北京郵電大學

《计算机网络》实验报告
基于 SOCKET 的文件传输

小组成员信息

姓名	班级	学号	分工
王何佳	2023211804	2023211603	代码构建、实验开展及报告撰写 50%
叶于琳	2023211803	2023211606	代码构建、实验开展及报告撰写 50%

2025 年 5 月 23 日

目录

一、实验内容

1. 实验目标

通过本实验，深入理解传输层的 TCP 协议原理以及 Socket 套接字网络通信的流程。

2. 实验内容

（一）使用 Socket API 通信实现文件传输客户端和服务器端 2 个程序，客户端发送文件传输请求，服务器端将文件数据发送给客户端，两个程序均在命令行方式下运行，要求至少能传输 1 个文本文件和 1 个图片文件（本实验测试了 txt、png、docx、pdf、zip）。
（二）客户端在命令行指定服务器的 IP 地址和文件名。为防止重名，客户端将收到的文件改名后保存在当前目录下。
（三）客户端应输出：新文件名、传输总字节数；或者差错报告；服务器端应输出：客户端的 IP 地址和端口号；发送的文件数据总字节数；必要的差错报告（如文件不存在）。
（四）支持 SSL 安全连接。

二、实验环境

两个 ubuntu 虚拟机，装有 OpenSSL 开发库、Wireshark，IP 如下：

server：192.168.130.139

client：192.168.130.140

编程语言：C++

开发工具：g++

三、软件设计

1. 数据结构

（一）服务器端（server.cpp）

```
struct sockaddr_in server_addr;            服务器监听地址（INADDR_ANY+端口）
struct sockaddr_in client_addr;            客户端连接地址（IP+端口）
char filename_buffer[256];                 存储客户端请求的文件名
char file_content_buffer[BUFFER_SIZE];     文件内容传输缓冲区
char file_exists_flag;                     文件存在标志
```

（二）客户端（client.cpp）

struct sockaddr_in server_addr;	服务器地址信息（IP+端口）
char buffer[BUFFER_SIZE];	文件传输缓冲区（默认 1024 字节）
char filename_to_request[256];	请求的文件名（从命令行参数读取）
std::string received_filename;	接收到的文件名（原文件名+.rcv）

（三）服务器端 SSL 加密（server_ssl.cpp）

SSL_CTX* ssl_ctx;	SSL 上下文
SSL* ssl;	SSL 会话对象
struct sockaddr_in server_addr;	服务器监听地址（INADDR_ANY+SSL 端口）
struct sockaddr_in client_addr;	客户端地址（IP+端口）
char filename_buffer[256];	客户端请求的文件名
char file_content_buffer[BUFFER_SIZE];	SSL 加密传输缓冲区
char file_exists_flag;	文件存在标志

（四）客户端 SSL 加密（client_ssl.cpp）

SSL_CTX* ssl_ctx;	SSL 上下文（全局）
SSL* ssl;	SSL 会话对象
struct sockaddr_in server_addr;	服务器地址（IP+SSL 端口 1146）
char buffer[BUFFER_SIZE];	SSL 加密传输缓冲区（1024 字节）
char filename_to_request[256];	请求的文件名
std::string received_filename;	接收文件名（原文件名+.rcv_ssl）

2. 主要功能模块的实现要点

（一）服务器端

- **网络监听与连接建立**
  - 创建套接字：使用 socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)创建 TCP 套接字。
  - 端口复用：通过 setsckopt 设置 SO_REUSEADDR 选项，允许服务器快速重启并绑定先前使用的端口。
  - 绑定地址与端口：调用 bind()将套接字与服务器 IP 地址（INADDR_ANY，监听所有可用网络接口）和指定端口(SERVER_PORT)关联。
  - 开始监听：listen()使服务器进入监听状态，准备接受客户端连接。
  - 接受连接：在主循环中，accept()阻塞等待并接受新的客户端连接，为每个连接返回一个新的套接字文件描述符。
  - 获取客户端信息: inet_ntop()和 ntohs()用于将网络字节序的客户端地址和端口转换成字符串和主机字节序整数，方便日志记录。
- **客户端会话处理 (handle_client_session 函数)**
  - 接收文件名：调用 recv()从客户端接收请求的文件名，存储于 filename_buffer。

- 文件存在性检查：使用 std::ifstream 以二进制读取模式(std::ios::binary | std::ios::in)尝试打开文件，并通过 is_open()判断文件是否成功打开。
- 发送文件状态：send()一个单字节的 file_exists_flag (1 表示存在，0 表示不存在)给客户端。
- 文件内容传输：
  - 若文件存在，通过 fin.read()将文件内容分块读入 file_content_buffer，fin.gcount()获取实际读取的字节数。
  - 使用一个内部 while 循环配合 send()，确保 file_content_buffer 中的数据块被完整发送，处理 send()可能只发送部分数据的情况。
- 关闭连接：会话结束后，close()关闭与该客户端连接的套接字。

（二）客户端

连接服务器 (connect_with_retry 函数)

- 创建套接字：socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)创建 TCP 套接字。
- 配置服务器地址：inet_pton()将命令行传入的服务器 IP 地址字符串转换为网络字节序格式，并设置 server_addr.sin_port 为 htons(SERVER_PORT)。
- 尝试连接与重连：while 循环中调用 connect()尝试连接。若失败，sleep(1)后重试，最多 MAX_CONNECT_RETRY 次。

文件请求与接收

- 发送文件名：连接成功后，send()将 filename_to_request (包含末尾的\0，长度为 strlen(filename_to_request) + 1)发送给服务器。
- 接收文件状态：recv()接收服务器返回的 file_exists_flag。
- 接收文件内容：
  - 若文件存在(file_exists_flag == 1)，以二进制写模式(std::ios::binary | std::ios::out)创建本地文件(received_filename)。
  - while 循环中调用 recv()从服务器接收数据块到 buffer，直到 recv()返回 0（连接关闭）或负值（错误）。
  - 接收到的数据用 fout.write()写入本地文件。
- 错误处理：若 recv()在传输过程中返回错误，remove(received_filename.c_str())删除可能不完整的本地文件。
- 关闭连接：close()关闭与服务器的套接字。

（三）服务器（ssl）

SSL 上下文初始化与配置 (initialize_server_ssl_ctx 函数)

- OpenSSL 库初始化：SSL_library_init()、OpenSSL_add_all_algorithms()、SSL_load_error_strings()。
- 创建 SSL 上下文：ssl_ctx = SSL_CTX_new(TLS_server_method())。
- 加载证书与私钥：SSL_CTX_use_certificate_file()加载 server.crt，SSL_CTX_use_PrivateKey_file()加载 server.key。
- 验证密钥与证书：SSL_CTX_check_private_key()确保私钥与证书匹配。

网络监听与 SSL 握手

- TCP 层面的监听与连接建立（socket, bind, listen, accept）与非 SSL 版本类似，但监听于 SSL_SERVER_PORT。

  - TCP 连接 accept()成功后：
    - 创建 SSL 会话对象：ssl = SSL_new(ssl_ctx)。
    - 关联 SSL 对象与 TCP 套接字：SSL_set_fd(ssl, client_sock_fd)。
    - 执行 SSL 握手：SSL_accept(ssl)。若出错，通过 ERR_print_errors_fp(stderr) 打印 OpenSSL 错误。

- **安全会话处理 (handle_client_session_ssl 函数)**
  - 安全接收文件名：SSL_read()通过加密通道接收文件名。
  - 安全发送文件状态：SSL_write()发送 file_exists_flag。
  - 安全文件内容传输:
    - 若文件存在，读取文件到缓冲区。
    - 内部 while 循环调用 SSL_write()安全发送数据块。错误通过 SSL_get_error() 和 ERR_print_errors_fp(stderr)获取和打印。

- **SSL 会话关闭与资源释放**
  - 关闭 SSL 会话：SSL_shutdown(ssl)。
  - 释放 SSL 对象：SSL_free(ssl)。
  - 关闭 TCP 套接字：close(client_sock_fd)。
  - 服务器退出时：SSL_CTX_free(ssl_ctx)和 EVP_cleanup()。

（四）客户端（ssl）

- **SSL 上下文初始化 (initialize_client_ssl_ctx 函数)**
  - OpenSSL 库初始化：同服务器端。
  - 创建 SSL 上下文：ssl_ctx = SSL_CTX_new(TLS_client_method())。
  - 配置证书验证模式：SSL_CTX_set_verify(ssl_ctx, SSL_VERIFY_NONE, NULL)设置为不验证服务器证书 （为实验简化）。

- **建立 TCP 连接与 SSL 握手**
  - 通过 connect_tcp_with_retry()建立与 SSL 服务器(SSL_SERVER_PORT)的基础 TCP 连接。
  - TCP 连接成功后：
    - 创建 SSL 会话对象：ssl = SSL_new(ssl_ctx)。
    - 关联 SSL 对象与 TCP 套接字：SSL_set_fd(ssl, sock_fd)。
    - 执行 SSL 握手：SSL_connect(ssl)。若出错，通过 ERR_print_errors_fp(stderr) 打印 OpenSSL 错误。

- **安全文件请求与接收**
  - 安全发送文件名：SSL_write()通过加密通道发送文件名。
  - 安全接收文件状态：SSL_read()接收 file_exists_flag。
  - 安全接收文件内容：
    - 若文件存在，创建本地文件(original_name.rcv_ssl)。
    - while 循环中 SSL_read()安全接收数据。错误通过 SSL_get_error()和 ERR_print_errors_fp(stderr)获取和打印。
    - 接收到的数据写入本地文件。

- **SSL 会话关闭与资源释放**

        - 同 SSL 服务器端对应部分：SSL_shutdown(ssl)、SSL_free(ssl)、close(sock_fd)，以及客户端退出时的 SSL_CTX_free(ssl_ctx)和 EVP_cleanup()。

3. 模块结构

（一）服务器端

- 网络初始化(socket, setsockopt, bind, listen)
- accept() (循环中)
    - handle_client_session()
        - recv() (接收文件名)
        - 文件操作(std::ifstream 系列)
        - send() (发送状态和文件内容)
        - close() (客户端 socket)

（二）客户端

- 网络初始化(socket)
- connect_with_retry()
    - connect()
- send() (发送文件名)
- recv() (接收状态和文件内容)
- 文件操作: (std::ofstream 系列)
- close() (客户端 socket)

（三）服务器（ssl）

- initialize_server_ssl_ctx() (SSL 上下文初始化)
- 网络初始化(socket, setsockopt, bind, listen)
- accept() (循环中)
    - SSL 握手预备(SSL_new, SSL_set_fd)
    - SSL_accept() (执行 SSL 握手)
    - handle_client_session_ssl()
        - SSL_read() (安全接收文件名)
        - 文件操作 (std::ifstream 系列)
        - SSL_write() (安全发送状态和文件内容)
    - SSL_shutdown(), SSL_free(), close() (客户端会话清理)
- 服务器退出清理(SSL_CTX_free, EVP_cleanup)

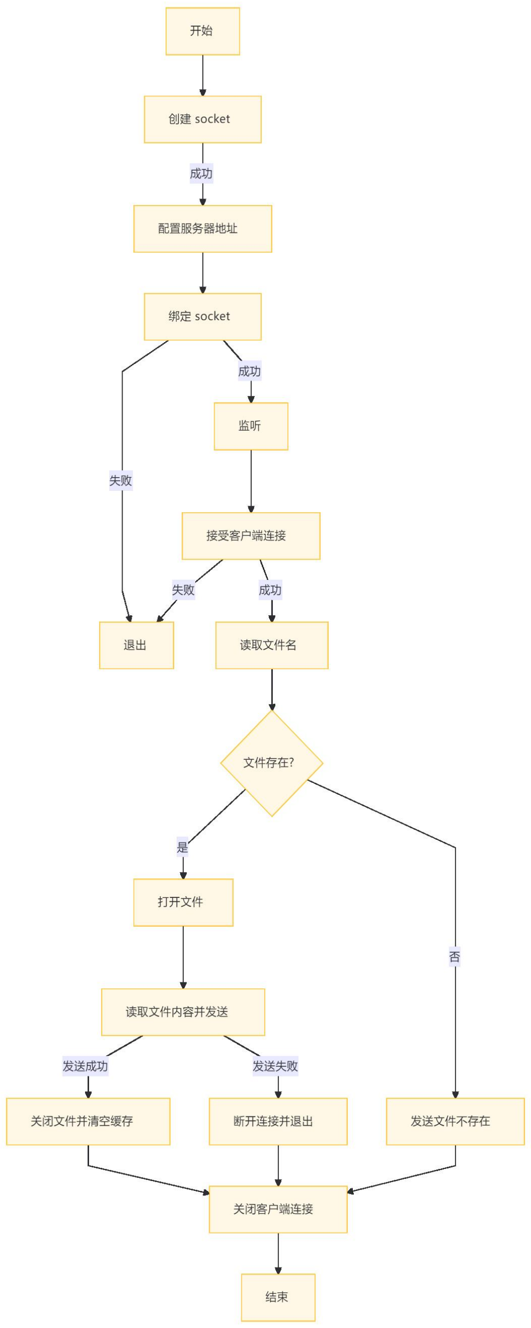
（四）客户端（ssl）

- initialize_client_ssl_ctx() (SSL 上下文初始化)
- 网络初始化(socket)

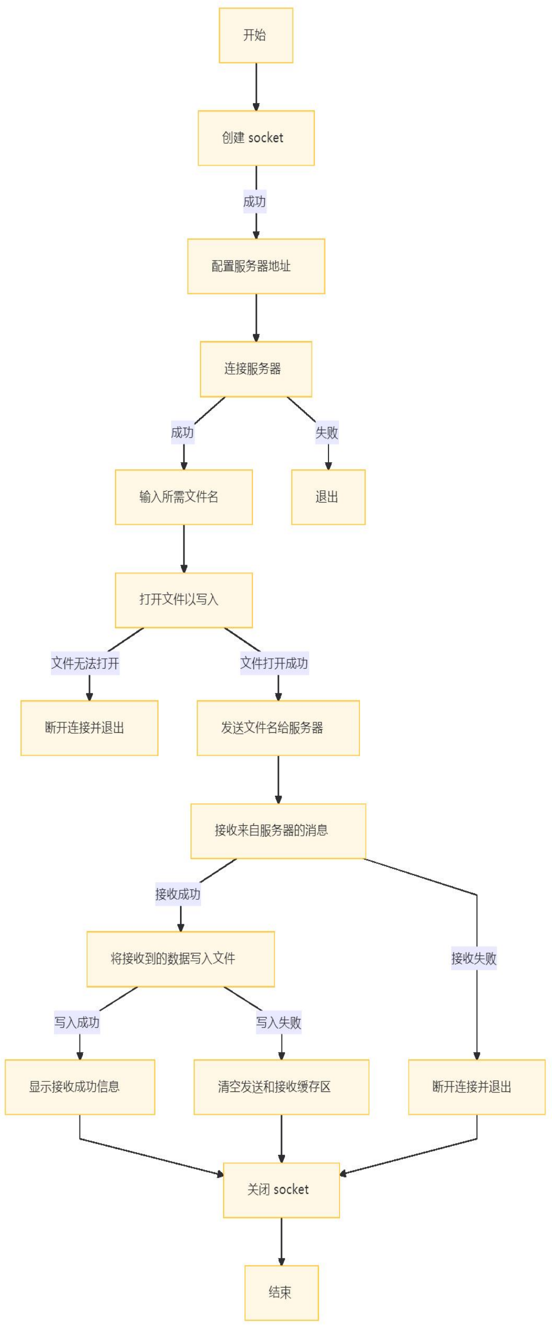
- connect_tcp_with_retry() (建立 TCP 连接)
- SSL 握手预备(SSL_new, SSL_set_fd)
- SSL_connect() (执行 SSL 握手)
- SSL_write() (安全发送文件名)
- SSL_read() (安全接收状态和文件内容)
- 文件操作(std::ofstream 系列)
- 客户端退出清理(SSL_shutdown, SSL_free, close, SSL_CTX_free,EVP_cleanup)

4. 程序流程图

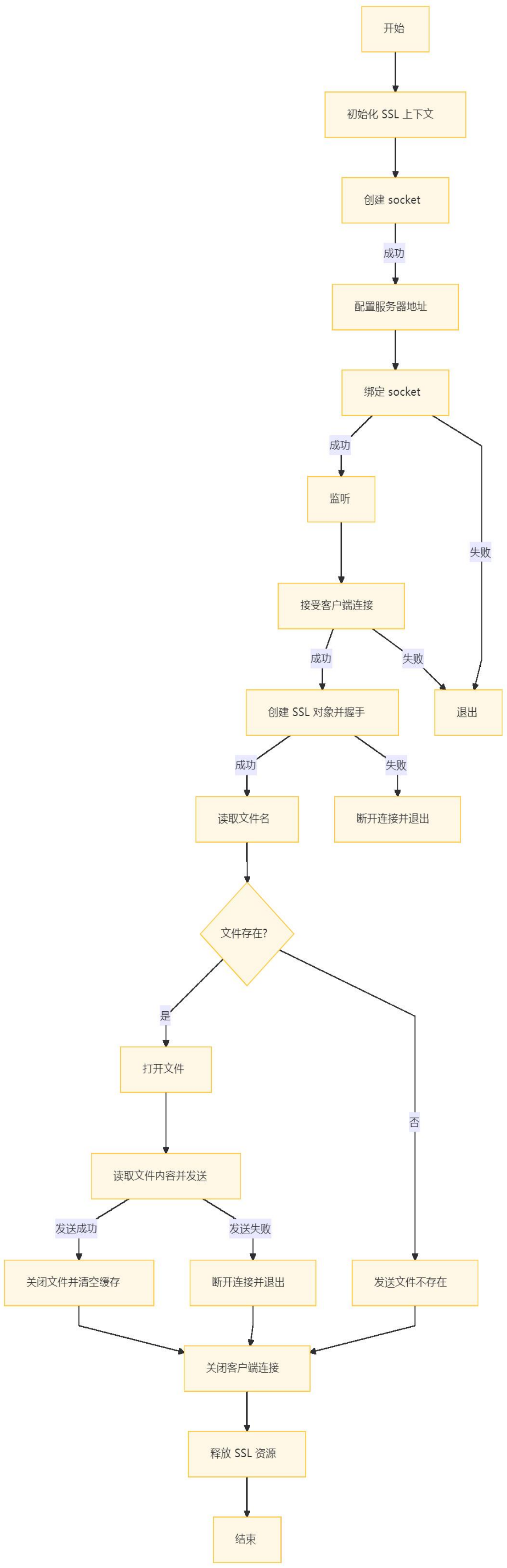
（一）服务器



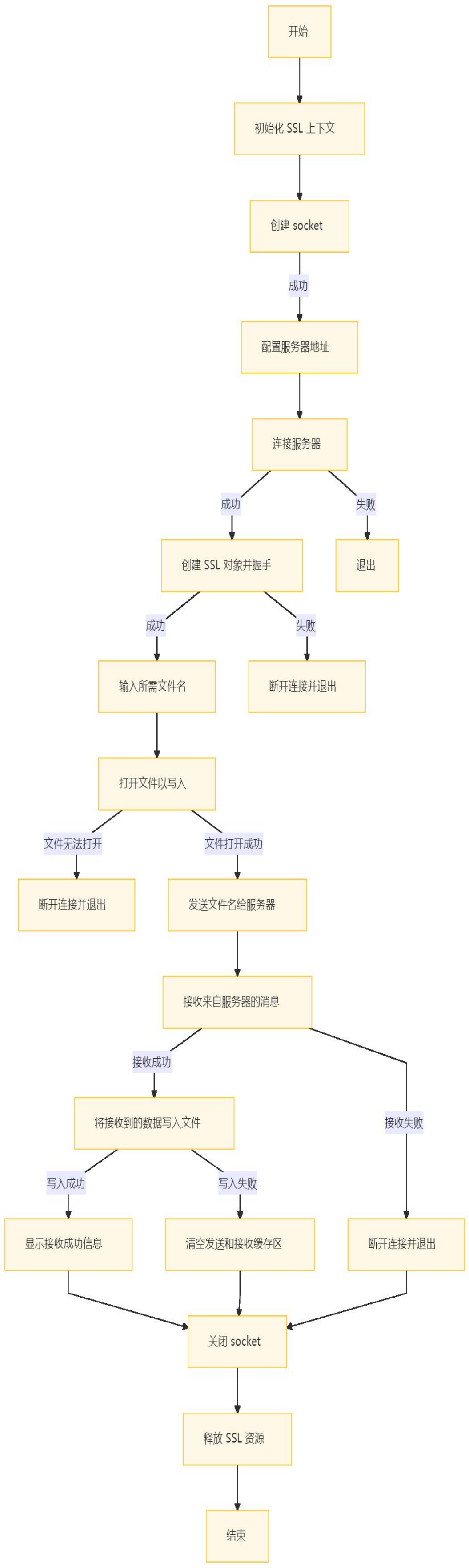
客户端



（二）服务器（ssl）



客户端（ssl）

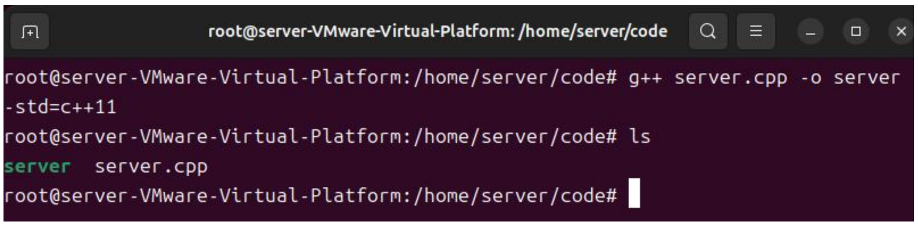


四、实验步骤

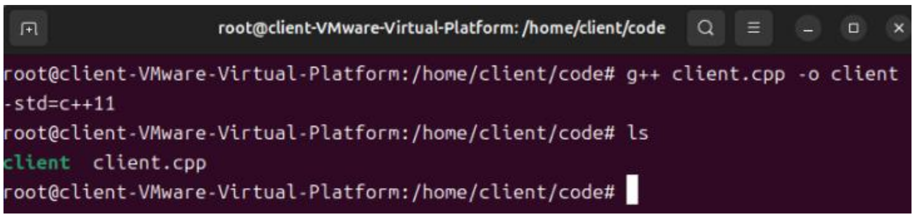
1. 编译

分别在 server 和 client 端编译代码

```
g++ server.cpp -o server -std=c++11
g++ client.cpp -o client -std=c++11
```



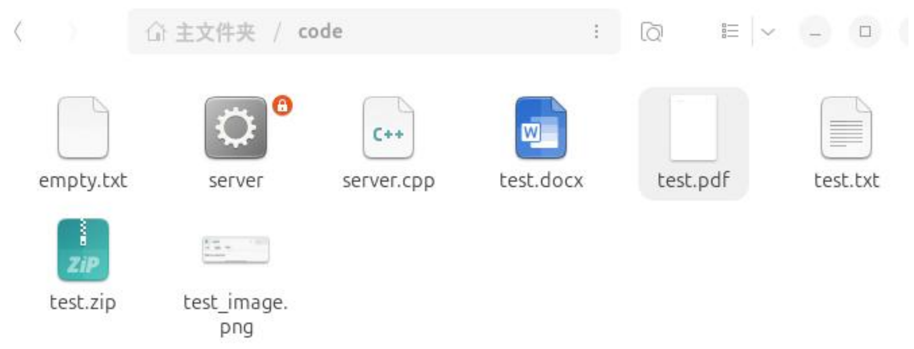
```
root@server-VMware-Virtual-Platform: /home/server/code
root@server-VMware-Virtual-Platform:/home/server/code# g++ server.cpp -o server
-std=c++11
root@server-VMware-Virtual-Platform:/home/server/code# ls
server  server.cpp
root@server-VMware-Virtual-Platform:/home/server/code#
```

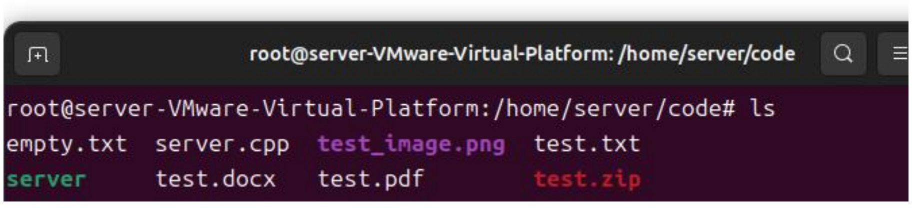


```
root@client-VMware-Virtual-Platform: /home/client/code
root@client-VMware-Virtual-Platform:/home/client/code# g++ client.cpp -o client
-std=c++11
root@client-VMware-Virtual-Platform:/home/client/code# ls
client  client.cpp
root@client-VMware-Virtual-Platform:/home/client/code#
```

2. 准备测试文件（在服务器虚拟机上）

包括：test.txt、test.docx、test.pdf、test.zip、test_image.png 以及一个空文件 empty.txt

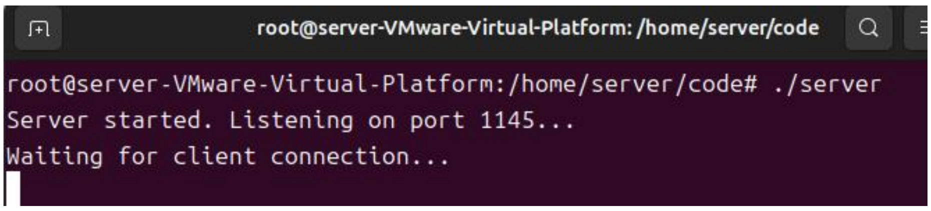




```
root@server-VMware-Virtual-Platform: /home/server/code
root@server-VMware-Virtual-Platform:/home/server/code# ls
empty.txt   server.cpp   test_image.png   test.txt
server      test.docx    test.pdf         test.zip
```

3. 启动服务器

```
./server
```

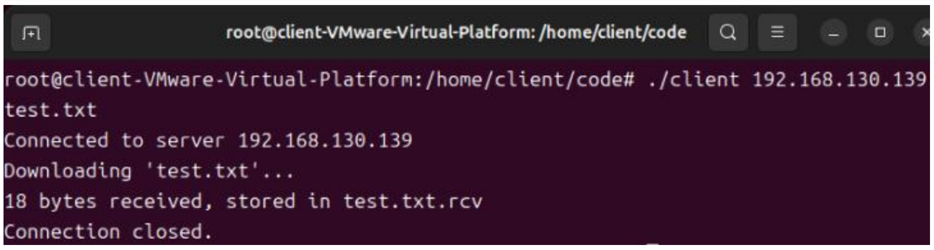


```
root@server-VMware-Virtual-Platform:/home/server/code# ./server
Server started. Listening on port 1145...
Waiting for client connection...
```

4. 客户端请求测试文件

（一）请求接收 txt 文件

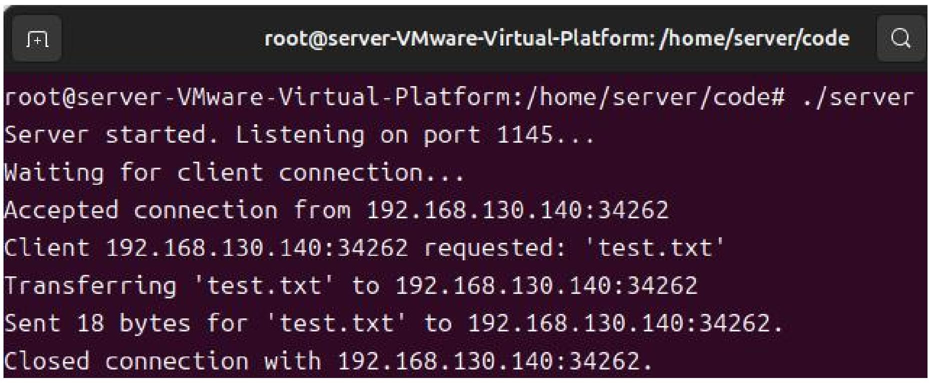
```
./client 192.168.130.139 test.txt
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test.txt
Connected to server 192.168.130.139
Downloading 'test.txt'...
18 bytes received, stored in test.txt.rcv
Connection closed.
```

显示接收到文件，存为 test.txt.rcv，共 18bytes

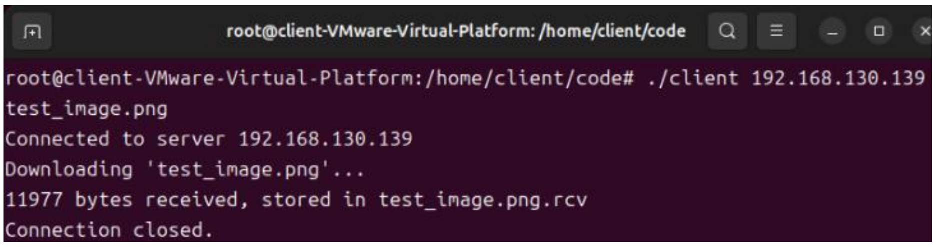
服务器端输出：客户端 IP 地址和端口号（192.168.130.140:34262）、发送的文件数据总字节数（18 bytes）



```
root@server-VMware-Virtual-Platform:/home/server/code# ./server
Server started. Listening on port 1145...
Waiting for client connection...
Accepted connection from 192.168.130.140:34262
Client 192.168.130.140:34262 requested: 'test.txt'
Transferring 'test.txt' to 192.168.130.140:34262
Sent 18 bytes for 'test.txt' to 192.168.130.140:34262.
Closed connection with 192.168.130.140:34262.
```

（二）请求接收 png 图片

```
./client 192.168.130.139 test_image.png
```

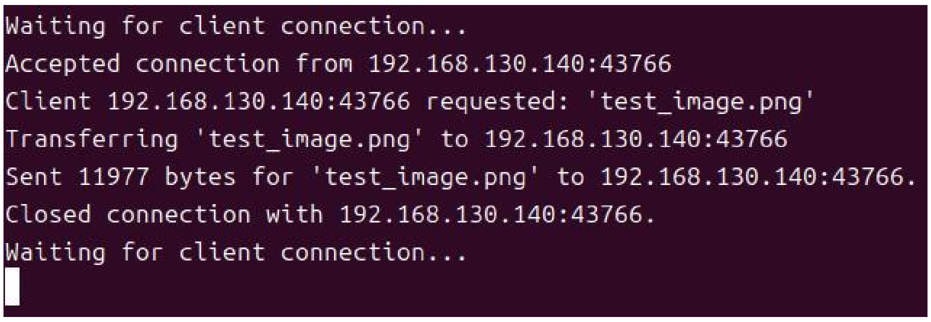


```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test_image.png
Connected to server 192.168.130.139
Downloading 'test_image.png'...
11977 bytes received, stored in test_image.png.rcv
Connection closed.
```

显示接收到文件，存为 test_image.png.rcv，共 11977bytes

服务器端输出：客户端 IP 地址和端口号（192.168.130.140:43766）、发送的文件

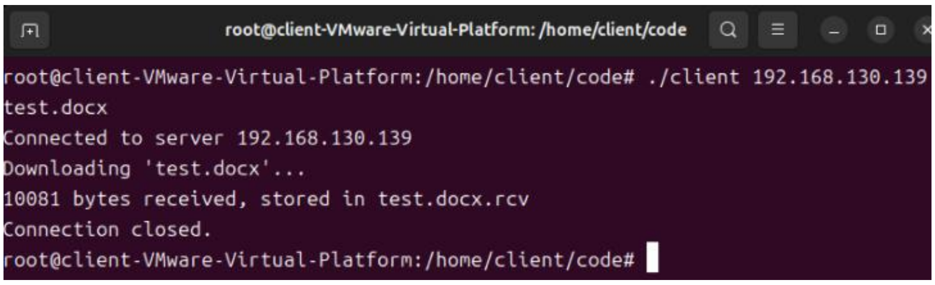
数据总字节数（11977bytes）



```
Waiting for client connection...
Accepted connection from 192.168.130.140:43766
Client 192.168.130.140:43766 requested: 'test_image.png'
Transferring 'test_image.png' to 192.168.130.140:43766
Sent 11977 bytes for 'test_image.png' to 192.168.130.140:43766.
Closed connection with 192.168.130.140:43766.
Waiting for client connection...
```

（三）请求接收 docx 文件

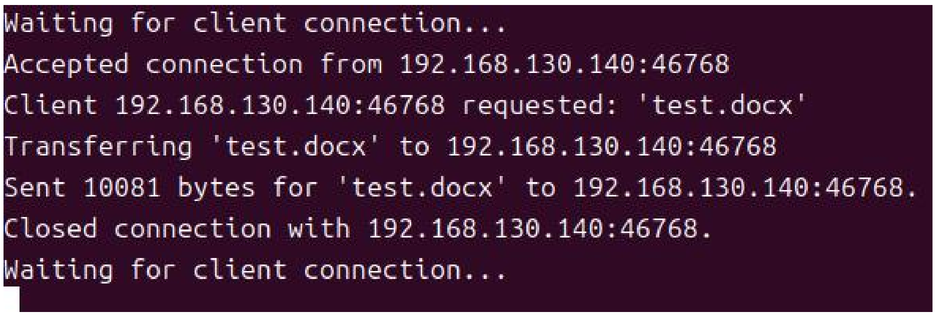
```
./client 192.168.130.139 test.docx
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test.docx
Connected to server 192.168.130.139
Downloading 'test.docx'...
10081 bytes received, stored in test.docx.rcv
Connection closed.
root@client-VMware-Virtual-Platform:/home/client/code#
```

显示接收到文件，存为 test.docx.rcv，共 10081bytes

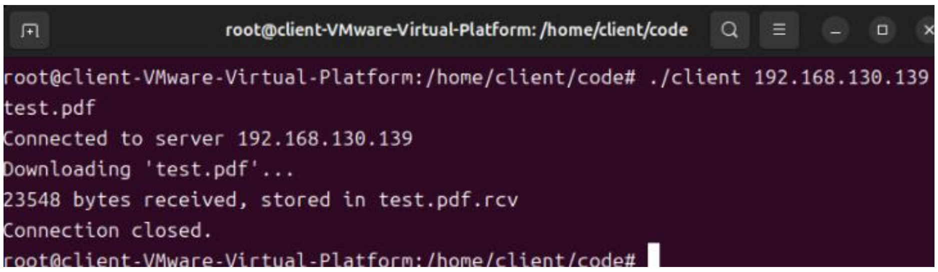
服务器端输出：客户端 IP 地址和端口号（192.168.130.140:46768）、发送的文件数据总字节数（10081bytes）



```
Waiting for client connection...
Accepted connection from 192.168.130.140:46768
Client 192.168.130.140:46768 requested: 'test.docx'
Transferring 'test.docx' to 192.168.130.140:46768
Sent 10081 bytes for 'test.docx' to 192.168.130.140:46768.
Closed connection with 192.168.130.140:46768.
Waiting for client connection...
```

（四）请求接收 pdf 文件

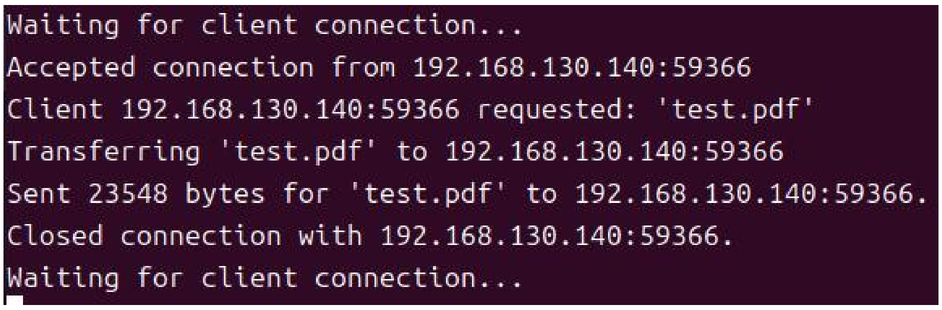
```
./client 192.168.130.139 test.pdf
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test.pdf
Connected to server 192.168.130.139
Downloading 'test.pdf'...
23548 bytes received, stored in test.pdf.rcv
Connection closed.
root@client-VMware-Virtual-Platform:/home/client/code#
```

显示接收到文件，存为 test.pdf.rcv，共 23548bytes

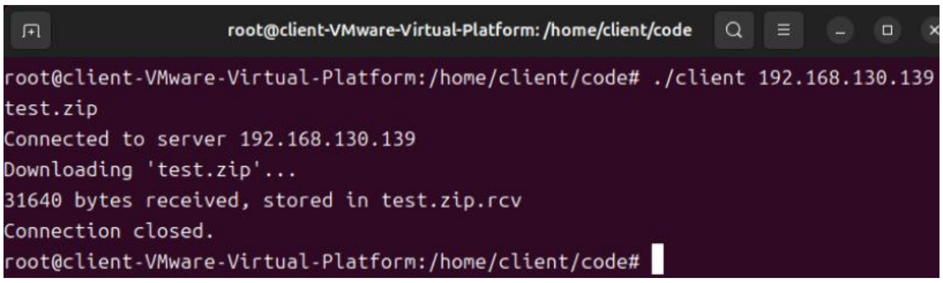
服务器端输出：客户端 IP 地址和端口号（192.168.130.140:59366）、发送的文件数据总字节数（23548bytes）



```
Waiting for client connection...
Accepted connection from 192.168.130.140:59366
Client 192.168.130.140:59366 requested: 'test.pdf'
Transferring 'test.pdf' to 192.168.130.140:59366
Sent 23548 bytes for 'test.pdf' to 192.168.130.140:59366.
Closed connection with 192.168.130.140:59366.
Waiting for client connection...
```

（五）请求接收 zip 压缩包

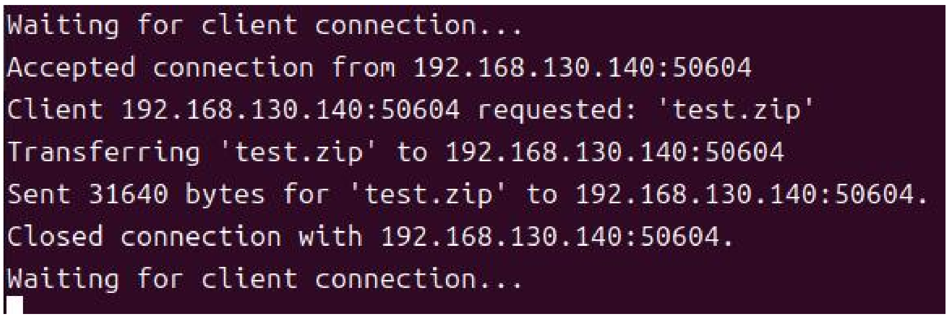
```
./client 192.168.130.139 test.zip
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test.zip
Connected to server 192.168.130.139
Downloading 'test.zip'...
31640 bytes received, stored in test.zip.rcv
Connection closed.
root@client-VMware-Virtual-Platform:/home/client/code# 
```

显示接收到文件，存为 test.zip.rcv，共 31640bytes

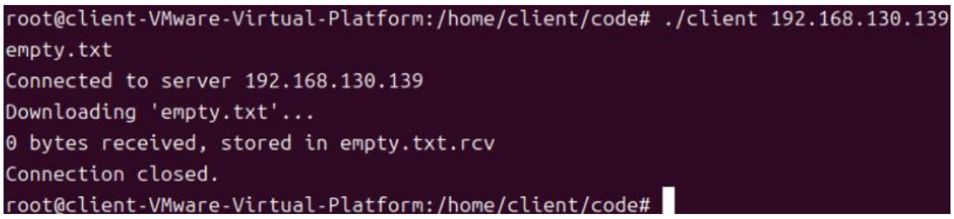
服务器端输出：客户端 IP 地址和端口号（192.168.130.140:50604）、发送的文件数据总字节数（31640bytes）



```
Waiting for client connection...
Accepted connection from 192.168.130.140:50604
Client 192.168.130.140:50604 requested: 'test.zip'
Transferring 'test.zip' to 192.168.130.140:50604
Sent 31640 bytes for 'test.zip' to 192.168.130.140:50604.
Closed connection with 192.168.130.140:50604.
Waiting for client connection...
```

（六）客户端请求接收空文件

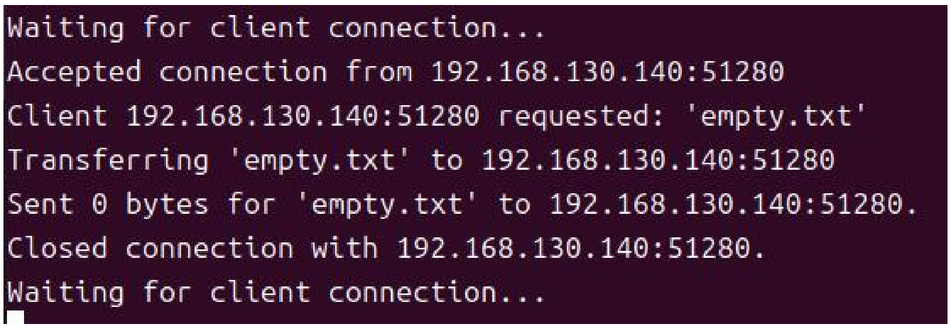
```
./client 192.168.130.139 empty.txt
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
empty.txt
Connected to server 192.168.130.139
Downloading 'empty.txt'...
0 bytes received, stored in empty.txt.rcv
Connection closed.
root@client-VMware-Virtual-Platform:/home/client/code#
```

显示接收到文件，存为 empty.txt.rcv，共 0bytes

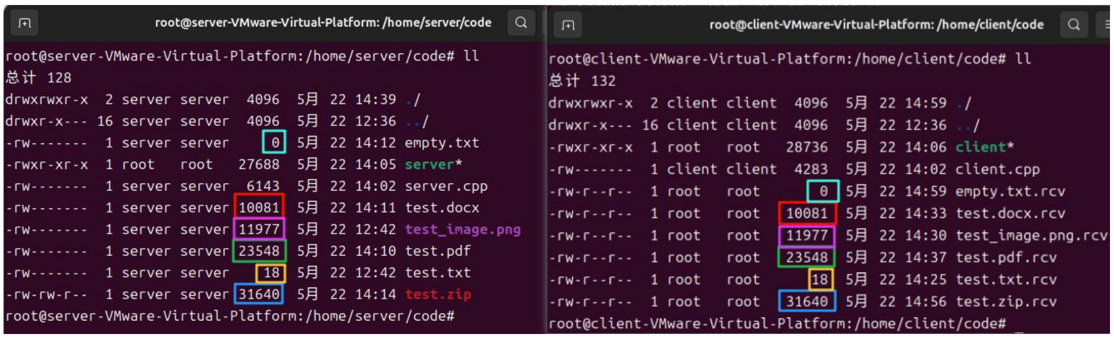
服务器端输出：客户端 IP 地址和端口号（192.168.130.140:51280）、发送的文件数据总字节数（0bytes）



```
Waiting for client connection...
Accepted connection from 192.168.130.140:51280
Client 192.168.130.140:51280 requested: 'empty.txt'
Transferring 'empty.txt' to 192.168.130.140:51280
Sent 0 bytes for 'empty.txt' to 192.168.130.140:51280.
Closed connection with 192.168.130.140:51280.
Waiting for client connection...
```

5. 验证文件一致性

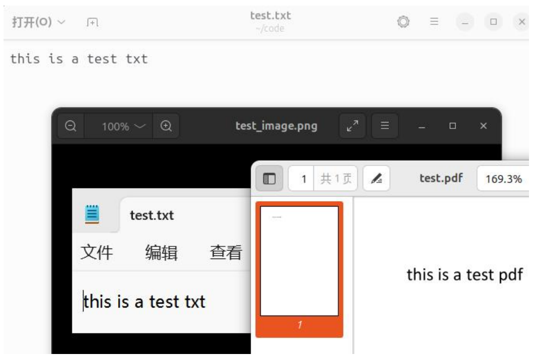
文件大小完全一致


root@server-VMware-Virtual-Platform: /home/server/code
root@client-VMware-Virtual-Platform: /home/client/code

```
root@server-VMware-Virtual-Platform:/home/server/code# ll
总计 128
drwxrwxr-x  2 server server  4096  5月  22 14:39 ./
drwxr-x--- 16 server server  4096  5月  22 12:36 ../
-rw-------  1 server server     0  5月  22 14:12 empty.txt
-rwxr-xr-x  1 root   root   27688  5月  22 14:05 server*
-rw-------  1 server server  6143  5月  22 14:02 server.cpp
-rw-------  1 server server 10081  5月  22 14:11 test.docx
-rw-------  1 server server 11977  5月  22 12:42 test_image.png
-rw-------  1 server server 23548  5月  22 14:10 test.pdf
-rw-------  1 server server    18  5月  22 12:42 test.txt
-rw-rw-r--  1 server server 31640  5月  22 14:14 test.zip
root@server-VMware-Virtual-Platform:/home/server/code#
```

```
root@client-VMware-Virtual-Platform:/home/client/code# ll
总计 132
drwxrwxr-x  2 client client  4096  5月  22 14:59 ./
drwxr-x--- 16 client client  4096  5月  22 12:36 ../
-rwxr-xr-x  1 root   root   28736  5月  22 14:06 client*
-rw-------  1 client client  4283  5月  22 14:02 client.cpp
-rw-r--r--  1 root   root       0  5月  22 14:59 empty.txt.rcv
-rw-r--r--  1 root   root   10081  5月  22 14:33 test.docx.rcv
-rw-r--r--  1 root   root   11977  5月  22 14:30 test_image.png.rcv
-rw-r--r--  1 root   root   23548  5月  22 14:37 test.pdf.rcv
-rw-r--r--  1 root   root      18  5月  22 14:25 test.txt.rcv
-rw-r--r--  1 root   root   31640  5月  22 14:56 test.zip.rcv
root@client-VMware-Virtual-Platform:/home/client/code#
```

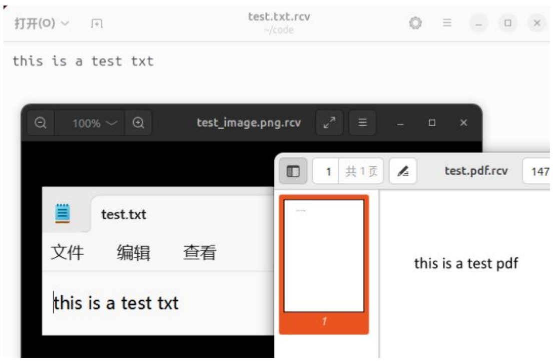
打开文件查看也是一样



this is a test txt

this is a test txt

this is a test pdf



this is a test txt

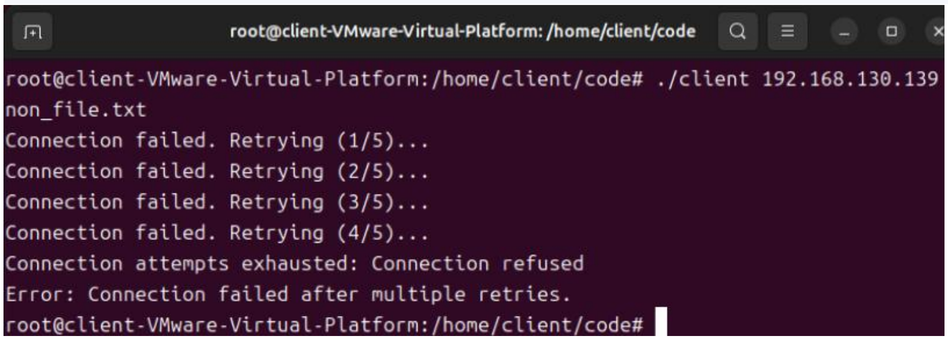
this is a test txt

this is a test pdf

6. 测试差错处理

（一）请求不存在的文件

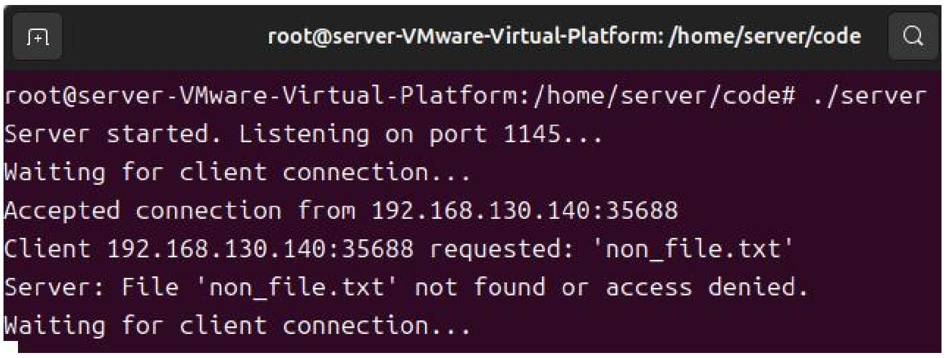
```
./client 192.168.130.139 non_file.txt
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
non_file.txt
Connection failed. Retrying (1/5)...
Connection failed. Retrying (2/5)...
Connection failed. Retrying (3/5)...
Connection failed. Retrying (4/5)...
Connection attempts exhausted: Connection refused
Error: Connection failed after multiple retries.
root@client-VMware-Virtual-Platform:/home/client/code#
```

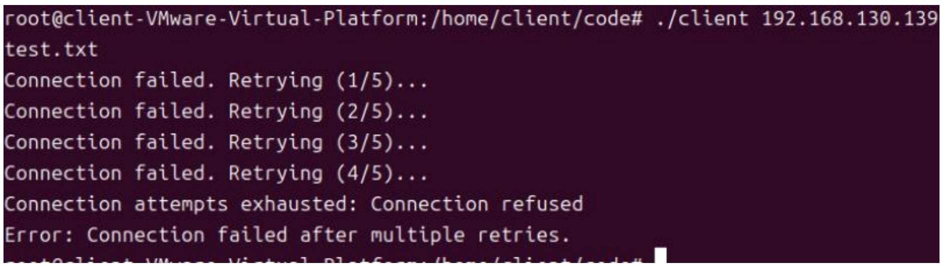
客户端请求 5 次失败后返回 Error

服务器端输出 not found



```
root@server-VMware-Virtual-Platform:/home/server/code# ./server
Server started. Listening on port 1145...
Waiting for client connection...
Accepted connection from 192.168.130.140:35688
Client 192.168.130.140:35688 requested: 'non_file.txt'
Server: File 'non_file.txt' not found or access denied.
Waiting for client connection...
```

（二）服务器未启动时客户端尝试连接

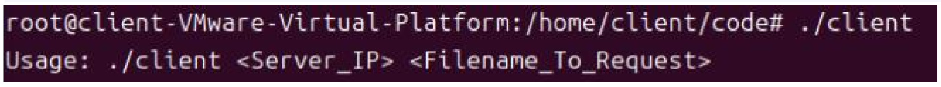


```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.139
test.txt
Connection failed. Retrying (1/5)...
Connection failed. Retrying (2/5)...
Connection failed. Retrying (3/5)...
Connection failed. Retrying (4/5)...
Connection attempts exhausted: Connection refused
Error: Connection failed after multiple retries.
```

客户端请求 5 次失败后返回 Error

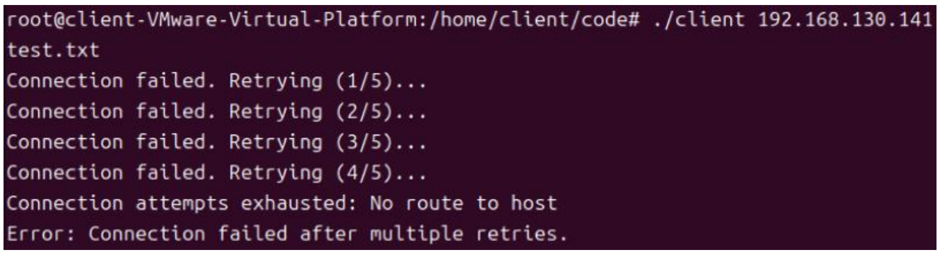
（三）客户端参数错误

```
./client
```



```
root@client-VMware-Virtual-Platform:/home/client/code# ./client
Usage: ./client <Server_IP> <Filename_To_Request>
```

（四）客户端连接不存在的服务器 IP

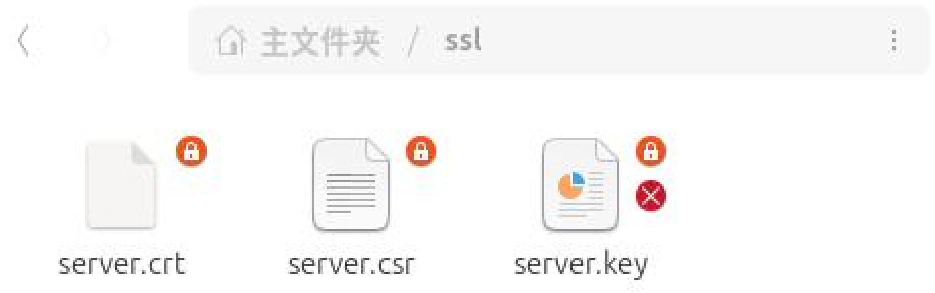


```
root@client-VMware-Virtual-Platform:/home/client/code# ./client 192.168.130.141
test.txt
Connection failed. Retrying (1/5)...
Connection failed. Retrying (2/5)...
Connection failed. Retrying (3/5)...
Connection failed. Retrying (4/5)...
Connection attempts exhausted: No route to host
Error: Connection failed after multiple retries.
```

7. ssl 实验

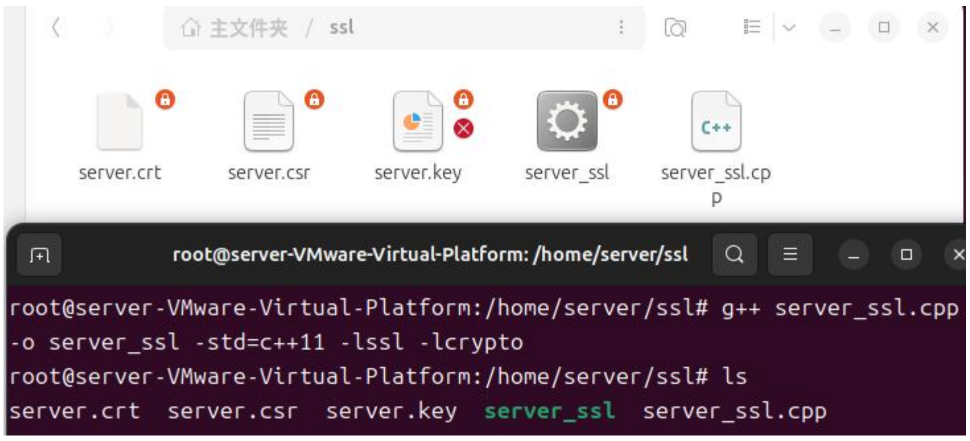
（一）生成服务器私钥（key），创建证书签名请求（csr），生成自签名证书（crt）

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

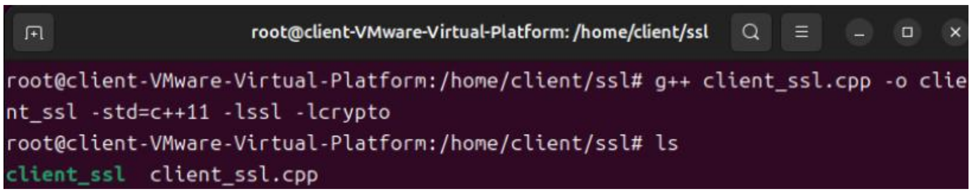


（二）编译

```
g++ server_ssl.cpp -o server_ssl -std=c++11 -lssl -lcrypto
g++ client_ssl.cpp -o client_ssl -std=c++11 -lssl -lcrypto
```



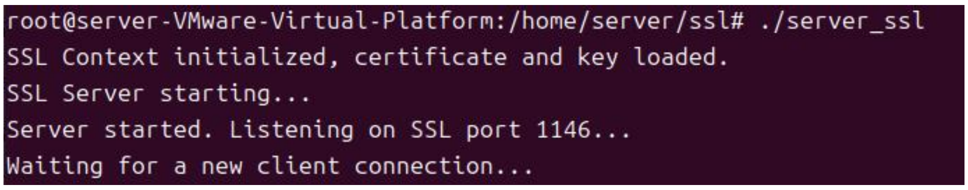
```
root@server-VMware-Virtual-Platform:/home/server/ssl# g++ server_ssl.cpp
-o server_ssl -std=c++11 -lssl -lcrypto
root@server-VMware-Virtual-Platform:/home/server/ssl# ls
server.crt  server.csr  server.key  server_ssl  server_ssl.cpp
```



root@client-VMware-Virtual-Platform: /home/client/ssl

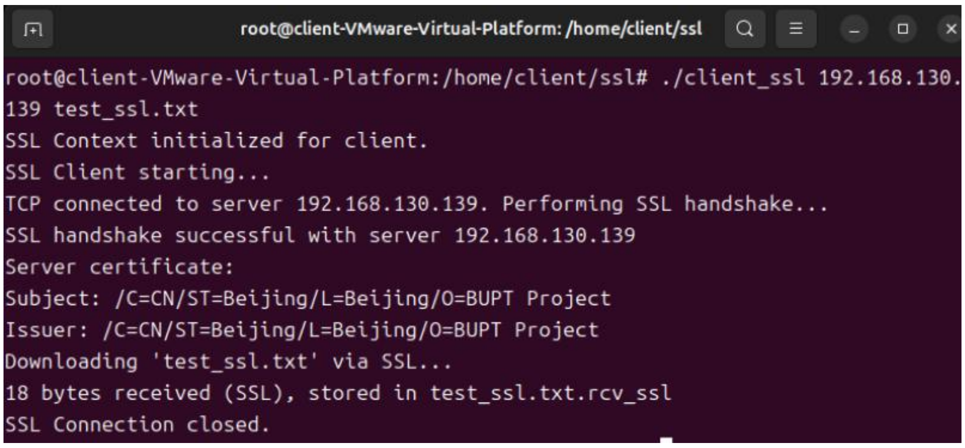
```
root@client-VMware-Virtual-Platform:/home/client/ssl# g++ client_ssl.cpp -o clie
nt_ssl -std=c++11 -lssl -lcrypto
root@client-VMware-Virtual-Platform:/home/client/ssl# ls
client_ssl  client_ssl.cpp
```

（三）启动服务



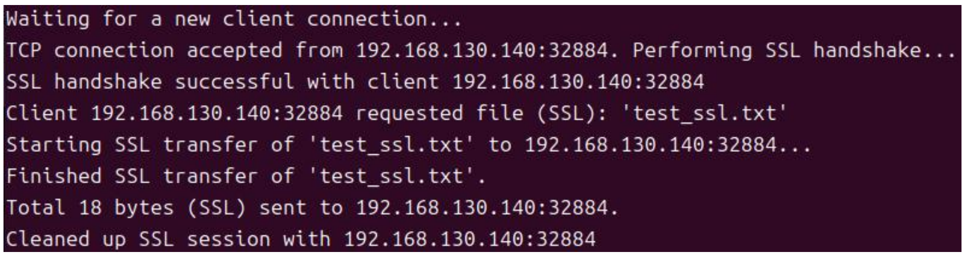
```
root@server-VMware-Virtual-Platform:/home/server/ssl# ./server_ssl
SSL Context initialized, certificate and key loaded.
SSL Server starting...
Server started. Listening on SSL port 1146...
Waiting for a new client connection...
```

（四）客户端请求 txt 文件



root@client-VMware-Virtual-Platform: /home/client/ssl

```
root@client-VMware-Virtual-Platform:/home/client/ssl# ./client_ssl 192.168.130.
139 test_ssl.txt
SSL Context initialized for client.
SSL Client starting...
TCP connected to server 192.168.130.139. Performing SSL handshake...
SSL handshake successful with server 192.168.130.139
Server certificate:
Subject: /C=CN/ST=Beijing/L=Beijing/O=BUPT Project
Issuer: /C=CN/ST=Beijing/L=Beijing/O=BUPT Project
Downloading 'test_ssl.txt' via SSL...
18 bytes received (SSL), stored in test_ssl.txt.rcv_ssl
SSL Connection closed.
```



```
Waiting for a new client connection...
TCP connection accepted from 192.168.130.140:32884. Performing SSL handshake...
SSL handshake successful with client 192.168.130.140:32884
Client 192.168.130.140:32884 requested file (SSL): 'test_ssl.txt'
Starting SSL transfer of 'test_ssl.txt' to 192.168.130.140:32884...
Finished SSL transfer of 'test_ssl.txt'.
Total 18 bytes (SSL) sent to 192.168.130.140:32884.
Cleaned up SSL session with 192.168.130.140:32884
```

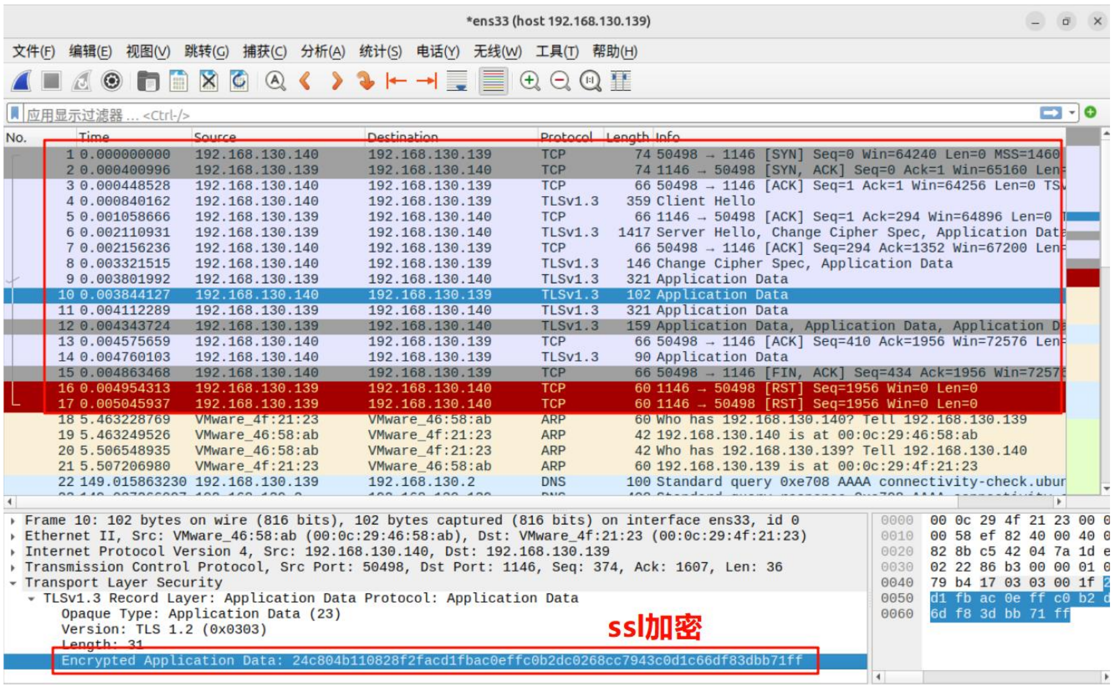
（五）wireshark 抓包

TCP 三次握手(数据包 1-3)：基础连接

TLSv1.3 握手(数据包 4-8)：安全通道建立

加密数据传输(数据包 9-14)

连接关闭方式(数据包 15-17)


*ens33 (host 192.168.130.139)
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
应用显示过滤器 … <Ctrl-/>
ssl加密

```
No.  Time           Source            Destination       Protocol Length Info
   1 0.000000000    192.168.130.140   192.168.130.139   TCP        74 50498 → 1146 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
   2 0.000400996    192.168.130.139   192.168.130.140   TCP        74 1146 → 50498 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len
   3 0.000448528    192.168.130.140   192.168.130.139   TCP        66 50498 → 1146 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TS
   4 0.000840162    192.168.130.140   192.168.130.139   TLSv1.3   359 Client Hello
   5 0.001058666    192.168.130.139   192.168.130.140   TCP        66 1146 → 50498 [ACK] Seq=1 Ack=294 Win=64896 Len=0
   6 0.002110931    192.168.130.139   192.168.130.140   TLSv1.3  1417 Server Hello, Change Cipher Spec, Application Dat
   7 0.002156236    192.168.130.140   192.168.130.139   TCP        66 50498 → 1146 [ACK] Seq=294 Ack=1352 Win=67200 Len
   8 0.003321515    192.168.130.140   192.168.130.139   TLSv1.3   146 Change Cipher Spec, Application Data
   9 0.003801992    192.168.130.139   192.168.130.140   TLSv1.3   321 Application Data
  10 0.003844127    192.168.130.140   192.168.130.139   TLSv1.3   102 Application Data
  11 0.004112289    192.168.130.139   192.168.130.140   TLSv1.3   321 Application Data
  12 0.004343724    192.168.130.139   192.168.130.140   TLSv1.3   159 Application Data, Application Data, Application D
  13 0.004575659    192.168.130.140   192.168.130.139   TCP        66 50498 → 1146 [ACK] Seq=410 Ack=1956 Win=72576 Len
  14 0.004760103    192.168.130.140   192.168.130.139   TLSv1.3    90 Application Data
  15 0.004863468    192.168.130.140   192.168.130.139   TCP        66 50498 → 1146 [FIN, ACK] Seq=434 Ack=1956 Win=7257
  16 0.004954313    192.168.130.139   192.168.130.140   TCP        60 1146 → 50498 [RST] Seq=1956 Win=0 Len=0
  17 0.005045937    192.168.130.139   192.168.130.140   TCP        60 1146 → 50498 [RST] Seq=1956 Win=0 Len=0
  18 5.463228769    VMware_4f:21:23   VMware_46:58:ab   ARP        60 Who has 192.168.130.140? Tell 192.168.130.139
  19 5.463249526    VMware_46:58:ab   VMware_4f:21:23   ARP        42 192.168.130.140 is at 00:0c:29:46:58:ab
  20 5.506548935    VMware_46:58:ab   VMware_4f:21:23   ARP        42 Who has 192.168.130.139? Tell 192.168.130.140
  21 5.507206980    VMware_4f:21:23   VMware_46:58:ab   ARP        60 192.168.130.139 is at 00:0c:29:4f:21:23
  22 149.015863230  192.168.130.139   192.168.130.2     DNS       100 Standard query 0xe708 AAAA connectivity-check.ubur
```

```
▸ Frame 10: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface ens33, id 0
▸ Ethernet II, Src: VMware_46:58:ab (00:0c:29:46:58:ab), Dst: VMware_4f:21:23 (00:0c:29:4f:21:23)
▸ Internet Protocol Version 4, Src: 192.168.130.140, Dst: 192.168.130.139
▸ Transmission Control Protocol, Src Port: 50498, Dst Port: 1146, Seq: 374, Ack: 1607, Len: 36
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 31
      Encrypted Application Data: 24c804b110828f2facd1fbac0effc0b2dc0268cc7943c0d1c66df83dbb71ff
```

```
0000  00 0c 29 4f 21 23 00 0
0010  00 58 ef 82 40 00 40 0
0020  82 8b c5 42 04 7a 1d e
0030  02 22 86 b3 00 00 01 0
0040  79 b4 17 03 03 00 1f 2
0050  d1 fb ac 0e ff c0 b2 d
0060  6d f8 3d bb 71 ff
```

五、实验总结与心得体会

本次实验过程共花费两天的时间，总体上并未遇到大问题。通过本次实验，我们掌握了 socket 等核心 API 构建客户端与服务器通信的流程，学习了设计应用层协议处理数据传输与基本异常。SSL 实现让我加深了对 TLS 握手与加密的认识。理论上，加深了对 TCP 可靠传输、端口、网络字节序等概念的理解。实践中，通过协作提升了模块化设计、编码和排错能力。

六、源代码

见附录：源程序清单

```
// client.cpp

#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

const int SERVER_PORT = 1145;
const int MAX_CONNECT_RETRY = 5;
const size_t BUFFER_SIZE = 1024;

// 连接服务器，失败则重试
int connect_with_retry(int sock_fd, sockaddr_in *server_addr) {
    int attempts = 0;
    while (connect(sock_fd, (sockaddr *)server_addr, sizeof(*server_ad
dr)) < 0) {
        attempts++;
        if (attempts >= MAX_CONNECT_RETRY) {
            perror("Connection attempts exhausted");
            return -1;
        }
        std::cout << "Connection failed. Retrying (" << attempts << "/"
 << MAX_CONNECT_RETRY << ")..." << std::endl;
        sleep(1);
    }
    return 0;
}

int main(int argc, char *argv[]) {
    if (argc != 3) { // 需要 Server_IP 和 Filename
        std::cerr << "Usage: " << argv[0] << " <Server_IP> <Filename_To
_Request>" << std::endl;
        return EXIT_FAILURE;
    }

    const char *server_ip_str = argv[1];
    const char *filename_to_request = argv[2];
    std::string received_filename = std::string(filename_to_request) +
 ".rcv";

    int sock_fd = -1;

    try {
        sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); // 创建 TC
P socket
        if (sock_fd < 0) {
            throw std::runtime_error("Error: Failed to create socket.")
;
        }

        sockaddr_in server_addr = {0};
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(SERVER_PORT);
        if (inet_pton(AF_INET, server_ip_str, &server_addr.sin_addr) <
= 0) { // IP地址转换
            throw std::runtime_error("Error: Invalid server IP address.
");
        }

        if (connect_with_retry(sock_fd, &server_addr) < 0) { // 连接服务
器
            throw std::runtime_error("Error: Connection failed after mu
ltiple retries.");
        }
        std::cout << "Connected to server " << server_ip_str << std::en
dl;

        if (send(sock_fd, filename_to_request, strlen(filename_to_requ
est) + 1, 0) < 0) { // 发送文件名
            throw std::runtime_error("Error: Failed to send filename to
 server.");
        }

        char file_exists_flag; // 接收文件存在状态
        ssize_t bytes_received_flag = recv(sock_fd, &file_exists_flag,
 sizeof(file_exists_flag), 0);
        if (bytes_received_flag <= 0) {
             throw std::runtime_error(bytes_received_flag < 0 ? "Error:
 Failed to receive file status." : "Error: Server closed connection (fi
le status).");
        }
        if (file_exists_flag == 0) {
            std::cerr << "Server error: File '" << filename_to_request
<< "' not found or access denied." << std::endl;
            close(sock_fd);
            return EXIT_FAILURE;
        }
        std::cout << "Downloading '" << filename_to_request << "'..." <
< std::endl;

        std::ofstream fout(received_filename.c_str(), std::ios::binary
 | std::ios::out); // 二进制写模式打开文件
        if (!fout.is_open()) {
            throw std::runtime_error("Error: Failed to create local fil
e: " + received_filename);
        }

        char buffer[BUFFER_SIZE];
        ssize_t bytes_received_this_iteration;
        long total_bytes_received = 0;
        while ((bytes_received_this_iteration = recv(sock_fd, buffer,
BUFFER_SIZE, 0)) > 0) { // 接收文件内容
            fout.write(buffer, bytes_received_this_iteration);
            if (fout.fail()) {
                throw std::runtime_error("Error: Failed to write to lo
cal file.");
            }
            total_bytes_received += bytes_received_this_iteration;
        }
        if (bytes_received_this_iteration < 0) { // recv 错误
            fout.close();
            remove(received_filename.c_str());
            throw std::runtime_error("Error: Receiving file data failed
.");
        }
        fout.close();
        std::cout << total_bytes_received << " bytes received, stored i
n " << received_filename << std::endl;

    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        if (sock_fd >= 0) close(sock_fd);
        return EXIT_FAILURE;
    }

    if (sock_fd >= 0) close(sock_fd);
    std::cout << "Connection closed." << std::endl;
    return EXIT_SUCCESS;
}

// server.cpp

#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <climits>

const int SERVER_PORT = 1145;
const size_t BUFFER_SIZE = 1024;
const int LISTEN_BACKLOG = 10;

// 处理客户端会话
void handle_client_session(int client_sock_fd, const std::string& clie
nt_ip, int client_port) {
    char filename_buffer[256] = {0};
    char file_content_buffer[BUFFER_SIZE];
    char file_exists_flag = 0;

    try {
        ssize_t bytes_received_filename = recv(client_sock_fd, filenam
e_buffer, sizeof(filename_buffer) - 1, 0); // 接收文件名
        if (bytes_received_filename <= 0) {
            std::cout << "Client " << client_ip << ":" << client_port <<
 (bytes_received_filename == 0 ? " closed connection early." : " recv f
ilename error.") << std::endl;
            close(client_sock_fd);
            return;
        }
        // filename_buffer[bytes_received_filename] = '\0'; // recv sho
uld have null-terminated or we rely on memset
        std::string requested_filename(filename_buffer);
        std::cout << "Client " << client_ip << ":" << client_port << " r
equested: '" << requested_filename << "'" << std::endl;

        std::ifstream fin(requested_filename.c_str(), std::ios::binary
 | std::ios::in); // 二进制读模式打开文件
        if (fin.is_open()) {
            file_exists_flag = 1;
        } else {
            std::cerr << "Server: File '" << requested_filename << "' n
ot found or access denied." << std::endl;
        }

        if (send(client_sock_fd, &file_exists_flag, sizeof(file_exists
_flag), 0) < 0) { // 发送文件存在状态
            if (fin.is_open()) fin.close();
            throw std::runtime_error("Error: Failed to send file status
 to client.");
        }

        if (file_exists_flag == 0) {
            close(client_sock_fd);
            return;
        }

        std::cout << "Transferring '" << requested_filename << "' to "
<< client_ip << ":" << client_port << std::endl;
        long total_bytes_sent = 0;
        while (fin.good()) { // 发送文件内容
            fin.read(file_content_buffer, BUFFER_SIZE);
            std::streamsize bytes_read_from_file = fin.gcount();

            if (bytes_read_from_file > 0) {
                char *ptr_to_send = file_content_buffer;
                std::streamsize bytes_left_to_send = bytes_read_from_fi
le;

                ssize_t bytes_sent_this_iteration;
                while (bytes_left_to_send > 0) { // 确保数据块完整发送
                    bytes_sent_this_iteration = send(client_sock_fd, pt
r_to_send, bytes_left_to_send, 0);
                    if (bytes_sent_this_iteration < 0) {
                        throw std::runtime_error("Error: Sending file da
ta failed.");
                    }
                    ptr_to_send += bytes_sent_this_iteration;
                    bytes_left_to_send -= bytes_sent_this_iteration;
                    total_bytes_sent += bytes_sent_this_iteration;
                }
            }
            if (fin.eof()) break;
            if (fin.fail() && !fin.eof()) {
                 throw std::runtime_error("Error: Reading from file fai
led.");
            }
        }
        if (fin.is_open()) fin.close();
        std::cout << "Sent " << total_bytes_sent << " bytes for '" << re
quested_filename << "' to " << client_ip << ":" << client_port << "." <
< std::endl;

    } catch (const std::exception& e) {
        std::cerr << "Exception in session (" << client_ip << ":" << cl
ient_port << "): " << e.what() << std::endl;
    }
    close(client_sock_fd); // 关闭客户端连接
    std::cout << "Closed connection with " << client_ip << ":" << clien
t_port << "." << std::endl;
}

int main(int argc, char* argv[]) {
    int server_sock_fd = -1;
    try {
        server_sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //
创建服务器 TCP socket
        if (server_sock_fd < 0) {
            throw std::runtime_error("Error: Failed to create server so
cket.");
        }

        int opt = 1; // 允许地址重用
        if (setsockopt(server_sock_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
 sizeof(opt)) < 0) {
            perror("Warning: setsockopt(SO_REUSEADDR) failed");
        }

        sockaddr_in server_addr = {0};
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 监听所有接口
        server_addr.sin_port = htons(SERVER_PORT);

        if (bind(server_sock_fd, (sockaddr*)&server_addr, sizeof(serve
r_addr)) < 0) { // 绑定端口
            throw std::runtime_error("Error: Port binding failed.");
        }
        if (listen(server_sock_fd, LISTEN_BACKLOG) < 0) { // 开始监听
            throw std::runtime_error("Error: Listen failed.");
        }
        std::cout << "Server started. Listening on port " << SERVER_POR
T << "..." << std::endl;

        while (true) { // 循环接受客户端连接
            sockaddr_in client_addr = {0};
            socklen_t client_addr_len = sizeof(client_addr);
            std::cout << "Waiting for client connection..." << std::end
l;
            int client_sock_fd = accept(server_sock_fd, (sockaddr*)&cli
ent_addr, &client_addr_len);
            if (client_sock_fd < 0) {
                perror("Error: Accept failed");
                continue;
            }

            char client_ip_str[INET_ADDRSTRLEN]; // 获取客户端IP和端口
            inet_ntop(AF_INET, &client_addr.sin_addr, client_ip_str, s
izeof(client_ip_str));
            int client_port = ntohs(client_addr.sin_port);
            std::cout << "Accepted connection from " << client_ip_str <
< ":" << client_port << std::endl;

            handle_client_session(client_sock_fd, std::string(client_i
p_str), client_port);
        }
    } catch (const std::exception& e) {
        std::cerr << "Server critical error: " << e.what() << std::endl
;
        if (server_sock_fd >= 0) close(server_sock_fd);
        return EXIT_FAILURE;
    }
    if (server_sock_fd >= 0) close(server_sock_fd); // 一般不会执行到
    return EXIT_SUCCESS;
}


// server_ssl.cpp

#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <climits>
```

```
// OpenSSL headers
#include <openssl/ssl.h>
#include <openssl/err.h>

const int SSL_SERVER_PORT = 1146; // 新的 SSL 端口
const size_t BUFFER_SIZE = 1024;
const int LISTEN_BACKLOG = 10;

// SSL 上下文对象，全局或传递给需要它的函数
SSL_CTX *ssl_ctx = nullptr;

// 初始化服务器端 SSL 上下文
bool initialize_server_ssl_ctx() {
    SSL_library_init();         // 初始化SSL库
    OpenSSL_add_all_algorithms(); // 加载算法
    SSL_load_error_strings();     // 加载错误字符串

    ssl_ctx = SSL_CTX_new(TLS_server_method()); // 创建新的CTX，使用TLS作
为协议
    if (!ssl_ctx) {
        ERR_print_errors_fp(stderr);
        return false;
    }

    // 加载服务器证书 ("server.crt")
    if (SSL_CTX_use_certificate_file(ssl_ctx, "server.crt", SSL_FILETY
PE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        SSL_CTX_free(ssl_ctx);
        ssl_ctx = nullptr;
        return false;
    }

    // 加载服务器私钥 ("server.key")
    if (SSL_CTX_use_PrivateKey_file(ssl_ctx, "server.key", SSL_FILETYP
E_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        SSL_CTX_free(ssl_ctx);
        ssl_ctx = nullptr;
        return false;
    }

    // 检查私钥是否与证书匹配
    if (!SSL_CTX_check_private_key(ssl_ctx)) {
        std::cerr << "Private key does not match the public certificate
" << std::endl;
        SSL_CTX_free(ssl_ctx);
        ssl_ctx = nullptr;
        return false;
    }
    std::cout << "SSL Context initialized, certificate and key loaded."
 << std::endl;
    return true;
}

// 处理客户端会话 (SSL 版本)
void handle_client_session_ssl(int client_sock_fd, SSL* ssl, const std
::string& client_ip, int client_port) {
    char filename_buffer[256] = {0};
    char file_content_buffer[BUFFER_SIZE];
    char file_exists_flag = 0;

    try {
        // 1. 接收文件名 (使用 SSL_read)
        ssize_t bytes_received_filename = SSL_read(ssl, filename_buffe
r, sizeof(filename_buffer) - 1);
        if (bytes_received_filename <= 0) {
            int ssl_error = SSL_get_error(ssl, bytes_received_filename)
;
            if (ssl_error == SSL_ERROR_ZERO_RETURN) {
                 std::cout << "Client " << client_ip << ":" << client_po
rt << " closed SSL connection gracefully before sending filename." << s
td::endl;
            } else {
                std::cerr << "SSL_read (filename) error or connection c
losed by client " << client_ip << ":" << client_port << std::endl;
                ERR_print_errors_fp(stderr);
            }
            return; // SSL_free 和 close 会在调用者中处理
        }
        // filename_buffer[bytes_received_filename] = '\0'; // SSL_rea
d 不保证空终止
        std::string requested_filename(filename_buffer, bytes_received
_filename); // 使用接收到的长度构造string
        std::cout << "Client " << client_ip << ":" << client_port << " r
equested file (SSL): '" << requested_filename << "'" << std::endl;

        // 2. 打开文件 (二进制读)
        std::ifstream fin(requested_filename.c_str(), std::ios::binary
 | std::ios::in);
        if (fin.is_open()) {
            file_exists_flag = 1;
        } else {
            std::cerr << "Server: File '" << requested_filename << "' n
ot found or access denied." << std::endl;
        }

        // 3. 发送文件状态确认 (使用 SSL_write)
        if (SSL_write(ssl, &file_exists_flag, sizeof(file_exists_flag)
) <= 0) {
            if (fin.is_open()) fin.close();
            throw std::runtime_error("Error: SSL_write (file status) fa
iled.");
        }
        if (file_exists_flag == 0) {
            std::cout << "Sent 'file not found' (SSL) for '"<< requeste
d_filename <<"' to client " << client_ip << ":" << client_port << std::
endl;
            // fin 已关闭或未打开
            return;
        }

        // 4. 发送文件内容 (使用 SSL_write)
        std::cout << "Starting SSL transfer of '" << requested_filename
 << "' to " << client_ip << ":" << client_port << "..." << std::endl;
        long total_bytes_sent = 0;
        while (fin.good()) {
            fin.read(file_content_buffer, BUFFER_SIZE);
            std::streamsize bytes_read_from_file = fin.gcount();

            if (bytes_read_from_file > 0) {
                char *ptr_to_send = file_content_buffer;
                std::streamsize bytes_left_to_send = bytes_read_from_fi
le;
                ssize_t bytes_sent_this_iteration;
                while (bytes_left_to_send > 0) {
                    bytes_sent_this_iteration = SSL_write(ssl, ptr_to_s
end, bytes_left_to_send);
                    if (bytes_sent_this_iteration <= 0) {
                        int ssl_error = SSL_get_error(ssl, bytes_sent_th
is_iteration);
                        ERR_print_errors_fp(stderr);
                        if (fin.is_open()) fin.close();
                        throw std::runtime_error("Error: SSL_write (fil
e data) failed. SSL Error: " + std::to_string(ssl_error));
                    }
                    ptr_to_send += bytes_sent_this_iteration;
                    bytes_left_to_send -= bytes_sent_this_iteration;
                    total_bytes_sent += bytes_sent_this_iteration;
                }
            }
            if (fin.eof()) break;
            if (fin.fail() && !fin.eof()) {
                 if (fin.is_open()) fin.close();
                 throw std::runtime_error("Error: Reading from file fai
led during transfer.");
            }
        }
        if (fin.is_open()) fin.close();
        std::cout << "Finished SSL transfer of '" << requested_filename
 << "'." << std::endl;
        std::cout << "Total " << total_bytes_sent << " bytes (SSL) sent
 to " << client_ip << ":" << client_port << "." << std::endl;

    } catch (const std::exception& e) {
        std::cerr << "Exception in SSL client session (" << client_ip <
< ":" << client_port << "): " << e.what() << std::endl;
    }
    // SSL_free 和 close 会在调用此函数的地方处理
}

int main(int argc, char* argv[]) {
    int server_sock_fd = -1;
    int client_sock_fd = -1;
    SSL *ssl = nullptr;

    // 初始化 SSL 上下文
    if (!initialize_server_ssl_ctx()) {
        std::cerr << "Failed to initialize SSL context. Exiting." << st
d::endl;
        return EXIT_FAILURE;
    }

    try {
        std::cout << "SSL Server starting..." << std::endl;
        server_sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (server_sock_fd < 0) {
            throw std::runtime_error("Error: Failed to create server so
cket.");
        }

        int opt = 1;
        if (setsockopt(server_sock_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
 sizeof(opt)) < 0) {
            perror("Warning: setsockopt(SO_REUSEADDR) failed");
        }

        sockaddr_in server_addr = {0};
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        server_addr.sin_port = htons(SSL_SERVER_PORT); // 使用新的SSL端
口

        if (bind(server_sock_fd, (sockaddr*)&server_addr, sizeof(serve
r_addr)) < 0) {
            throw std::runtime_error("Error: Port binding failed.");
        }
        if (listen(server_sock_fd, LISTEN_BACKLOG) < 0) {
            throw std::runtime_error("Error: Listen failed.");
        }
        std::cout << "Server started. Listening on SSL port " << SSL_SE
RVER_PORT << "..." << std::endl;

        while (true) {
            sockaddr_in client_addr = {0};
            socklen_t client_addr_len = sizeof(client_addr);

            std::cout << "Waiting for a new client connection..." << st
d::endl;
            client_sock_fd = accept(server_sock_fd, (sockaddr*)&client
_addr, &client_addr_len);
            if (client_sock_fd < 0) {
                perror("Error: Accept failed");
                continue;
            }

            char client_ip_str[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, &client_addr.sin_addr, client_ip_str, s
izeof(client_ip_str));
            int client_port = ntohs(client_addr.sin_port);
            std::cout << "TCP connection accepted from " << client_ip_s
tr << ":" << client_port << ". Performing SSL handshake..." << std::end
l;

            // 创建 SSL 对象并进行握手
            ssl = SSL_new(ssl_ctx);
            if (!ssl) {
                std::cerr << "Error creating SSL object." << std::endl;
                ERR_print_errors_fp(stderr);
                close(client_sock_fd);
                continue;
            }
            SSL_set_fd(ssl, client_sock_fd);

            if (SSL_accept(ssl) <= 0) { // SSL 握手
                std::cerr << "SSL handshake failed with client " << clie
nt_ip_str << ":" << client_port << std::endl;
                ERR_print_errors_fp(stderr);
                SSL_free(ssl);
                ssl = nullptr;
                close(client_sock_fd);
                continue;
            }
            std::cout << "SSL handshake successful with client " << cli
ent_ip_str << ":" << client_port << std::endl;

            handle_client_session_ssl(client_sock_fd, ssl, std::string
(client_ip_str), client_port);

            // 清理 SSL 会话资源
            SSL_shutdown(ssl); // 优雅关闭SSL层
            SSL_free(ssl);
            ssl = nullptr;
            close(client_sock_fd); // 关闭TCP socket
            std::cout << "Cleaned up SSL session with " << client_ip_st
r << ":" << client_port << std::endl;
        }

    } catch (const std::exception& e) {
        std::cerr << "Server critical error: " << e.what() << std::endl
;
        if (ssl) SSL_free(ssl);
        if (client_sock_fd >= 0) close(client_sock_fd);
        if (server_sock_fd >= 0) close(server_sock_fd);
        if (ssl_ctx) SSL_CTX_free(ssl_ctx);
        EVP_cleanup(); // 清理OpenSSL资源
        return EXIT_FAILURE;
    }

    if (server_sock_fd >= 0) close(server_sock_fd);
    if (ssl_ctx) SSL_CTX_free(ssl_ctx);
    EVP_cleanup();
    std::cout << "Server shutting down." << std::endl;
    return EXIT_SUCCESS;
}

// client_ssl.cpp

#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

// OpenSSL headers
#include <openssl/ssl.h>
#include <openssl/err.h>

const int SSL_SERVER_PORT = 1146; // SSL 服务器端口号
const int MAX_CONNECT_RETRY = 5;
const size_t BUFFER_SIZE = 1024;

// SSL 上下文对象
SSL_CTX *ssl_ctx = nullptr;

// 初始化客户端 SSL 上下文
bool initialize_client_ssl_ctx() {
    SSL_library_init();
    OpenSSL_add_all_algorithms();
    SSL_load_error_strings();

    ssl_ctx = SSL_CTX_new(TLS_client_method());
    if (!ssl_ctx) {
        ERR_print_errors_fp(stderr);
        return false;
    }
    // 对于自签名证书，客户端通常需要特殊配置才能验证或选择跳过验证。
    // 为了实验简单，我们这里跳过服务器证书验证。
    // **警告：在生产环境中，务必验证服务器证书！**
    // SSL_CTX_set_verify(ssl_ctx, SSL_VERIFY_PEER, NULL); // 开启验证
    // SSL_CTX_load_verify_locations(ssl_ctx, "ca.crt", NULL); // 加载C
A证书
```

```
    SSL_CTX_set_verify(ssl_ctx, SSL_VERIFY_NONE, NULL); // 实验目的：禁用
服务器证书验证

    std::cout << "SSL Context initialized for client." << std::endl;
    return true;
}

// 带重试的 TCP 连接
int connect_tcp_with_retry(int sock_fd, sockaddr_in *server_addr) {
    int attempts = 0;
    while (connect(sock_fd, (sockaddr *)server_addr, sizeof(*server_ad
dr)) < 0) {
        attempts++;
        if (attempts >= MAX_CONNECT_RETRY) {
            perror("TCP Connection attempts exhausted");
            return -1;
        }
        std::cout << "TCP Connection failed. Retrying (" << attempts <<
 "/" << MAX_CONNECT_RETRY << ")..." << std::endl;
        sleep(1);
    }
    return 0;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <Server_IP> <Filename_To
_Request>" << std::endl;
        return EXIT_FAILURE;
    }

    const char *server_ip_str = argv[1];
    const char *filename_to_request = argv[2];
    std::string received_filename = std::string(filename_to_request) +
 ".rcv_ssl";

    int sock_fd = -1;
    SSL *ssl = nullptr;

    // 初始化 SSL 上下文
    if (!initialize_client_ssl_ctx()) {
        std::cerr << "Failed to initialize SSL context. Exiting." << st
d::endl;
        return EXIT_FAILURE;
    }

    try {
        std::cout << "SSL Client starting..." << std::endl;
        sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock_fd < 0) {
            throw std::runtime_error("Error: Failed to create socket.")
;
        }

        sockaddr_in server_addr = {0};
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(SSL_SERVER_PORT); // 使用SSL端口
        if (inet_pton(AF_INET, server_ip_str, &server_addr.sin_addr) <
= 0) {
            throw std::runtime_error("Error: Invalid server IP address.
");
        }

        // 1. TCP 连接
        if (connect_tcp_with_retry(sock_fd, &server_addr) < 0) {
            throw std::runtime_error("Error: TCP connection failed afte
r multiple retries.");
        }
        std::cout << "TCP connected to server " << server_ip_str << ".
Performing SSL handshake..." << std::endl;

        // 2. 创建 SSL 对象并进行 SSL 握手
        ssl = SSL_new(ssl_ctx);
        if (!ssl) {
            throw std::runtime_error("Error creating SSL object.");
        }
        SSL_set_fd(ssl, sock_fd);
        if (SSL_connect(ssl) <= 0) { // SSL 握手
            ERR_print_errors_fp(stderr);
            throw std::runtime_error("Error: SSL handshake failed.");
        }
        std::cout << "SSL handshake successful with server " << server_
ip_str << std::endl;

        // （可选）打印服务器证书信息
        X509 *server_cert = SSL_get_peer_certificate(ssl);
        if (server_cert) {
            std::cout << "Server certificate:" << std::endl;
            char *line = X509_NAME_oneline(X509_get_subject_name(serve
r_cert), 0, 0);
            std::cout << "Subject: " << line << std::endl;
            free(line);
            line = X509_NAME_oneline(X509_get_issuer_name(server_cert)
, 0, 0);
            std::cout << "Issuer: " << line << std::endl;
            free(line);
            X509_free(server_cert);
        } else {
            std::cout << "No server certificate presented." << std::end
l;
        }

        // 3. 发送文件名 (使用 SSL_write)
        if (SSL_write(ssl, filename_to_request, strlen(filename_to_req
uest) + 1) <= 0) {
            ERR_print_errors_fp(stderr);
            throw std::runtime_error("Error: SSL_write (filename) faile
d.");
        }

        // 4. 接收文件状态确认 (使用 SSL_read)
        char file_exists_flag;
        ssize_t bytes_received_flag = SSL_read(ssl, &file_exists_flag,
 sizeof(file_exists_flag));
        if (bytes_received_flag <= 0) {
            ERR_print_errors_fp(stderr);
            throw std::runtime_error(bytes_received_flag < 0 ? "Error:
SSL_read (file status) failed." : "Error: Server closed SSL connection
(file status).");
        }
        if (file_exists_flag == 0) {
            std::cerr << "Server error: File '" << filename_to_request
<< "' not found or access denied." << std::endl;
            // SSL_free, close等会在catch中处理
            return EXIT_FAILURE; // 或者也可以throw异常
        }
        std::cout << "Downloading '" << filename_to_request << "' via S
SL..." << std::endl;

        // 5. 接收文件内容 (使用 SSL_read)
        std::ofstream fout(received_filename.c_str(), std::ios::binary
 | std::ios::out);
        if (!fout.is_open()) {
            throw std::runtime_error("Error: Failed to create local fil
e: " + received_filename);
        }

        char buffer[BUFFER_SIZE];
        ssize_t bytes_received_this_iteration;
        long total_bytes_received = 0;
        while ((bytes_received_this_iteration = SSL_read(ssl, buffer,
BUFFER_SIZE)) > 0) {
            fout.write(buffer, bytes_received_this_iteration);
            if (fout.fail()) {
                throw std::runtime_error("Error: Failed to write to lo
cal file.");
            }
            total_bytes_received += bytes_received_this_iteration;
        }
        int ssl_read_error = SSL_get_error(ssl, bytes_received_this_it
eration);
        if (bytes_received_this_iteration < 0 && ssl_read_error != SSL_
ERROR_ZERO_RETURN ) { // SSL_read 错误且不是正常关闭
            fout.close();
            remove(received_filename.c_str());
            ERR_print_errors_fp(stderr);
            throw std::runtime_error("Error: SSL_read (file data) faile
d. SSL Error: " + std::to_string(ssl_read_error));
        }
            fout.close();
            std::cout << total_bytes_received << " bytes received (SSL), st
ored in " << received_filename << std::endl;

    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        if (ssl) { SSL_shutdown(ssl); SSL_free(ssl); }
        if (sock_fd >= 0) close(sock_fd);
        if (ssl_ctx) SSL_CTX_free(ssl_ctx);
        EVP_cleanup();
        return EXIT_FAILURE;
    }

    // 6. 关闭 SSL 和 TCP 连接
    if (ssl) { SSL_shutdown(ssl); SSL_free(ssl); } // 优雅关闭SSL层
    if (sock_fd >= 0) close(sock_fd);             // 关闭TCP socket
    if (ssl_ctx) SSL_CTX_free(ssl_ctx);           // 清理CTX
    EVP_cleanup();                               // 清理OpenSSL其他资源
    std::cout << "SSL Connection closed." << std::endl;
    return EXIT_SUCCESS;
}
```