uint8_t plainText_bit[64];
7     for (int i = 0; i < 64; i++) {
8         plainText_bit[i] = plainText[IP[i] - 1];
9     }
10
11     uint8_t L[32], R[32];
12     for (int i = 0; i < 32; i++) {
13         L[i] = plainText_bit[i];
14         R[i] = plainText_bit[i + 32];
15     }
16
17     uint8_t temp[32];
18     for (int i = 0; i < 16; i++) {
19         Copy(temp, R, 32);
20         F(R, subKeys[isEncrypt ? i : (15 - i)], R);
21         Xor(R, L, 32);
22         Copy(L, temp, 32);
23     }
24
25     for (int i = 0; i < 32; i++) {
26         plainText_bit[i] = R[i];
27         plainText_bit[i + 32] = L[i];
```

```
28     }
29
30     for (int i = 0; i < 64; i++) {
31         cipherText[i] = plainText_bit[IP_INV[i] - 1];
32     }
33
34     BitToByte(plainText, plainText_byte, 8);
35     BitToByte(cipherText, cipherText_byte, 8);
36 }
```

### DES 函数解释

首先将输入的字节数组转换为位数组，并进行初始置换。然后将数据分为左右两部分，进行 16 轮迭代，每轮中调用 F 函数并进行左右部分的交换和异或操作。最后进行逆初始置换，并将结果转换回字节数组。

### Encrypt 函数

```
1 void DesOp::Encrypt(char* plainText, int plainTextLength, char*& cipherText, int&
↪ cipherTextLength) {
2     int padding = 8 - plainTextLength % 8;
3     cipherTextLength = plainTextLength + padding;
4     cipherText = new char[cipherTextLength];
5     for (int i = 0; i < plainTextLength; i++) {
6         cipherText[i] = plainText[i];
7     }
8     for (int i = plainTextLength; i < cipherTextLength; i++) {
9         cipherText[i] = padding;
10    }
11
12    uint8_t plainTextBlock[8], cipherTextBlock[8];
13    for (int i = 0; i < cipherTextLength; i += 8) {
14        for (int j = 0; j < 8; j++) {
15            plainTextBlock[j] = cipherText[i + j];
16        }
17        DES(plainTextBlock, cipherTextBlock, true);
18        for (int j = 0; j < 8; j++) {
19            cipherText[i + j] = cipherTextBlock[j];
20        }
21    }
22 }
```

### Encrypt 函数解释

首先计算需要填充的字节数，然后分配足够的空间存储密文，并将明文复制到密文中，同时进行填充。接着将密文分成 8 字节的块，对每个块调用 DES 函数进行加密，并将加密后的结果存储回密文中。

### Decrypt 函数

```
1 void DesOp::Decrypt(char* cipherText, int cipherTextLength, char*& plainText, int&
  ↩ plainTextLength) {
2     uint8_t plainTextBlock[8], cipherTextBlock[8];
3     for (int i = 0; i < cipherTextLength; i += 8) {
4         for (int j = 0; j < 8; j++) {
5             cipherTextBlock[j] = cipherText[i + j];
6         }
7         DES(cipherTextBlock, plainTextBlock, false);
8         for (int j = 0; j < 8; j++) {
9             cipherText[i + j] = plainTextBlock[j];
10        }
11    }
12
13    int padding = cipherText[cipherTextLength - 1];
14    plainTextLength = cipherTextLength - padding;
15    plainText = new char[plainTextLength + 1];
16    for (int i = 0; i < plainTextLength; i++) {
17        plainText[i] = cipherText[i];
18    }
19    plainText[plainTextLength] = '\0';
20 }
```

### Decrypt 函数解释

首先将密文分成 8 字节的块，对每个块调用 DES 函数进行解密，并将解密后的结果存储回密文中。然后根据填充字节数去除填充部分，得到原始的明文，并将明文存储到新分配的内存中。

这样，我们就实现了 DES 加解密的全部函数啦。

## 5.3 基于 TCP 协议的聊天室实现

该部分与上学期的《计算机网络》实验一基本相同，所以可以很轻松地完成该部分的代码。

### 5.3.1 chat.h

该部分我们定义了 TCP 聊天所需要的一些变量和函数，具体代码如下所示。

首先展示一下为了实现 TCP 通信而定义的一些变量。

```
1 #define DEFAULT_SERVER_IP "127.0.0.1" // 默认服务器 IP 地址
2 #define DEFAULT_SERVER_PORT 8888      // 默认服务器端口号
3 #define MAX_MESSAGE_LENGTH 512        // 最大消息长度
4 #define EXIT_COMMAND "quit"          // 退出命令
5 #define KEY "Luhaozhe"                // DES 密钥，一共八位
```

可以看到，我们将默认的 IP 地址定义为 127.0.0.1，端口号定义为 8888，最大消息长度为 512，使用 quit 进行退出，DES 的八位密钥为我的姓名英文 Luhaozhe。

然后，我们还定义了许多函数，如下所示：

```
1 class Chat {
2 private:
3     bool isServer = false;           // 是否为服务器
4     int serverSocket = -1;            // 服务器套接字
5     int clientSocket = -1;           // 客户端套接字
6     const char* serverIp = DEFAULT_SERVER_IP; // 服务器 IP 地址
7     int serverPort = DEFAULT_SERVER_PORT;    // 服务器端口号
8     char message[MAX_MESSAGE_LENGTH] = {0};  // 消息缓冲区
9     char buffer[MAX_MESSAGE_LENGTH] = {0};   // 数据缓冲区
10    std::atomic<bool> isRunning = false;      // 运行状态
11    std::thread receiveThread;                 // 接收线程
12    DesOp des;                                // DES 操作类
13
14    void Init();                               // 初始化
15    void Connect();                             // 连接服务器
16    void Send();                                // 发送消息
17    void ReceiveThread();                       // 接收线程
18    void Close();                              // 关闭连接
19
20 public:
21     Chat();                                   // 构造函数
22     ~Chat();                                  // 析构函数
23     void RunServer();                          // 运行服务器
24     void RunClient();                          // 运行客户端
25 };
```

各部分的功能在上述代码中已经进行了注释，后续在 cpp 文件中也会进行详细的解释，因此此处不再详细阐述。

### 5.3.2 chat.cpp

下面，我们分别对各函数进行分析。

## Chat 函数

```
1 Chat::Chat() {
2     Init();
3     #ifdef _WIN32
4         WSADATA wsaData;
5         if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
6             std::cerr << "Error: Failed to initialize winsock." << std::endl;
7         }
8     #endif
9 }
```

### Chat 函数解释

如果是 linux 环境下，我们就直接通过 init 函数来进行初始化；如果是 win32 环境下，我们就使用 WSAStartup 来进行初始化。

## ~Chat 函数

```
1 Chat::~~Chat() {
2     Close();
3     #ifdef _WIN32
4         WSACleanup();
5     #endif
6 }
```

### ~Chat 函数解释

直接使用 close 函数来进行析构，如果是 win 系统的话，需要额外使用 WSACleanup 来进行清理。

## Init 函数

```
1 void Chat::Init() {
2     isRunning = false;
3     des.SetKey(KEY);
4 }
```

### Init 函数解释

首先，将 isRunning 的值调整为 false，为 DES 盒添加对应的八位密钥。

## 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
8     sockaddr_in serverAddr{};
9     serverAddr.sin_family = AF_INET;
10    serverAddr.sin_port = htons(serverPort);
11    serverAddr.sin_addr.s_addr = inet_addr(serverIp);
12
13    if (connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) <
    ↪ 0) {
14        std::cerr << "Error: Failed to connect to server." << std::endl;
15        return;
16    }
17    isRunning = true;
18
19    receiveThread = std::thread(&Chat::ReceiveThread, this);
20 }
```

### Connect 函数解释

进行 TCP 的连接，使用 socket 函数来进行连接。定义我们的服务地址、端口和 ip 族，然后开启连接，将 isRunning 变量的值调整为 true，代表正在运行中。然后新开一个线程，执行 ReceiveThread 函数来接收我们的消息。

### Send 函数

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

### Send 函数解释

首先，从标准输入读取一行消息；然后调用 `des.Encrypt` 函数对消息进行加密；尝试将加密后的消息发送给服务器，若发送失败，输出错误信息。最后，释放加密后的消息所占用的内存。

### 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         memset(buffer, 0, sizeof(buffer));
8         ssize_t len = recv(clientSocket, buffer, sizeof(buffer), 0);
9         if (len <= 0) {
10             std::cerr << "Error: Failed to receive message or connection closed."
11             << std::endl;
12             isRunning = false;
13             break;
14         }
15         des.Decrypt(buffer, len, plainText, plainTextLength);
16         std::cout << info << ": " << plainText << std::endl;
17         delete[] plainText;
18
19         if(0==memcmp("quit",buffer,4)){
20             std::cout << "Connection closed by " << info << "." << std::endl;
21             isRunning = false;
22             break;
23         }
24     }
25 }
```

### ReceiveThread 函数解释

在 `isRunning` 标志为 `true` 的情况下，持续接收消息。首先，清空接收缓冲区，尝试接收消息，若接收失败或连接关闭，输出错误信息并将 `isRunning` 标志设置为 `false`。然后，调用 `des.Decrypt` 函数对接收的消息进行解密。最后，输出解密后的消息，并释放解密后的消息所占用的内存。另外，需要实现输入 `quit` 后断开的功能，我们通过判断输入的 `buffer` 内容是否为 `quit`，如果是的话就将 `isRunning` 的值修改为 `false`，同时 `break` 退出循环即可。

### Close 函数

```
1 void Chat::Close() {
2     isRunning = false;
3     if (serverSocket >= 0) {
4         close(serverSocket);
5     }
6     if (clientSocket >= 0) {
7         close(clientSocket);
8     }
9     if (receiveThread.joinable()) {
10        receiveThread.join();
11    }
12 }
```

### Close 函数解释

关闭连接，首先将 isRunning 变量的值修改为 false。然后，分别关闭服务器套接字、客户端套接字，等待接收线程结束即可。

### RunServer 函数

```
1 void Chat::RunServer() {
2     isServer = true;
3
4     serverSocket = socket(AF_INET, SOCK_STREAM, 0);
5     if (serverSocket < 0) {
6         std::cerr << "Error: Failed to create socket." << std::endl;
7         return;
8     }
9
10    sockaddr_in serverAddr{};
11    serverAddr.sin_family = AF_INET;
12    serverAddr.sin_port = htons(serverPort);
13    serverAddr.sin_addr.s_addr = INADDR_ANY;
14
15    if (bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0)
16    ↪ {
17        std::cerr << "Error: Failed to bind." << std::endl;
18        return;
19    }
20
21    if (listen(serverSocket, 1) < 0) {
22        std::cerr << "Error: Failed to listen." << std::endl;
23    }
```

```
22         return;
23     }
24
25     sockaddr_in clientAddr{};
26     socklen_t clientAddrLen = sizeof(clientAddr);
27     clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr,
28         ↪ &clientAddrLen);
29     if (clientSocket < 0) {
30         std::cerr << "Error: Failed to accept." << std::endl;
31         return;
32     }
33
34     isRunning = true;
35     receiveThread = std::thread(&Chat::ReceiveThread, this);
36
37     while (isRunning) {
38         Send();
39     }
40
41     Close();
42 }
```

### RunServer 函数解释

首先，将 isServer 标志设置为 true；创建一个 TCP 套接字，若套接字创建失败，输出错误信息并返回。然后，配置服务器地址信息并绑定套接字，若绑定失败，输出错误信息并返回。然后，开始监听客户端连接，若监听失败，输出错误信息并返回。接受客户端连接，若接受失败，输出错误信息并返回。接着将 isRunning 标志设置为 true，表示聊天会话已启动。开启一个新线程执行 ReceiveThread 函数来接收消息。在 isRunning 标志为 true 的情况下，持续调用 Send() 函数发送消息。最后，调用 Close() 函数关闭聊天会话。

### RunClient 函数

```
1 void Chat::RunClient() {
2     isServer = false;
3     Connect();
4     while (isRunning) {
5         Send();
6     }
7     Close();
8 }
```

### RunClient 函数解释

首先，将 isServer 标志设置为 false，调用 Connect() 函数连接到服务器。在 isRunning 标志为 true 的情况下，持续调用 Send() 函数发送消息。最后，调用 Close() 函数关闭聊天会话。

## 6 实验遇到的问题及其解决方法

### 6.1 在 linux 上编译 C++ 项目

本次实验是我在 linux 系统上第一次进行 c++ 的编程，之前在数据安全的实验中，了解过 cmake 的编译和使用，但是都没有自己进行尝试。本次为了在 linux 上编译整个项目，我学习了 cmake 编译的方法，规范代码格式，最后成功完成了在 linux 上的代码编译与运行。

### 6.2 DES 相关算法实现

上学期在《密码学》课程中，我们只是简单学习了 DES 相关的算法步骤，但是没有自己去进行实现，因此在本次实验中，我独立完成了该算法的设计。并对 DES 更加了解了。

### 6.3 .gitignore 的使用

由于本人有一个习惯，就是在完成课程实验后，都会将自己的实验代码和报告都上传至 github，但是由于本次实验使用了 cmake，所以导致编译后的文件会多出许多内容。于是，我通过 .gitignore 的使用，成功解决了这个问题，并成功的上传了源代码到 github 上。

## 7 实验结论

下面我们来展示一下本次实验的结论。我们在 linux 服务器上，通过新建两个终端来模拟客户端和服务器的环境，最后完成了全部功能的测试。

### 7.1 利用 cmake 编译项目

我们使用以下命令行来对我们的项目进行编译：

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

其中，我们的 CMakeLists.txt 内容如下所示：

```
1 cmake_minimum_required(VERSION 3.21)
2 project(DES_chat)
3
4 set(CMAKE_CXX_STANDARD 17)
5 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
6
7 add_executable(DES_chat main.cpp
8               src/DES_Operation.cpp
```

```
9         src/chat.cpp)
10
11 target_include_directories(DES_chat PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
12
13 if(WIN32)
14     target_link_libraries(DES_chat ws2_32)
15 endif()
```

上述的 makefile 文件代表,我们使用 3.21 版本的 cmake 工具,对 main.cpp、src/DES\_Operation.cpp 和 src/chat.cpp 进行编译,得到可执行文件 DES\_chat。

我们进行测试,得到结果如下所示:


```
root@tomorin:/home/Network_Security_Technology/Lab01# cmake . -B build && cmake --build build
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/Network_Security_Technology/Lab01/build
[ 25%] Building CXX object CMakeFiles/DES_chat.dir/main.cpp.o
[ 50%] Building CXX object CMakeFiles/DES_chat.dir/src/DES_Operation.cpp.o
[ 75%] Building CXX object CMakeFiles/DES_chat.dir/src/chat.cpp.o
[100%] Linking CXX executable ../bin/DES_chat
[100%] Built target DES_chat
root@tomorin:/home/Network_Security_Technology/Lab01#
```

图 7.5: cmake 后的结果

可以发现我们成功完成了编译,然后,我们进入到 bin 文件目录下,输入命令行./DES\_chat,就可以对可执行文件进行执行了。

## 7.2 新建服务器端与客户端

执行可执行文件后,我们新开两个终端,先跳转到 bin 目录下,执行我们的可执行文件,分别输入 s 和 c,代表我们进入服务端和客户端,如下所示:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/h/Lab01# bash
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
[]

(base) Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin:/h/h/Lab01# bash
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): c
Server:
[]
```

图 7.6: 服务器端和客户端

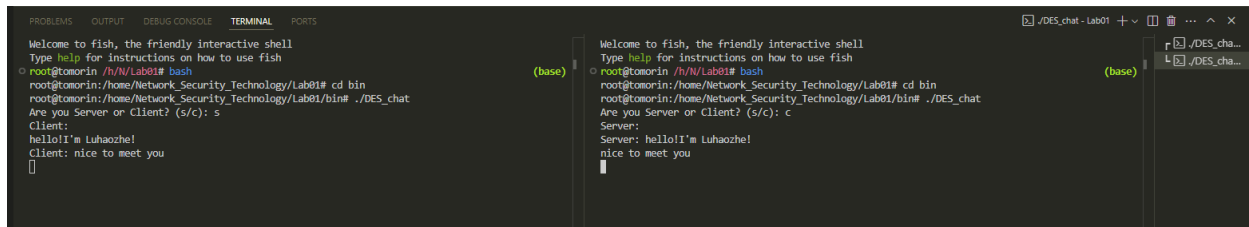
可以看到,我们的服务器端和客户端完成了相互的连接,说明我们的 TCP 编程是正确的!

## 7.3 功能测试

在完成对应的连接之后，我们开始进行功能的测试。

### 7.3.1 英文输入测试

我们首先在客户端输入一句英文，然后发现服务器端也出现了对应的英文；同样的，我们在服务器端输入一句英文，发现客户端也输出了对应的英文。

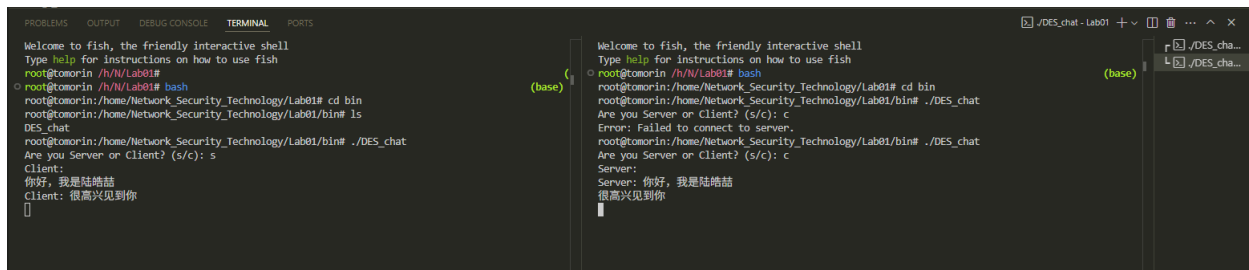


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin: /h/N/Lab01# bash
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
hello! I'm Luhaozhe!
Server:
nice to meet you
Client: nice to meet you
[]
```

图 7.7: 英文测试

### 7.3.2 中文输入测试

我们首先在客户端输入一句中文，然后发现服务器端也出现了对应的中文；同样的，我们在服务器端输入一句中文，发现客户端也输出了对应的中文。

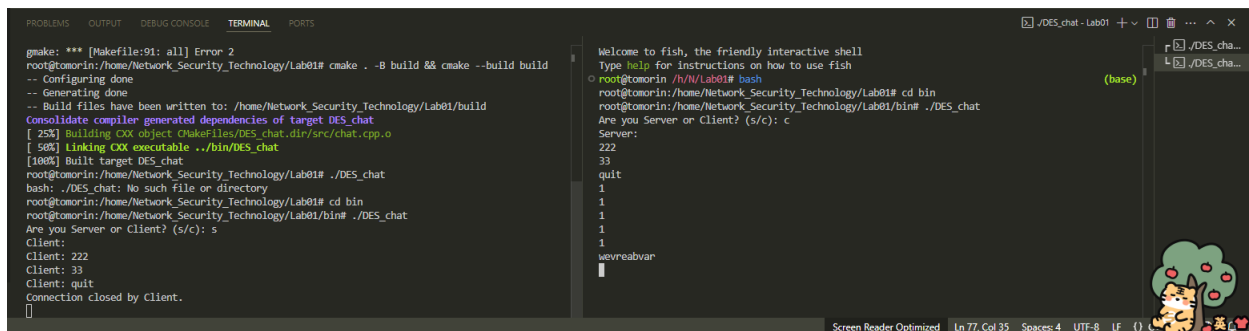


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@tomorin: /h/N/Lab01# bash
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ls
DES_chat
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
你好，我是陆皓喆
Server:
很高兴见到你
Client: 很高兴见到你
[]
```

图 7.8: 中文测试

### 7.3.3 断开连接测试

我们通过输入 quit，来检验我们的连接是否能正常断开。如下所示，首先我们在 client 端输入 quit，得到以下结果：

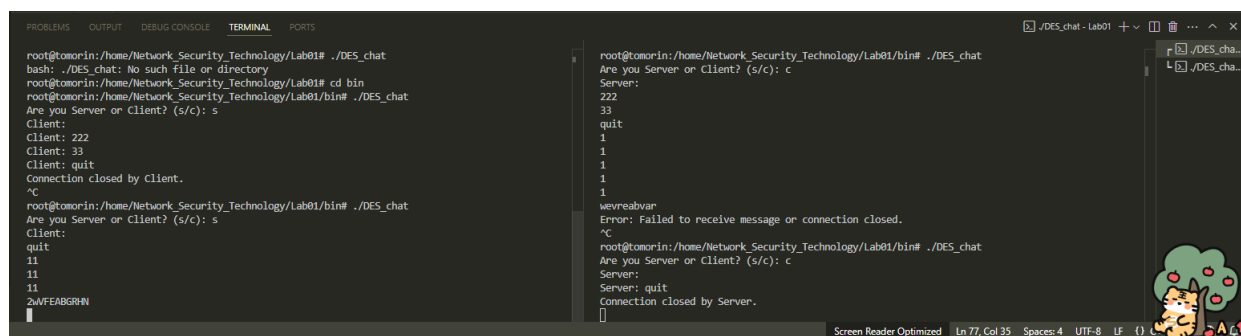


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
gnake: *** [Makefile:91: all] Error 2
root@tomorin:/home/Network_Security_Technology/Lab01# cmake . -B build && cmake --build build
-- Configuring done
-- Generating done
-- Build files have been written to: /home/Network_Security_Technology/Lab01/build
Consolidate compiler generated dependencies of target DES_chat
[ 25%] Building CXX object CMakeFiles/DES_chat.dir/src/chat.cpp.o
[ 50%] Linking CXX executable ../bin/DES_chat
[100%] Built target DES_chat
root@tomorin:/home/Network_Security_Technology/Lab01# ./DES_chat
bash: ./DES_chat: No such file or directory
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
Client: 222
Client: 33
Client: quit
Connection closed by Client.
[]
```

图 7.9: 客户端断开连接

我们发现，可以成功断开，我们在服务器中输入一些字符，不会在客户端中显示了！

然后我们在 server 端输入 quit，得到以下结果：



```
root@tomorin:/home/Network_Security_Technology/Lab01# ./DES_chat
bash: ./DES_chat: No such file or directory
root@tomorin:/home/Network_Security_Technology/Lab01# cd bin
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
Client: 222
Client: 33
Client: quit
Connection closed by Client.
^C
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): s
Client:
quit
11
11
11
2wFEABGRN

root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): c
Server:
222
33
quit
1
1
1
1
1
1
weureahvar
Error: Failed to receive message or connection closed.
^C
root@tomorin:/home/Network_Security_Technology/Lab01/bin# ./DES_chat
Are you Server or Client? (s/c): c
Server:
Server: quit
Connection closed by Server.
```

图 7.10: 服务器断开连接

我们同样发现，一边断开后，我们不能在另一边进行信息的传输了，验证成功！

综上所述，我们发现我们成功实现了我们所需要的功能，实验成功！

## 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：

```
Lab01
├── .gitignore
├── CMakeLists.txt
├── main.cpp
├── README.md
├── src
│   ├── chat.cpp
│   └── DES_Operation.cpp
├── include
│   ├── chat.h
│   └── DES_Operation.h
├── bin
│   └── DES_chat
└── build
```

## 9 参考文献

- 第 3 章：基于 DES 加密的 TCP 聊天程序
- 通俗易懂，十分钟读懂 DES，详解 DES 加密算法原理，DES 攻击手段以及 3DES 原理
- DES 加密算法原理与实现
- 在线 DES 加密 | DES 解密- 在线工具