

北京邮电大学实验报告

实验名称	计算机网络实验二	学 院	网络空间 安全学院	指导教师	杨震
班 级	班内序号	学 号	学生姓名	分工	
				代码构建、实验实施及截图（50%）	
				资料整理、流程图及报告撰写（50%）	
实 验 内 容	本实验旨在使用 Socket 编程实现一个基于 TCP 协议的文件传输系统，包含服务器端和客户端两个程序。客户端发送文件传输请求，服务器端接收请求并将指定文件传输给客户端。				
	具体包括以下内容：				
	（1）使用 Socket API 通信实现文件传输客户端和服务端 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；				
	（2）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。客户端应输出：新文件名、传输总字节数；或者差错报告；				
学 生 实 验 报 告 (附页)	（3）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）。				
	（4）可选功能：支持 SSL 安全连接。				
实 验 成 绩 评 定	《详见下方“实验报告”》				
	评语:				
	成绩:				
			指导教师签名:		
			年 月 日		

目录

实验二 基于 SOCKET 的文件传输	3
1 实验内容与实验环境	3
1.1 实验内容	3
1.2 实验目的	3
1.3 实验环境	3
2 软件设计	3
2.1 数据结构	3
2.1.1 服务器端	3
2.1.2 客户端	5
2.2 模块结构	7
2.3 算法流程	10
2.3.1 服务器端流程图	10
2.3.2 客户端流程图	12
2.3.3 完整流程图	13
2.4 主要功能模块的实现要点	14
2.4.1 网络通信模块	14
2.4.2 安全通信模块	14
2.4.3 文件传输模块	14
2.4.4 错误处理模块	14
2.4.5 用户交互模块	14
3 实验步骤	15
3.1 实验准备	15
3.2 实验一：不使用 SSL 的 TCP 文件传输	16
3.3 实验二：使用 SSL 的 TCP 文件传输	19
4 实验总结和心得体会	23
4.1 实际上机调试时间	23
4.2 调试过程中遇到的问题及解决过程	23
4.3 收获和提高	23
4.4 实验设计和安排的不足	24
附录：程序源代码	25

实验二 基于 SOCKET 的文件传输

1 实验内容与实验环境

1.1 实验内容

- 1.使用 Socket API 通信实现文件传输客户端和服务端 2 个程序，客户端发送文件传输请求，服务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；
- 2.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。客户端应输出：新文件名、传输总字节数；或者差错报告；
- 3.服务端应输出：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）。
- 4.可选功能：支持 SSL 安全连接。

1.2 实验目的

通过本实验，学生可以深入理解传输层的 TCP 协议原理以及 Socket 套接字网络通信的流程，同时还将学习如何进行网络编程中的错误处理，提升数据传输的可靠性与稳定性。

1.3 实验环境

操作系统：Ubuntu 虚拟机，Linux 系统

编程语言：Python

开发工具：Visual Studio Code

测试设备：一台联网的计算机，打开两个窗口，同时作为服务器和客户端

2 软件设计

2.1 数据结构

2.1.1 服务器端

- 1.socket.socket 对象

`listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

功能：提供基础网络通信能力

特点：创建监听套接字、支持 IPv4 地址族(AF_INET)和流式套接字(SOCK_STREAM)、设置 SO_REUSEADDR 选项允许地址重用

2.ssl.SSLContext 对象

`ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)`

功能：管理 SSL/TLS 配置

特点：使用 PROTOCOL_TLS_SERVER 协议、加载证书链(load_cert_chain)、支持服务器端 SSL 包装

3.ssl.SSLSocket 对象

`actual_conn = ssl_context.wrap_socket(conn_plain, server_side=True)`

功能：提供加密通信能力

特点：包装普通 TCP 套接字、自动处理 SSL 握手、提供加密数据传输

4.bytes 对象

`filename_request_bytes = conn.recv(BUFFER_SIZE)`

功能：处理二进制数据

特点：用于网络数据接收、支持缓冲区操作、与字符串相互转换

5.str 对象

`filename_request = filename_request_bytes.decode('utf-8')`

功能：处理文本信息

特点：用于命令解析、支持格式化输出、可转换为 bytes 用于网络传输

6.文件对象

`with open(filename_request, 'rb') as f:`

功能：文件 I/O 操作

特点：二进制读取模式、支持上下文管理(with 语句)、自动处理文件关闭

7.元组

`(host, port)`

功能：存储地址信息

特点：不可变序列、用于网络地址存储、支持解包操作

8.常量

`BUFFER_SIZE = 4096`

功能：配置参数

特点：定义缓冲区大小、控制数据块传输、影响性能参数

9.`argparse.Namespace` 对象

`args = parser.parse_args()`

功能：命令行参数解析

特点：存储解析后的参数、支持多种参数类型、提供帮助信息生成

10. `datetime` 对象

功能：时间处理

`datetime.datetime.now()`

特点：记录连接时间、格式化输出、用于日志记录

这些数据结构协同工作，实现了网络监听和连接管理、安全通信配置、文件传输处理、错误处理和日志记录、命令行参数解析、文件 I/O 操作。

程序通过灵活组合这些数据结构，既支持普通 TCP 通信，又支持 SSL 加密通信，同时保持了良好的代码结构和可维护性。服务器能够同时处理多个客户端连接，并根据配置自动选择通信模式。

2.1.2 客户端

1.`socket.socket` 对象

`plain_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

功能：提供基础网络通信能力

特点：创建原始 TCP 套接字、支持 IPv4 地址族(AF_INET)和流式套接字(SOCK_STREAM)

2.`ssl.SSLContext` 对象

`ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)`

功能：管理 SSL/TLS 配置和安全设置

特点：使用 PROTOCOL_TLS_CLIENT 协议、可配置证书验证模式(verify_mode)、可控制主机名检查(check_hostname)

3.`ssl.SSLSocket` 对象

```
actual_sock = ssl_context.wrap_socket(plain_sock, server_hostname=server_ip)
```

功能：提供加密通信能力

特点：包装普通 TCP 套接字、自动处理 SSL 握手、提供加密数据传输

4.bytes 对象

```
server_response_bytes = actual_sock.recv(BUFFER_SIZE)
```

功能：处理二进制数据

特点：用于网络数据接收、支持缓冲区操作、与字符串相互转换

5.str 对象

```
server_response = server_response_bytes.decode('utf-8')
```

功能：处理文本信息

特点：用于命令解析、支持格式化输出、可转换为 bytes 用于网络传输

6.文件对象

```
with open(new_filename, 'wb') as f:
```

功能：文件 I/O 操作

特点：二进制写入模式、支持上下文管理(with 语句)、自动处理文件关闭

7.元组

```
(server_ip, server_port)
```

功能：存储地址信息

特点：不可变序列、用于网络地址存储、支持解包操作

8.常量

```
BUFFER_SIZE = 4096
```

功能：配置参数

特点：定义缓冲区大小、控制数据块传输、影响性能参数

9 argparse.Namespace 对象

```
args = parser.parse_args()
```

功能：命令行参数解析

特点：存储解析后的参数、支持多种参数类型、提供帮助信息生成

10.datetime 对象

```
timestamp = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
```

功能：时间处理

特点：生成时间戳、格式化输出、用于文件名唯一性

这些数据结构协同工作，实现了网络连接管理（普通 TCP 和 SSL）、安全通信配置、文件传输处理、错误处理和日志记录、命令行参数解析、文件 I/O 操作。实现了一个基于 SSL 的安全文件传输客户端，能够安全地请求和接收服务器上的文件。

2.2 模块结构

1.服务器端

①创建 Socket，绑定地址，监听连接请求：

创建主监听 socket (socket.socket())、设置 socket 选项(SO_REUSEADDR)、绑定到指定主机和端口(bind())、开始监听连接(listen())

可选 SSL 配置：加载 SSL 证书和私钥(ssl.SSLContext.load_cert_chain())、将普通 socket 包装为 SSL socket (wrap_socket())

②处理客户端连接：

接收客户端连接请求(accept())、接收客户端发送的文件名请求(recv())、检查文件是否存在 (os.path.exists())、发送文件存在确认和大小信息 (sendall())、等待客户端准备确认(recv())、分块读取文件内容并发送给客户端 (open(), read(), sendall())

③错误处理和资源清理：

捕获和处理各种异常（SSL 错误、socket 错误等）、确保所有连接和文件被正确关闭(close())、响应键盘中断(KeyboardInterrupt)

2.客户端

①创建 Socket 并连接服务器：

创建普通 TCP socket (socket.socket())

可选 SSL 配置：创建 SSL 上下文(ssl.SSLContext())、禁用证书验证（仅用于测试）、将普通 socket 包装为 SSL socket(wrap_socket())、连接到服务器(connect())

②文件传输处理：

发送请求的文件名(sendall())、接收服务器响应(recv())、处理"FILE_EXISTS"响应（解析文件大小）、处理"FILE_NOT_FOUND"响应、发送准备确认(sendall())、分块接收文件内容并写入本地文件(recv(), open(), write())

③错误处理和资源清理：

捕获和处理连接错误（拒绝连接、超时等）、处理 SSL 相关错误（证书验证失败等）、确保所有连接和文件被正确关闭、生成唯一的接收文件名（包含时间戳和传输类型

3.通信协议设计

①客户端首先发送纯文件名

②服务器响应： "FILE_EXISTS [文件大小]"（成功）、"FILE_NOT_FOUND"（失败）

③客户端确认："ACK_READY"

④文件数据传输（二进制模式）。

4.安全特性

①可选 SSL/TLS 加密传输

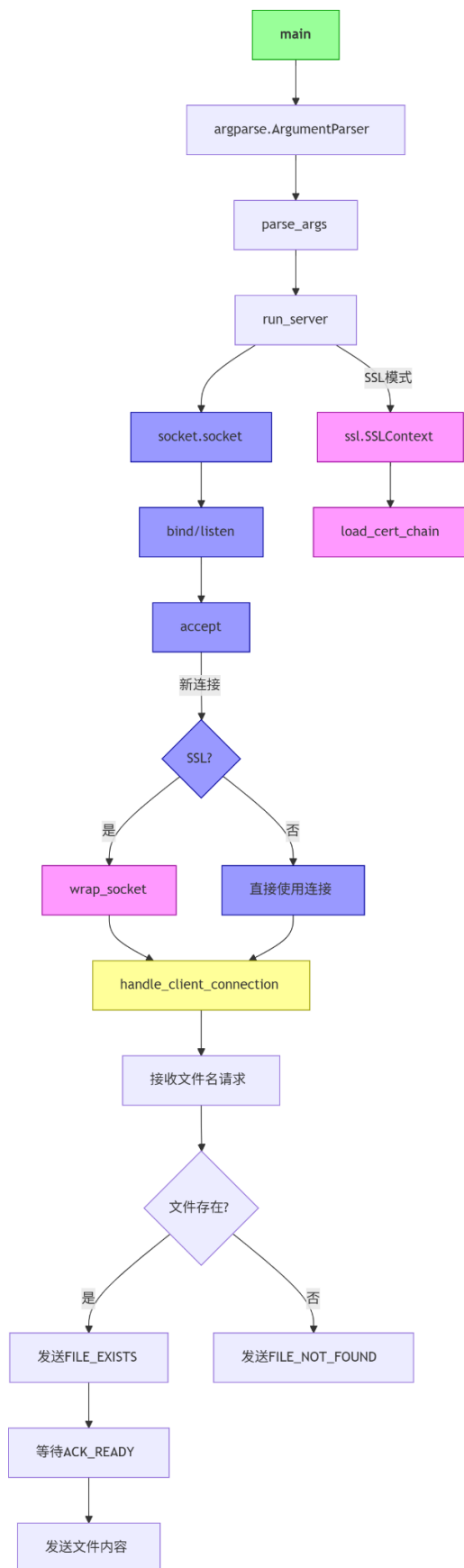
②服务器端证书验证（需预先配置）

③缓冲区大小固定为 4096 字节，所有网络操作都有超时处理

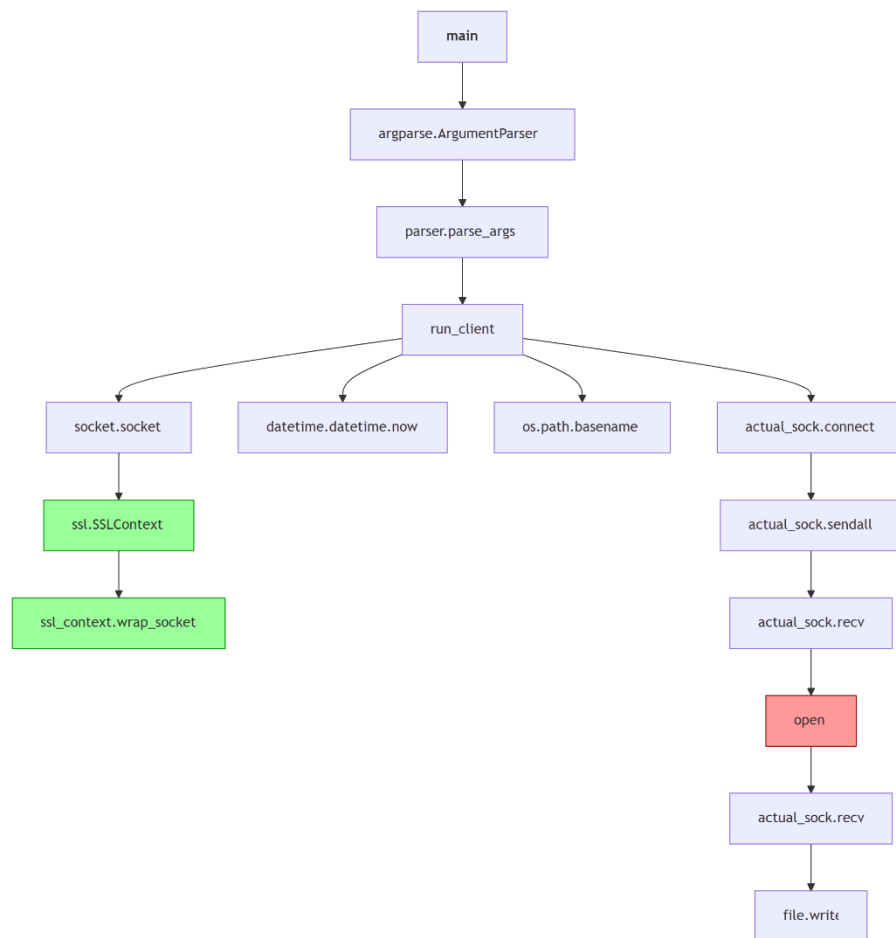
④这个模块结构清晰地展示了服务器和客户端的分工，以及它们如何处理文件传输的各个阶段，包括连接建立、数据传输和错误处理。SSL/TLS 支持作为可选模块集成在两个端中。

5.子程序之间程序调用关系图

①服务器端

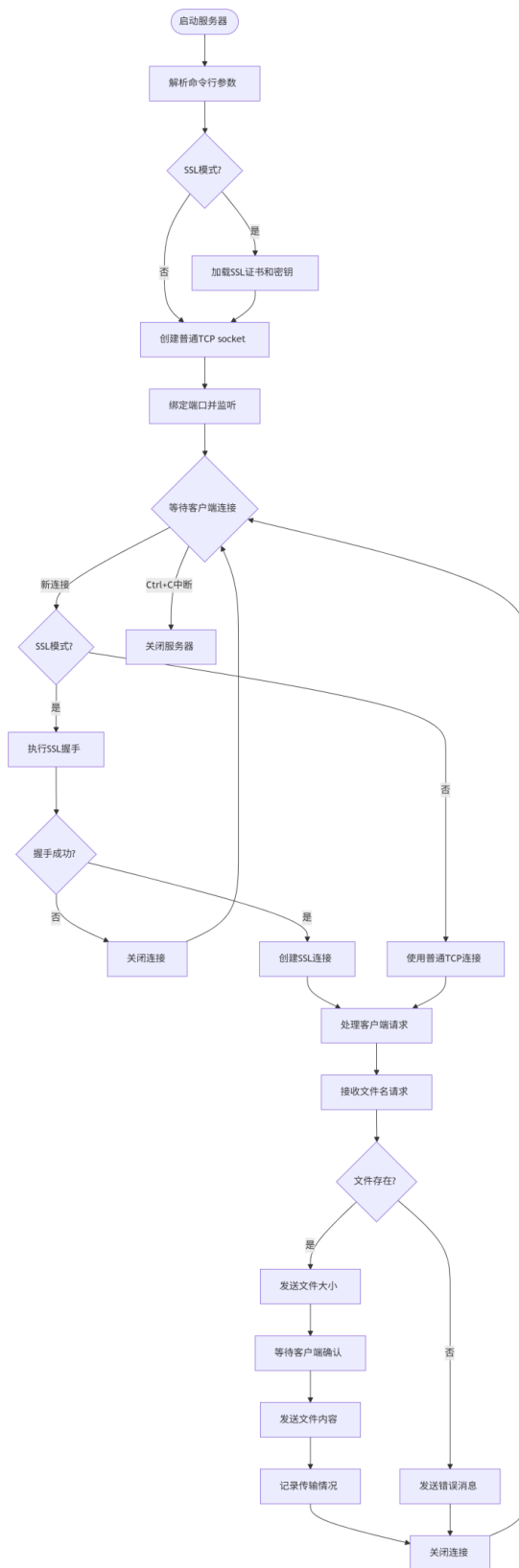


②客户端

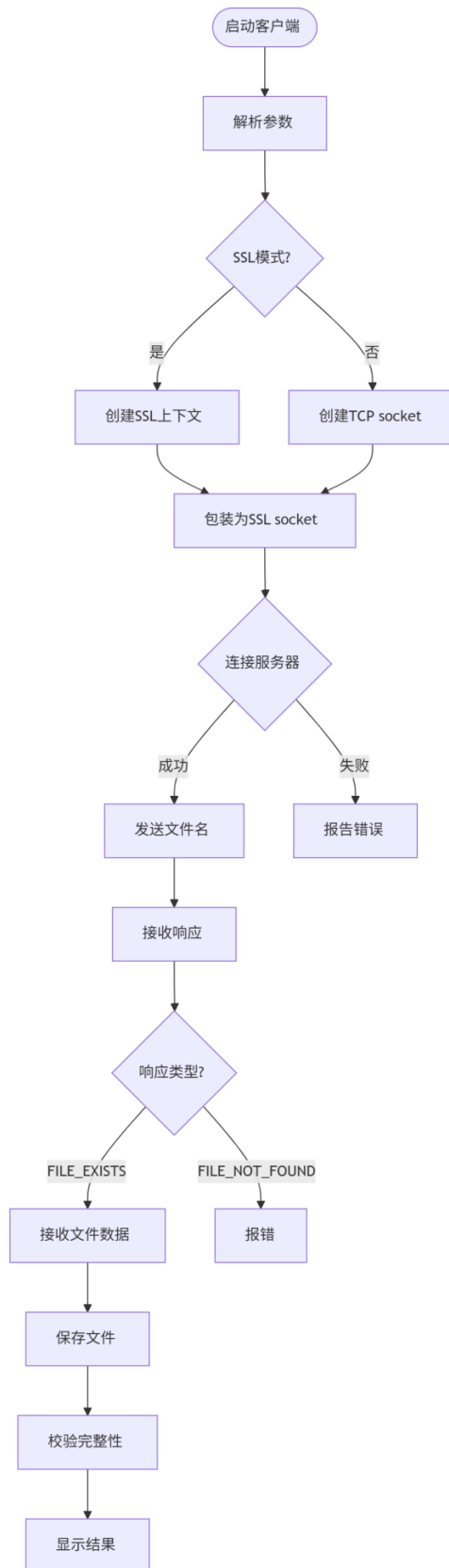


2.3 算法流程

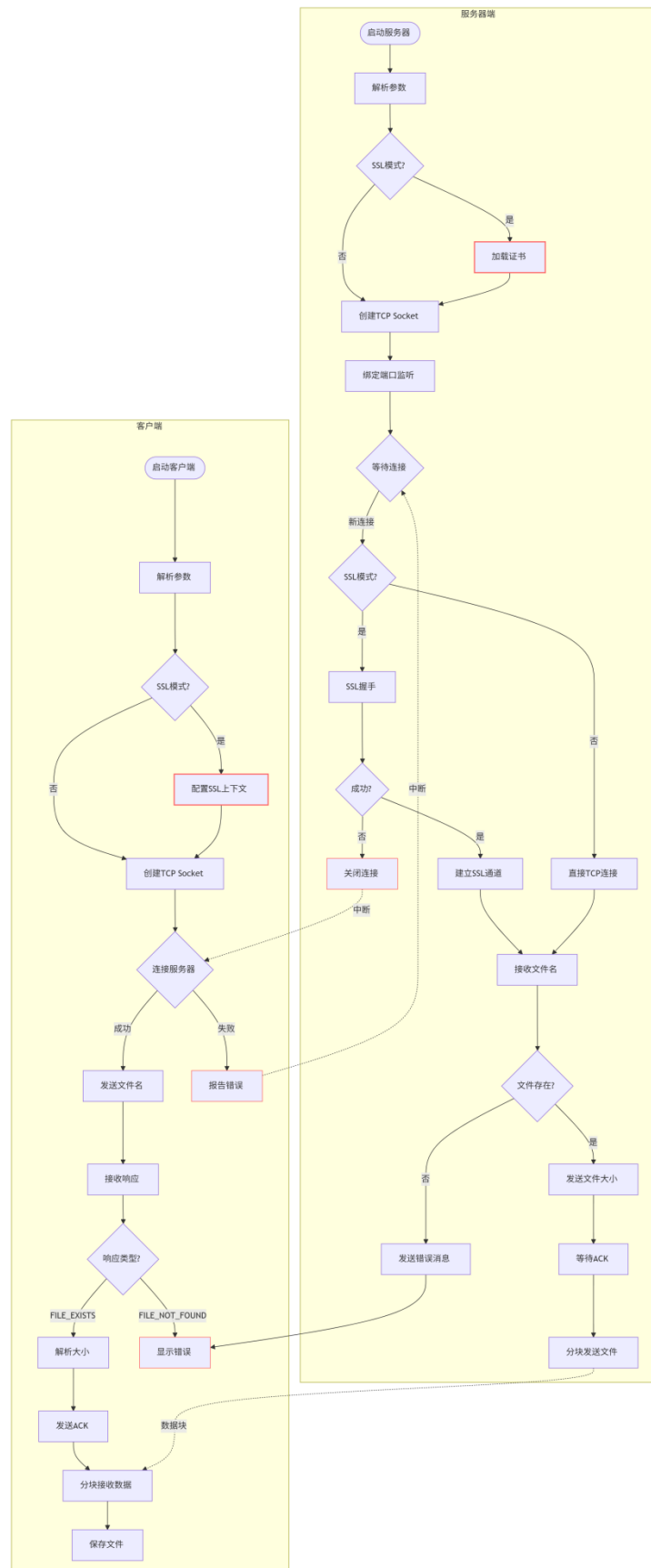
2.3.1 服务器端流程图



2.3.2 客户端流程图



2.3.3 完整流程图



2.4 主要功能模块的实现要点

2.4.1 网络通信模块

1.Socket 连接管理

服务器端创建监听 Socket，绑定指定 IP 和端口，设置最大连接数。

客户端创建 Socket 并连接服务器。

使用 SO_REUSEADDR 选项避免端口占用问题。

2.数据传输协议

设计简单应用层协议：FILE_EXISTS <size>/FILE_NOT_FOUND。

固定缓冲区大小（如 4096 字节）进行分块传输。

实现 ACK 确认机制确保关键指令送达。

2.4.2 安全通信模块

1.SSL/TLS 配置：服务器端加载证书和私钥文件。客户端配置 SSL 上下文（实验环境可禁用证书验证）。使用 PROTOCOL_TLS_SERVER/PROTOCOL_TLS_CLIENT 确保协议兼容性。

2.加密连接建立：服务器将普通 Socket 包装为 SSL Socket。客户端在连接时启用 SSL 加密。处理 SSL 握手过程中的各种异常情况。

2.4.3 文件传输模块

1.文件发送逻辑：服务器检查请求文件是否存在及可读性。分块读取文件内容并发送。实时记录已发送字节数。

2.文件接收逻辑：客户端根据服务器响应创建本地文件。分块接收数据并写入文件。校验接收文件完整性（对比预期大小和实际大小）。

2.4.4 错误处理模块

1.异常分类处理：网络相关异常（连接拒绝、超时等）、文件操作异常（权限不足、文件不存在等）、SSL 特有异常（证书验证失败、协议不匹配等）。

2.资源释放机制：使用 with 语句确保文件/Socket 正确关闭、finally 块中进行最终资源清理、记录详细的错误日志。

2.4.5 用户交互模块

1.命令行接口：支持灵活配置 IP、端口等参数，通过标志位切换 SSL 模式，参数类型检查和默认值设置。

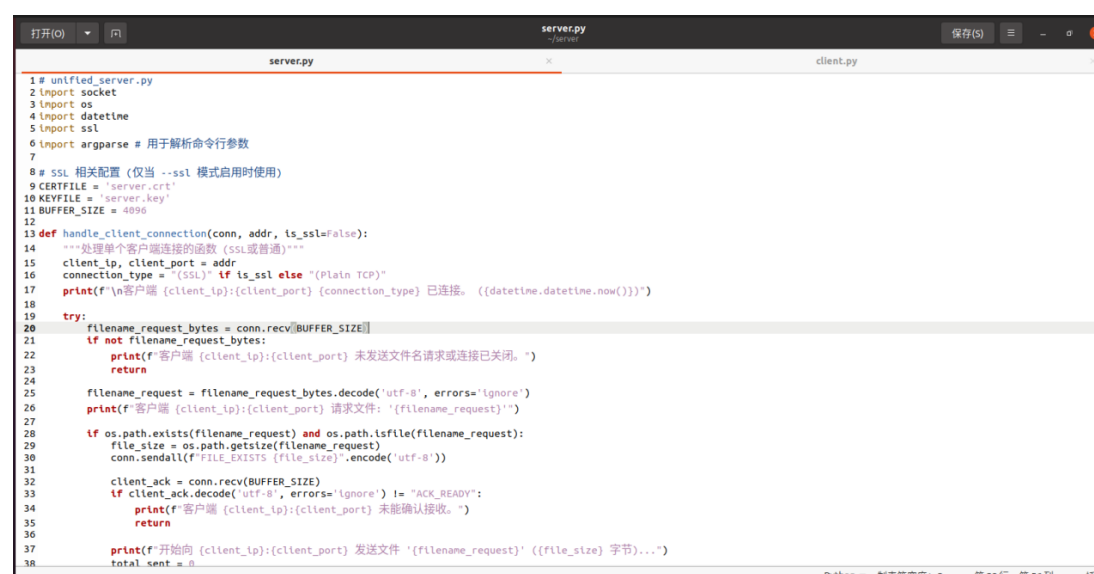
2.运行状态反馈：实时显示连接建立/断开信息，文件传输进度提示，错误信息的友好展示。

3 实验步骤

3.1 实验准备

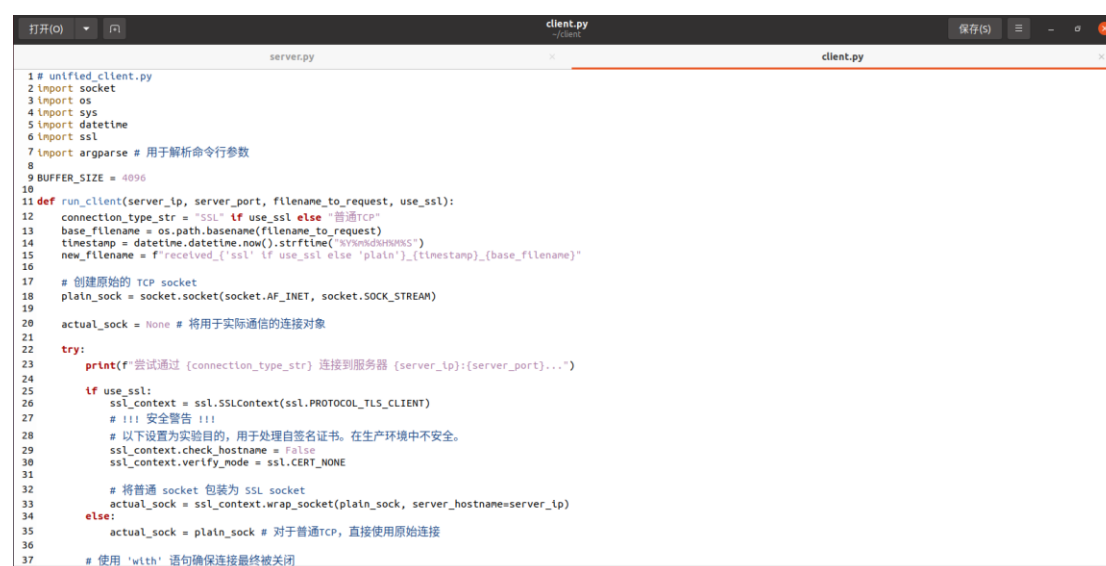
1.实验代码

①服务器端代码 server.py（仅展示部分，详细见程序源代码）



```
1 # unified_server.py
2 import socket
3 import os
4 import datetime
5 import ssl
6 import argparse # 用于解析命令行参数
7
8 # SSL 相关配置 (仅当 --ssl 模式启用时使用)
9 CERTFILE = 'server.crt'
10 KEYFILE = 'server.key'
11 BUFFER_SIZE = 4096
12
13 def handle_client_connection(conn, addr, is_ssl=False):
14     """处理单个客户端连接 (SSL或普通)"""
15     client_ip, client_port = addr
16     connection_type = "(SSL)" if is_ssl else "(Plain TCP)"
17     print(f"\n客户端 {client_ip}:{client_port} {connection_type} 已连接。 ({datetime.datetime.now()})")
18
19     try:
20         filename_request_bytes = conn.recv(BUFFER_SIZE)
21         if not filename_request_bytes:
22             print(f"客户端 {client_ip}:{client_port} 未发送文件名请求或连接已关闭。")
23             return
24
25         filename_request = filename_request_bytes.decode('utf-8', errors='ignore')
26         print(f"客户端 {client_ip}:{client_port} 请求文件: '{filename_request}'")
27
28         if os.path.exists(filename_request) and os.path.isfile(filename_request):
29             file_size = os.path.getsize(filename_request)
30             conn.sendall(f"FILE EXISTS {file_size}".encode('utf-8'))
31
32             client_ack = conn.recv(BUFFER_SIZE)
33             if client_ack.decode('utf-8', errors='ignore') != "ACK_READY":
34                 print(f"客户端 {client_ip}:{client_port} 未能确认接收。")
35                 return
36
37             print(f"开始向 {client_ip}:{client_port} 发送文件 '{filename_request}' ({file_size} 字节)...")
38             total_sent = 0
```

②客户端代码 client.py（仅展示部分，详细见程序源代码）



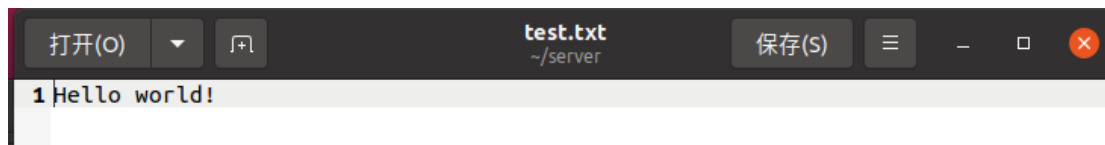
```
1 # unified_client.py
2 import socket
3 import os
4 import sys
5 import datetime
6 import ssl
7 import argparse # 用于解析命令行参数
8
9 BUFFER_SIZE = 4096
10
11 def run_client(server_ip, server_port, filename_to_request, use_ssl):
12     connection_type_str = "SSL" if use_ssl else "普通TCP"
13     base_filename = os.path.basename(filename_to_request)
14     timestamp = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
15     new_filename = f"received_{('ssl' if use_ssl else 'plain')}_{timestamp}_{base_filename}"
16
17     # 创建原始的 TCP socket
18     plain_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19
20     actual_sock = None # 将用于实际通信的连接对象
21
22     try:
23         print(f"尝试通过 {connection_type_str} 连接到服务器 {server_ip}:{server_port}...")
24
25         if use_ssl:
26             ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
27             # !!! 安全警告 !!!
28             # 以下设置为实验目的, 用于处理自签名证书。在生产环境中不安全。
29             ssl_context.check_hostname = False
30             ssl_context.verify_mode = ssl.CERT_NONE
31
32             # 将普通 socket 包装为 SSL socket
33             actual_sock = ssl_context.wrap_socket(plain_sock, server_hostname=server_ip)
34         else:
35             actual_sock = plain_sock # 对于普通TCP, 直接使用原始连接
36
37     # 使用 'with' 语句确保连接最终被关闭
```

2.在服务器端准备好传输文件

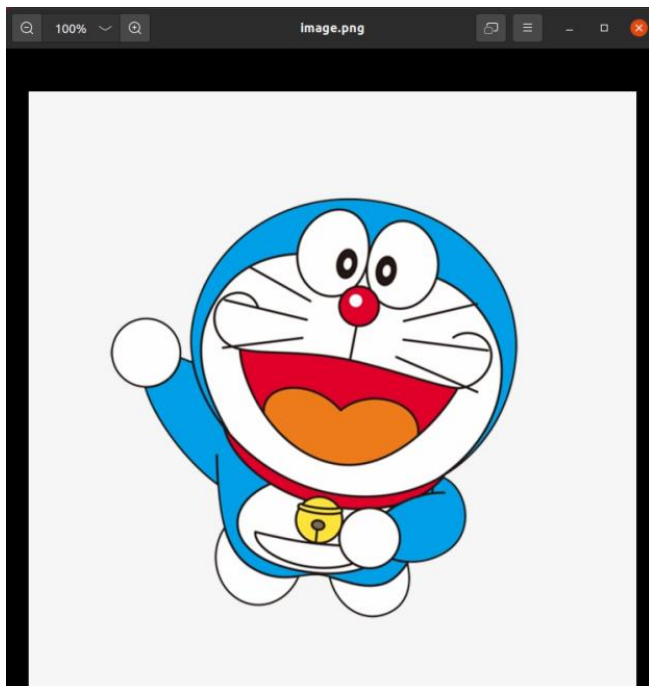
```
ln@ubuntu:~$ cd server
ln@ubuntu:~/server$ ls
image.png  server.crt  test.txt
```

① text.txt

文本内容为“Hello world!”。



② image.png



3.2 实验一：不使用 SSL 的 TCP 文件传输

1. 启动服务器

打开一个终端，切换到 `server.py` 文件目录下。运行以下命令启动服务器，监听一个普通 TCP 端口 12345。

```
ln@ubuntu:~$ cd server
ln@ubuntu:~/server$ python3 server.py --port 12345
普通TCP服务器正在监听 0.0.0.0 端口 12345...
```

2. 启动客户端并请求文本文件 `test.txt`

打开另一个新的终端窗口，切换到 `client.py` 文件目录下。运行以下命令连接到服务器并请求文本文件 `test.txt`。

用户端：

```
ln@ubuntu:~$ cd client
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 test.txt
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
已成功通过 普通TCP 连接到服务器 127.0.0.1:12345
已发送文件请求: 'test.txt'
服务器确认文件存在, 大小为: 12 字节。
开始接收文件, 将保存为 'received_plain_20250511194720_test.txt'...

文件 'test.txt' 已成功接收并保存为 'received_plain_20250511194720_test.txt'。
接收总字节数: 12
```

服务器端：

```
客户端 127.0.0.1:57956 (Plain TCP) 已连接。 (2025-05-11 19:47:20.260765)
客户端 127.0.0.1:57956 请求文件: 'test.txt'
开始向 127.0.0.1:57956 发送文件 'test.txt' (12 字节)...
文件 'test.txt' 发送完成。总共发送 12 字节给 127.0.0.1:57956。
与客户端 127.0.0.1:57956 的事务结束。
```

可以观察到，客户端接收到的文件与服务器端原始文件的大小一致。

3.客户端请求图片文件 image.png

在客户端窗口下运行以下命令连接到服务器并请求图片文件 image.png。

用户端：

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 image.png
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
已成功通过 普通TCP 连接到服务器 127.0.0.1:12345
已发送文件请求: 'image.png'
服务器确认文件存在, 大小为: 194241 字节。
开始接收文件, 将保存为 'received_plain_20250511195042_image.png'...

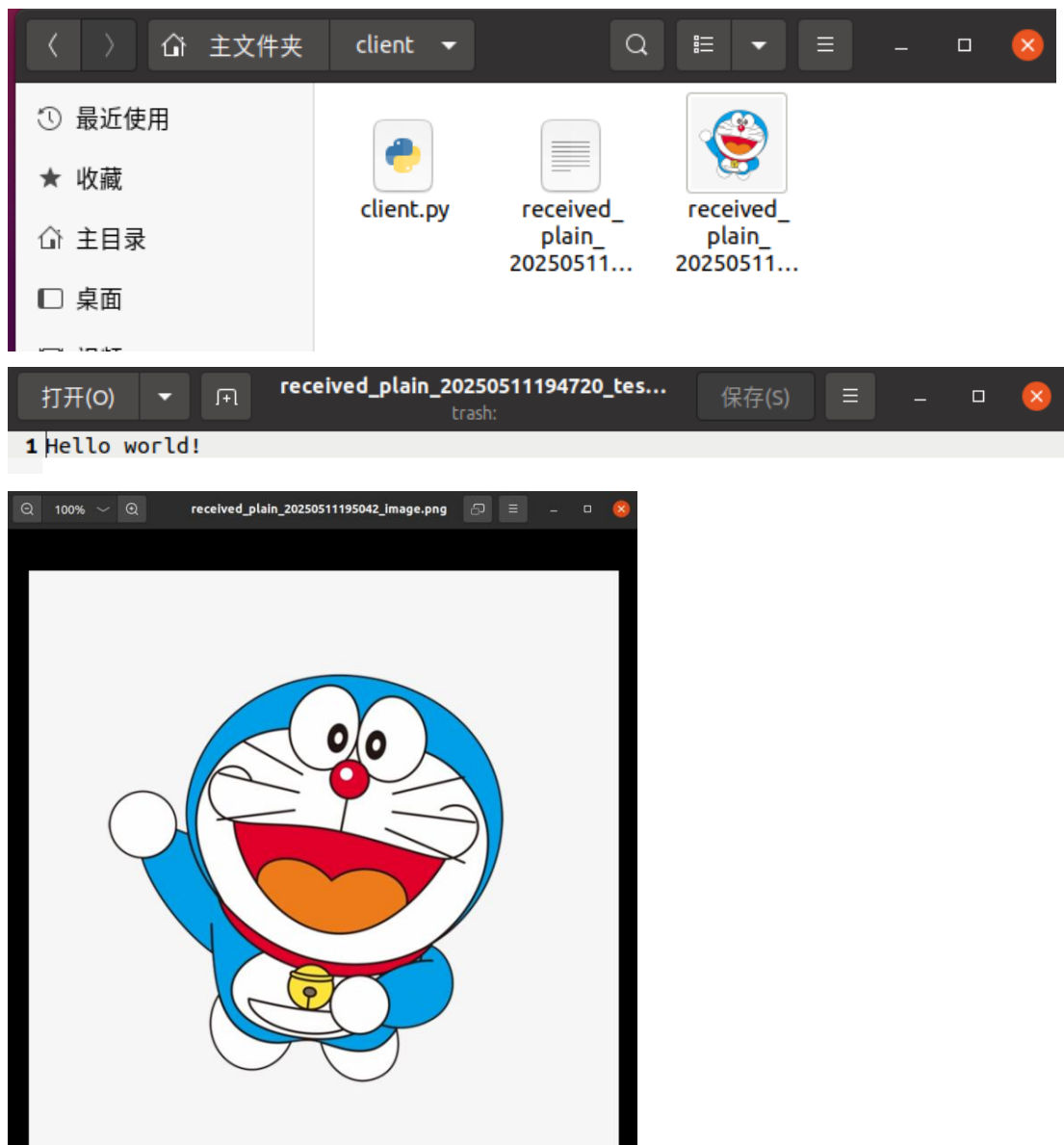
文件 'image.png' 已成功接收并保存为 'received_plain_20250511195042_image.png'。
接收总字节数: 194241
```

服务器端：

```
客户端 127.0.0.1:32936 (Plain TCP) 已连接。 (2025-05-11 19:50:42.970489)
客户端 127.0.0.1:32936 请求文件: 'image.png'
开始向 127.0.0.1:32936 发送文件 'image.png' (194241 字节)...
文件 'image.png' 发送完成。总共发送 194241 字节给 127.0.0.1:32936。
与客户端 127.0.0.1:32936 的事务结束。
```

可以观察到，客户端接收到的文件与服务器端原始文件的大小一致。

4.观察客户端收到的文件



发现文本文件和图片文件都已经收到，内容也完全一致。

5.测试不存在的文件的传输请求

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 test12.txt
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
已成功通过 普通TCP 连接到服务器 127.0.0.1:12345
已发送文件请求: 'test12.txt'
错误报告: 服务器报告文件 'test12.txt' 未找到。
```

```
客户端 127.0.0.1:56046 (Plain TCP) 已连接。 (2025-05-11 19:59:12.569562)
客户端 127.0.0.1:56046 请求文件: 'test12.txt'
错误: 文件 'test12.txt' 在服务器上不存在或不是一个文件。
已通知客户端 127.0.0.1:56046 文件未找到。
与客户端 127.0.0.1:56046 的事务结束。
```

6.断开连接

```
^C
普通TCP 服务器正在关闭...
服务器已关闭。
```

7. 差错处理功能

①若客户端请求文件之前服务器端未开启，客户端提示连接到服务器被拒绝。

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 test.txt
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
错误报告: 连接到服务器 127.0.0.1:12345 被拒绝。
```

②若传输文件不存在，提示文件未找到。

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 test12.txt
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
已成功通过 普通TCP 连接到服务器 127.0.0.1:12345
已发送文件请求: 'test12.txt'
错误报告: 服务器报告文件 'test12.txt' 未找到。
```

```
客户端 127.0.0.1:56046 (Plain TCP) 已连接。 (2025-05-11 19:59:12.569562)
客户端 127.0.0.1:56046 请求文件: 'test12.txt'
错误: 文件 'test12.txt' 在服务器上不存在或不是一个文件。
已通知客户端 127.0.0.1:56046 文件未找到。
与客户端 127.0.0.1:56046 的事务结束。
```

客户端和服务端都会发送错误报告，提示文件未找到。

③若客户端接收文件内容过程中服务器端断开连接，提示文件接收可能不完整，并显示了文件接收的预期大小和实际接收大小。

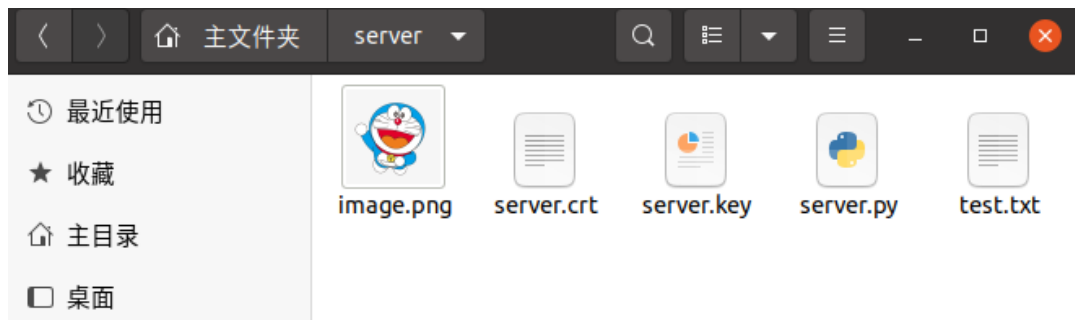
```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12345 large_test_file.dat
尝试通过 普通TCP 连接到服务器 127.0.0.1:12345...
已成功通过 普通TCP 连接到服务器 127.0.0.1:12345
已发送文件请求: 'large_test_file.dat'
服务器确认文件存在，大小为: 1048576000 字节。
开始接收文件，将保存为 'received_plain_20250511210726_large_test_file.dat'...
错误: 文件接收未完成，服务器可能已关闭连接。

文件接收可能不完整。预期大小: 1048576000，实际接收: 225595392
```

3.3 实验二：使用 SSL 的 TCP 文件传输

1.在服务器窗口下，执行 OpenSSL 命令生成 SSL 证书文件和私钥文件，即 server.crt 和 server.key。

```
ln@ubuntu:~/server$ openssl req -new -x509 -days 365 -nodes -out server.crt -key
out server.key -subj "/CN=localhost"
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'server.key'
-----
```



2.启动服务器

运行以下命令启动服务器，监听一个 SSL 端口 12346，并启用 SSL 模式。

```
ln@ubuntu:~/server$ python3 server.py --port 12346 --ssl
SSL 服务器准备在 0.0.0.0 端口 12346 监听...
SSL 证书 'server.crt' 和私钥 'server.key' 加载成功。
```

3.启动客户端并请求文本文件 test.txt

在客户端的终端窗口中，运行以下命令连接到 SSL 服务器并请求文本文件 test.txt，并启用 SSL 模式。

客户端：

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 test.txt --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
已成功通过 SSL 连接到服务器 127.0.0.1:12346
已发送文件请求: 'test.txt'
服务器确认文件存在，大小为: 12 字节。
开始接收文件，将保存为 'received_ssl_20250511201221_test.txt'...

文件 'test.txt' 已成功接收并保存为 'received_ssl_20250511201221_test.txt'。
接收总字节数: 12
```

服务器端：

```
客户端 127.0.0.1:54510 (SSL) 已连接。 (2025-05-11 20:12:21.454918)
客户端 127.0.0.1:54510 请求文件: 'test.txt'
开始向 127.0.0.1:54510 发送文件 'test.txt' (12 字节)...
文件 'test.txt' 发送完成。总共发送 12 字节给 127.0.0.1:54510。
与客户端 127.0.0.1:54510 的事务结束。
```

可以观察到，客户端接收到的文件与服务器端原始文件的大小一致。

4.客户端请求图片文件 image.png

在客户端的终端窗口中，运行以下命令连接到 SSL 服务器并请求图片文件 image.png，并启用 SSL 模式。

客户端：

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 image.png --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
已成功通过 SSL 连接到服务器 127.0.0.1:12346
已发送文件请求: 'image.png'
服务器确认文件存在，大小为: 194241 字节。
开始接收文件，将保存为 'received_ssl_20250511201440_image.png'...

文件 'image.png' 已成功接收并保存为 'received_ssl_20250511201440_image.png'。
接收总字节数: 194241
```

服务器端：

```
客户端 127.0.0.1:55146 (SSL) 已连接。 (2025-05-11 20:14:40.438110)
客户端 127.0.0.1:55146 请求文件: 'image.png'
开始向 127.0.0.1:55146 发送文件 'image.png' (194241 字节)...
文件 'image.png' 发送完成。总共发送 194241 字节给 127.0.0.1:55146。
与客户端 127.0.0.1:55146 的事务结束。
```

可以观察到，客户端接收到的文件与服务器端原始文件的大小一致。

5.观察客户端收到的文件



发现文本文件和图片文件都已经收到，内容也完全一致。

6.测试不存在的文件的传输请求


```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 test22.txt --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
已成功通过 SSL 连接到服务器 127.0.0.1:12346
已发送文件请求: 'test22.txt'
错误报告: 服务器报告文件 'test22.txt' 未找到。
```

```
客户端 127.0.0.1:41024 (SSL) 已连接。(2025-05-11 20:23:28.629501)
客户端 127.0.0.1:41024 请求文件: 'test22.txt'
错误: 文件 'test22.txt' 在服务器上不存在或不是一个文件。
已通知客户端 127.0.0.1:41024 文件未找到。
与客户端 127.0.0.1:41024 的事务结束。
```

7.断开连接

```
^C
SSL 服务器正在关闭...
服务器已关闭。
```

8.差错处理功能

①若客户端请求文件之前服务器端未开启，客户端提示连接到服务器被拒绝。

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 test.txt --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
错误报告: 连接到服务器 127.0.0.1:12346 被拒绝。
```

②若传输文件不存在，提示文件未找到。

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 test22.txt --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
已成功通过 SSL 连接到服务器 127.0.0.1:12346
已发送文件请求: 'test22.txt'
错误报告: 服务器报告文件 'test22.txt' 未找到。
```

```
客户端 127.0.0.1:41024 (SSL) 已连接。(2025-05-11 20:23:28.629501)
客户端 127.0.0.1:41024 请求文件: 'test22.txt'
错误: 文件 'test22.txt' 在服务器上不存在或不是一个文件。
已通知客户端 127.0.0.1:41024 文件未找到。
与客户端 127.0.0.1:41024 的事务结束。
```

客户端和服务端都会发送错误报告，提示文件未找到。

③若客户端接收文件内容过程中服务器端断开连接，提示文件接收可能不完整，并显示了文件接收的预期大小和实际接收大小。

```
ln@ubuntu:~/client$ python3 client.py 127.0.0.1 12346 large_test_file.dat --ssl
尝试通过 SSL 连接到服务器 127.0.0.1:12346...
已成功通过 SSL 连接到服务器 127.0.0.1:12346
已发送文件请求: 'large_test_file.dat'
服务器确认文件存在，大小为: 1048576000 字节。
开始接收文件，将保存为 'received_ssl_20250511210312_large_test_file.dat'...
错误: 文件接收未完成，服务器可能已关闭连接。

文件接收可能不完整。预期大小: 1048576000，实际接收: 150556672
```

4 实验总结和心得体会

4.1 实际上机调试时间

完成本次实验的核心代码编写和初步调试大约花费了 6 小时，后续针对各种差错情况的测试、SSL 功能的集成大约花费了 5 小时，实验报告的撰写又花费了 6 小时，总计约 17 小时。

4.2 调试过程中遇到的问题及解决过程

1.问题一：一开始想使用 Windows 系统进行 OpenSSL 连接，但是在生成证书和密钥时一直报错。

解决方法：在网上搜索和多次尝试无果后，考虑到可能是之前电脑下载过 OpenSSL 时留下的故障，后转去使用 Linux 系统的虚拟机来实现 OpenSSL 连接，在生成证书和密钥时直接成功，未出现报错。

2.问题二：在 SSL 模式下，客户端尝试连接服务器时失败，报告 SSL 相关的错误，如证书验证失败。

解决方法：将生成的证书 `server.crt` 和私钥 `server.key` 与 `server.py` 程序放在同一目录下。

3.问题三：在测试客户端接收文件内容过程中服务器端断开连接时的差错处理功能时，刚开始请求的文件大小太小，文件传输的特别快，还未关闭服务器前，文件已经传输完成。

解决方法：客户端请求一个文件大小较大的文件，使得有充分的时间关闭服务器，来测试这个差错处理功能，最后测试成功。

4.3 收获和提高

通过本次基于 Socket API 的文件传输实验，我们在多个方面都获得了宝贵的经验和显著的提高。在 SOCKET 机制方面，我们对 Socket 的完整生命周期，从创建、绑定、监听到连接、数据传输及关闭，有了清晰的实践认识，深刻体会了 C/S 架构中请求-响应模型的实现，并对阻塞操作和 SSL/TLS 的集成有了初步的感性认识。在协议软件设计方面，我们学习了如何设计简单的应用层信令来协调通信，理解了差错处理在网络编程中的极端重要性，掌握了数据分块传输的原理，并通过命令行参数提升了程序的灵活性。

理论学习上，本次实验巩固了我们对 TCP 协议面向连接、可靠传输特性的理解，并让我们初步接触了 SSL/TLS 协议及其核心概念。在软件工程方面，通过函数封装实现了模块化，通过统一代码和命令行参数提高了代码复用性，并通过大量的异常捕获增强了程序的健壮性，同时也锻炼了我们对网络程序的调试技巧。

4.4 实验设计和安排的不足

回顾本次实验，我们认为其设计和安排在几个方面可以进一步完善以提升学习效果。例如，虽然我们实现了基本的差错处理，但实验可以引导我们思考更细致的错误分类、用户提示或重试机制。当前的服务器是迭代式的，若能引导设计并发服务器，将更贴近实际应用需求，并能深入探讨并发带来的挑战。此外，实验可以增加应用层数据校验的要求，以确保文件内容的绝对完整性，尤其是在模拟不可靠网络或考虑数据篡改时。

在 SSL 安全连接部分，可以更深入地探讨证书的严格验证、不同 SSL 错误的具体处理以及更安全的客户端配置方法，而不仅仅是为自签名证书简化验证流程。最后，引入对性能优化、更复杂应用层协议设计的思考，也能使实验内容更具深度和广度。

附录：程序源代码

1. 客户端代码

```
# unified_client.py
import socket
import os
import sys
import datetime
import ssl
import argparse # 用于解析命令行参数

BUFFER_SIZE = 4096

def run_client(server_ip, server_port, filename_to_request, use_ssl):
    connection_type_str = "SSL" if use_ssl else "普通 TCP"
    base_filename = os.path.basename(filename_to_request)
    timestamp = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
    new_filename = f"received_{'ssl' if use_ssl else
'plain'}_{timestamp}_{base_filename}"

    # 创建原始的 TCP socket
    plain_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    actual_sock = None # 将用于实际通信的连接对象

    try:
        print(f"尝试通过 {connection_type_str} 连接到服务器
{server_ip}:{server_port}...")

        if use_ssl:
            ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
            ssl_context.check_hostname = False
            ssl_context.verify_mode = ssl.CERT_NONE

            # 将普通 socket 包装为 SSL socket
            actual_sock = ssl_context.wrap_socket(plain_sock,
server_hostname=server_ip)
        else:
            actual_sock = plain_sock # 对于普通 TCP，直接使用原始连接

    # 使用 'with' 语句确保连接最终被关闭
    with actual_sock:
        actual_sock.connect((server_ip, server_port))
```

```

        print(f"已成功通过 {connection_type_str} 连接到服务器
{server_ip}:{server_port}")
        if use_ssl:
            pass

        actual_sock.sendall(filename_to_request.encode('utf-8'))
        print(f"已发送文件请求: '{filename_to_request}'")

        server_response_bytes = actual_sock.recv(BUFFER_SIZE)
        server_response = server_response_bytes.decode('utf-8',
errors='ignore')

        if server_response.startswith("FILE_EXISTS"):
            try:
                parts = server_response.split()
                file_size = int(parts[1])
                print(f"服务器确认文件存在, 大小为: {file_size} 字节。
")

            except (IndexError, ValueError):
                print("错误: 服务器返回的文件大小格式不正确。")
                return # 已经退出了 with actual_sock, 所以会自动关闭

            actual_sock.sendall("ACK_READY".encode('utf-8'))

            print(f"开始接收文件, 将保存为 '{new_filename}'...")
            total_received = 0
            with open(new_filename, 'wb') as f:
                while total_received < file_size:
                    bytes_to_receive_now = min(BUFFER_SIZE,
file_size - total_received)
                    bytes_received =
actual_sock.recv(bytes_to_receive_now)
                    if not bytes_received:
                        print("错误: 文件接收未完成, 服务器可能已关闭连
接。")
                        break
                    f.write(bytes_received)
                    total_received += len(bytes_received)

            if total_received == file_size:
                print(f"\n 文件 '{filename_to_request}' 已成功接收并保
存为 '{new_filename}'。")
                print(f"接收总字节数: {total_received}")
            else:

```

```

        print(f"\n 文件接收可能不完整。预期大小: {file_size},
实际接收: {total_received}")

        elif server_response == "FILE_NOT_FOUND":
            print(f"错误报告: 服务器报告文件 '{filename_to_request}'
未找到。")
        else:
            print(f"收到未知的服务器响应: '{server_response}'")

    except ConnectionRefusedError:
        print(f"错误报告: 连接到服务器 {server_ip}:{server_port} 被拒绝。")
    except socket.timeout:
        print("错误报告: 连接超时。")
    except ssl.SSLCertVerificationError as e: # SSL 特有的错误
        print(f"错误报告: SSL 证书验证失败: {e}")
        print("这可能是由于服务器使用的是自签名证书, 并且客户端未配置为信任
它。")
    except ssl.SSLError as e: # 其他 SSL 错误
        print(f"错误报告: 发生 SSL 错误: {e}")
    except socket.error as e:
        print(f"错误报告: 发生套接字错误: {e}")
    except Exception as e:
        print(f"错误报告: 发生预料之外的错误: {e}")
    finally:
        if actual_sock is None and plain_sock.fileno() != -1: # 如果包
装失败且原始 socket 还打开
            plain_sock.close()
        elif actual_sock is not None and actual_sock.fileno() == -1
and plain_sock.fileno() != -1: # 如果包装的 socket 关闭了, 但原始的没关
(不太可能)
            plain_sock.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="TCP/SSL 文件客户端")
    parser.add_argument('server_ip', help="服务器的 IP 地址")
    parser.add_argument('server_port', type=int, help="服务器的端口号")
    parser.add_argument('filename', help="要从服务器请求的文件名")
    parser.add_argument('--ssl', action='store_true', help="使用 SSL 连
接模式")

    args = parser.parse_args()

```

```
run_client(args.server_ip, args.server_port, args.filename,
args.ssl)
```

2.服务器端代码

```
# unified_server.py
import socket
import os
import datetime
import ssl
import argparse # 用于解析命令行参数

# SSL 相关配置 (仅当 --ssl 模式启用时使用)
CERTFILE = 'server.crt'
KEYFILE = 'server.key'
BUFFER_SIZE = 4096

def handle_client_connection(conn, addr, is_ssl=False):
    """处理单个客户端连接的函数 (SSL 或普通)"""
    client_ip, client_port = addr
    connection_type = "(SSL)" if is_ssl else "(Plain TCP)"
    print(f"\n客户端 {client_ip}:{client_port} {connection_type} 已连接。 ({datetime.datetime.now()})")

    try:
        filename_request_bytes = conn.recv(BUFFER_SIZE)
        if not filename_request_bytes:
            print(f"客户端 {client_ip}:{client_port} 未发送文件名请求或连接已关闭。")
            return

        filename_request = filename_request_bytes.decode('utf-8',
errors='ignore')
        print(f"客户端 {client_ip}:{client_port} 请求文件: '{filename_request}'")

        if os.path.exists(filename_request) and
os.path.isfile(filename_request):
            file_size = os.path.getsize(filename_request)
            conn.sendall(f"FILE_EXISTS {file_size}".encode('utf-8'))

            client_ack = conn.recv(BUFFER_SIZE)
            if client_ack.decode('utf-8', errors='ignore') !=
"ACK_READY":
                print(f"客户端 {client_ip}:{client_port} 未能确认接收。")
```

```

        return

        print(f"开始向 {client_ip}:{client_port} 发送文件
'{filename_request}' ({file_size} 字节)...")
        total_sent = 0
        with open(filename_request, 'rb') as f:
            while True:
                bytes_read = f.read(BUFFER_SIZE)
                if not bytes_read:
                    break
                conn.sendall(bytes_read)
                total_sent += len(bytes_read)
            print(f"文件 '{filename_request}' 发送完成。总共发送
{total_sent} 字节给 {client_ip}:{client_port}。")

        else:
            error_msg = f"错误: 文件 '{filename_request}' 在服务器上不存在或不是一个文件。"
            print(error_msg)
            conn.sendall("FILE_NOT_FOUND".encode('utf-8'))
            print(f"已通知客户端 {client_ip}:{client_port} 文件未找到。")

    except ssl.SSLError as e: # SSL 特有的错误
        print(f"与客户端 {client_ip}:{client_port} 通信时发生 SSL 错误:
{e}")
    except socket.error as e:
        print(f"与客户端 {client_ip}:{client_port} 通信时发生套接字错误:
{e}")
    except UnicodeDecodeError:
        print(f"无法解码来自 {client_ip}:{client_port} 的文件名请求。")
    except Exception as e:
        print(f"处理客户端 {client_ip}:{client_port} 时发生未知错误:
{e}")
    finally:
        print(f"与客户端 {client_ip}:{client_port} 的事务结束。")
        if conn:
            conn.close()

def run_server(host, port, use_ssl):
    # 创建主监听 socket
    listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_sock.bind((host, port))
    listen_sock.listen(5)

```

```

ssl_context = None
if use_ssl:
    print(f"SSL 服务器准备在 {host} 端口 {port} 监听...")
    ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    try:
        ssl_context.load_cert_chain(certfile=CERTFILE,
keyfile=KEYFILE)
        print(f"SSL 证书 '{CERTFILE}' 和私钥 '{KEYFILE}' 加载成功。")
    except FileNotFoundError:
        print(f"错误: SSL 证书文件 '{CERTFILE}' 或私钥文件
'{KEYFILE}' 未找到。")
        print("请确保已生成证书和私钥，并将它们放在正确的路径下。")
        print("例如: openssl req -new -x509 -days 365 -nodes -out
server.crt -keyout server.key -subj \"/CN=localhost\"")
        listen_sock.close()
        return
    except ssl.SSLError as e:
        print(f"加载 SSL 证书时发生错误: {e}")
        listen_sock.close()
        return
else:
    print(f"普通 TCP 服务器正在监听 {host} 端口 {port}...")

try:
    while True:
        conn_plain = None # 初始化为 None
        try:
            conn_plain, addr = listen_sock.accept() # 接受原始连接

            actual_conn = None # 将用于实际通信的连接对象
            if use_ssl and ssl_context:
                try:
                    # 将接受的连接包装为 SSL socket
                    actual_conn =
ssl_context.wrap_socket(conn_plain, server_side=True)
                except ssl.SSLError as e:
                    print(f"SSL 握手失败来自 {addr}: {e}")
                    if conn_plain: conn_plain.close()
                    continue
                except Exception as e:
                    print(f"包装套接字为 SSL 时发生未知错误来自 {addr}:
{e}")

                    if conn_plain: conn_plain.close()

```

```

        continue
    else:
        actual_conn = conn_plain # 对于普通 TCP，直接使用原始
连接

        # 使用 'with' 语句确保连接最终被关闭
        with actual_conn:
            handle_client_connection(actual_conn, addr,
is_ssl=use_ssl)

    except socket.error as e:
        # 这个错误通常是针对 listen_sock.accept() 的
        print(f"接受连接时发生套接字错误: {e}")
        if conn_plain: # 确保在发生错误时关闭已接受的原始连接
            conn_plain.close()
    except Exception as e: # 捕获 accept 或 wrap_socket 阶段的其
他潜在错误

        print(f"处理新连接时发生意外错误: {e}")
        if conn_plain:
            conn_plain.close()

    except KeyboardInterrupt:
        server_type = "SSL" if use_ssl else "普通 TCP"
        print(f"\n{server_type} 服务器正在关闭...")
    finally:
        if listen_sock:
            listen_sock.close()
        print("服务器已关闭。")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="TCP/SSL 文件服务器")
    parser.add_argument('--host', default='0.0.0.0', help="服务器监听的主机地址 (默认: 0.0.0.0)")
    parser.add_argument('--port', type=int, required=True, help="服务器监听的端口号 (例如: 12345 用于普通 TCP, 12346 用于 SSL)")
    parser.add_argument('--ssl', action='store_true', help="启用 SSL 模式 (需要 server.crt 和 server.key 文件)")

    args = parser.parse_args()

    run_server(args.host, args.port, args.ssl)

```