

分布式期末项目——共享文档编辑系统

【专业】计算机科学与技术

【学号】22336259

【姓名】谢宇桐

一、实验题目

设计并实现可同时支持多人进行文档编辑的系统。允许每个人进行读写操作，并能够保障系统的一致性，此外还应具备一定的容错能力。

要求

- 支持多人同时在线编辑文档；
- 通信方式选择 RPC；
- 具备分布式系统互斥协议；
- 支持至少一种系统一致性；
- 具备一定的容错能力；

二、实验环境

- 操作系统：Windows
- 编程语言：Python
- 相关库：xmlrpc.server、xmlrpc.client、threading、socket、sys、shlex

三、系统设计

1. 系统架构

系统由两个主要部分组成：服务器端（server.py）和客户端（client.py）。

2. 服务器端（server.py）

服务器端实现了一个基于 XML-RPC 和套接字的文档编辑服务器。支持多个客户端同时连接，并对共享文档进行编辑、删除和替换操作。以下是对代码的详细解释：

(1) 导入模块

```
import xmlrpc.server
import threading
import socket
```

- xmlrpc.server：用于创建 XML-RPC 服务器，允许客户端通过 HTTP 协议调用服务器端的方法。
- threading：用于创建和管理线程，实现多线程功能，例如处理多个客户端连接。
- socket：用于实现基于套接字的网络通信，用于客户端的实时通知。

(2) SharedDoc 类

SharedDoc 类主要负责以下内容：

- **文档内容的管理**：维护共享文档的内容，实现编辑（添加、删除、替换）文档操作。
- **线程安全**：确保文档的修改操作是线程安全的，防止多个客户端同时修改导致数据不一致。
- **客户端通知**：在文档内容被修改时，负责通知所有已连接的客户端，以保持文档的一致性。

初始化

```
class SharedDoc:
    def __init__(self):
        self.doc = "" # 共享文档的初始内容为空
        self.lock = threading.Lock() # 用于线程安全的锁，实现分布式系统互斥协议
        self.clients = {} # 存储客户端套接字的字典，用于实时通知客户端文档的变化
```

文档操作（编辑、删除、替换、查看）

with self.lock 用于确保对共享资源（即文档内容）的访问是线程安全的。它使用了上下文管理器的方式来自动获取和释放锁。当多个线程尝试同时修改共享资源时，**with self.lock** 确保一次只有一个线程可以执行被保护的代码块。实现了分布式系统互斥协议。

```
class SharedDoc:
    # ...初始化

    def edit(self, user_id, content):
        with self.lock:
            print(f"{user_id} is editing the document.") # 告诉服务器端用户正在
操作文档
            self.doc += content # 将 content 添加到文档末尾
            self.notify_clients(user_id, "edit", content) # 调用
notify_clients通知所有客户端文档已被编辑
            print(f"Document after edit by {user_id}: {self.doc}") # 打印当前
文档内容

    def delete(self, user_id, text):
        with self.lock:
            print(f"{user_id} is deleting text from the document.")
            self.doc = self.doc.replace(text, "") # 使用replace方法进行删除文档
中的 text
            self.notify_clients(user_id, "delete", text)
            print(f"Document after deletion by {user_id}: {self.doc}")

    def replace(self, user_id, old_text, new_text):
        with self.lock:
            print(f"{user_id} is replacing text in the document.")
            self.doc = self.doc.replace(old_text, new_text) # 使用replace方法
将文档中的old_text替换为new_text
            self.notify_clients(user_id, "replace", old_text + " -> " +
new_text)
```

```

        print(f"Document after replacement by {user_id}: {self.doc}")

    def get_doc(self):
        return self.doc # 返回当前文档的内容

```

客户端注册

`fileno()` 属于文件对象，用于获取文件对象的文件描述符（file descriptor）。它是一个整数值，用于在底层操作系统中唯一标识一个打开的文件或套接字。

```

def register_client(self, client_socket):
    # 将客户端的套接字注册到clients字典中，使用fileno()作为键。
    self.clients[client_socket.fileno()] = client_socket

```

客户端通知

在系统中，我加入了一个同步通知消息功能，在其他客户对共享文档操作后会通知其它所有已登录客户其修改操作。便于让每个客户及时更新进度。

```

def notify_clients(self, user_id, action, details):
    message = f"{user_id} {action} '{details}'\n" # 通知消息
    for client_socket in self.clients.values(): # 遍历所有已注册的客户端套接字
        try:
            client_socket.sendall(message.encode()) # 将通知消息发送给每个客户端
        except Exception as e:
            print(f"Error notifying client: {e}") # 如果发送失败，打印错误信息

```

(3) DocService 类

`DocService` 类充当了一个接口或适配器的角色，它作为客户端和 `SharedDoc` 之间的桥梁，主要负责：

- **封装和暴露方法**：将 `SharedDoc` 类的方法封装起来，通过 XML-RPC 服务器暴露给客户端。这样，客户端可以通过网络调用这些方法来编辑文档。
- **参数解析**：params 参数用于从 XML-RPC 客户端接收方法调用时传递的参数，该类从这个参数中提取所需的信息，并调用 `SharedDoc` 的相应方法。
- **错误处理和响应**：处理客户端请求时可能发生的错误，并返回适当的响应给客户端。

也就是说 `DocService` 类它不直接管理文档内容，而是通过调用 `SharedDoc` 的方法来实现文档的编辑和查询。

XML-RPC 方法

```

class DocService:
    def __init__(self, shared_doc):
        # 将SharedDocument的方法暴露给 XML-RPC 客户端
        self.shared_doc = shared_doc

```

```

def edit_doc(self, params):
    user_id, content = params # 解析参数
    self.shared_doc.edit(user_id, content) # 调用方法
    return f"Edit by {user_id} successful."

def delete_doc(self, params):
    user_id, text = params
    self.shared_doc.delete(user_id, text)
    return f"Deletion by {user_id} successful."

def replace_doc(self, params):
    user_id, old_text, new_text = params
    self.shared_doc.replace(user_id, old_text, new_text)
    return f"Replacement by {user_id} successful."

def get_doc(self):
    return self.shared_doc.get_doc()

def register_client(self, client_socket):
    self.shared_doc.register_client(client_socket)

```

(4) 启动服务器

```

def start_server(shared_doc, port=8000):
    print("Attempting to start server on port 8000...")
    try:
        # 启动 XML-RPC 服务器
        server = xmlrpc.server.SimpleXMLRPCServer(("localhost", port))
        # 将 DocumentService 实例注册到服务器, 使其方法可以通过 XML-RPC 调用
        server.register_instance(DocService(shared_doc))
        print("Server started on port 8000...")

        # 启动套接字服务器
        sock_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock_server.bind(("localhost", port + 1)) # 使用一个额外的端口
        sock_server.listen(5)
        print(f"Socket server started on port {port + 1}...")

        def handle_socket_connections():
            while True: # 不断接受新的客户端连接
                # 将客户端套接字注册到 SharedDoc 中
                client_socket, addr = sock_server.accept()
                print(f"New client connected: {addr}")
                shared_doc.register_client(client_socket)

        # 创建并启动一个守护线程来处理套接字连接。守护线程不会阻止主程序退出
        threading.Thread(target=handle_socket_connections,
            daemon=True).start()

        # 启动 XML-RPC 服务器的主循环, 开始接受和处理 XML-RPC 请求
        server.serve_forever()
    except socket.error as e:
        print(f"Socket error: {e}")

```

```
except Exception as e:
    print(f"Error: {e}")
```

(5) 程序入口

创建实例，服务器在后台运行，等待客户端的连接和操作

```
if __name__ == "__main__":
    shared_document = SharedDocument()
    # 启动一个线程运行 start_server 函数，传入 shared_doc 和端口号 8000
    server_thread = threading.Thread(target=start_server, args=
    (shared_document, 8000))
    server_thread.start()
```

3. 客户端 (client.py)

客户端程序用于通过 XML-RPC 和套接字与服务器进行交互，实现文档编辑功能。以下是对代码的详细解释：

(1) 导入模块

```
import xmlrpc.client
import sys
import threading
import shlex
import socket
```

- xmlrpc.client：用于实现 XML-RPC 协议的客户端功能，允许客户端通过 HTTP 调用服务器端的方法。
- sys：用于访问与 Python 解释器相关的功能，例如命令行参数。
- shlex：用于解析字符串，类似于 shell 的方式，将字符串分割成命令和参数。

(2) Client 类

client 类主要作用是作为用户与服务器之间的接口，允许用户通过命令行与共享文档进行交互。

初始化

```
class Client:
    def __init__(self, user_id, port=8000):
        self.user_id = user_id
        self.port = port
        # 创建一个 XML-RPC 客户端代理对象 client_proxy, 用于调用服务器端的方法
        self.client_proxy = xmlrpc.client.ServerProxy(f"http://localhost:{self.port}")
        self.socket_port = port + 1 # 套接字端口
        self.socket_client = socket.socket(socket.AF_INET,
        socket.SOCK_STREAM)
        self.socket_client.connect(("localhost", self.socket_port))
        print("Connected to the server via socket.")

        # 启动接收通知的线程
        threading.Thread(target=self.receive_notifications,
        daemon=True).start()
```

接收通知

```
def receive_notifications(self):
    while True:
        try:
            # 接收服务器发送的消息, 并打印通知内容
            message = self.socket_client.recv(1024).decode()
            if message:
                print(f"Notification: {message.strip()}")
                # print(self.client_proxy.get_doc())
        except Exception as e:
            print(f"Error receiving notification: {e}")
            # break
```

文档编辑功能

客户端提供了以下文档编辑功能:

```
def edit_doc(self, content):
    try:
        # 将用户 ID 和内容传递给服务器
        self.client_proxy.edit_doc((self.user_id, content))
        print("Document after edit:")
        # 操作完后显示修改后的文档
        if(self.client_proxy.get_doc()):
            print(self.client_proxy.get_doc())
    except Exception as e:
        print(f"Error editing document: {e}")

def delete_doc(self, text):
    try:
        self.client_proxy.delete_doc((self.user_id, text))
```

```

        print("Document after deletion:")
        print(self.client_proxy.get_doc())
    except Exception as e:
        print(f"Error deleting from document: {e}")

def replace_doc(self, old_text, new_text):
    try:
        # 将用户 ID、旧文本和新文本传递给服务器
        self.client_proxy.replace_doc((self.user_id, old_text, new_text))
        print("Document after replacement:")
        print(self.client_proxy.get_doc())
    except Exception as e:
        print(f"Error replacing in document: {e}")

def view_doc(self):
    try:
        print("Current document content:")
        # 获取当前文档内容并打印
        print(self.client_proxy.get_doc())
    except Exception as e:
        print(f"Error viewing document: {e}")

```

(3) 命令帮助

在系统中我加入了命令查询语句。客户端可以进入系统后输入help查看支持的各命令的用法：

```

def print_help():
    print("Client Help:")
    print("  Commands:")
    print("    edit \"content\"    - Edit the document with the given content.")
    print("    delete \"text\"      - Delete the text from the document.")
    print("    replace \"old_text\" \"new_text\" - Replace old_text with new_text in the document.")
    print("    view              - view the current document.")
    print("    exit              - Exit the system.")
    print("    help              - Show this help message.")

```

(4) 命令解析和执行

根据用户输入的命令和参数，调用相应的客户端方法。如果命令无效或参数不足，打印错误信息并显示帮助信息。

```

def command_handler(client, command, args):
    if command == "edit":
        if len(args) > 0:
            client.edit_doc(" ".join(args))

```

```

        else:
            print("Please specify content to edit.")
    elif command == "delete":
        if len(args) > 0:
            client.delete_doc(" ".join(args))
        else:
            print("Please specify text to delete.")
    elif command == "replace":
        if len(args) >= 2:
            client.replace_doc(args[0], args[1])
        else:
            print("Please specify old text and new text to replace.")
    elif command == "view":
        client.view_doc()
    elif command == "exit":
        print(f"User {client.user_id} has logged out.")
        sys.exit(0)
    else:
        print("Invalid command or arguments.")
        print_help()

```

(5) 主循环

```

def main_loop(user_id, port=8000):
    # 初始化客户端
    client = Client(user_id, port)
    print(f"User {user_id} has logged in. Type 'help' for commands.")
    while True: # 进入主循环, 等待用户输入命令
        try:
            input_cmd = input("Enter command: ").strip() # 解析用户输入的命令和参
数

            if not input_cmd:
                continue

            command_parts = shlex.split(input_cmd)
            cmd = command_parts[0].lower()
            args = command_parts[1:]
            # 调用 command_handler 函数处理用户输入的命令
            if cmd in ["edit", "delete", "replace", "view", "exit", "help"]:
                command_handler(client, cmd, args)
            else:
                print("Invalid command or arguments.")
                print_help()
        except Exception as e:
            print(f"An error occurred: {e}")

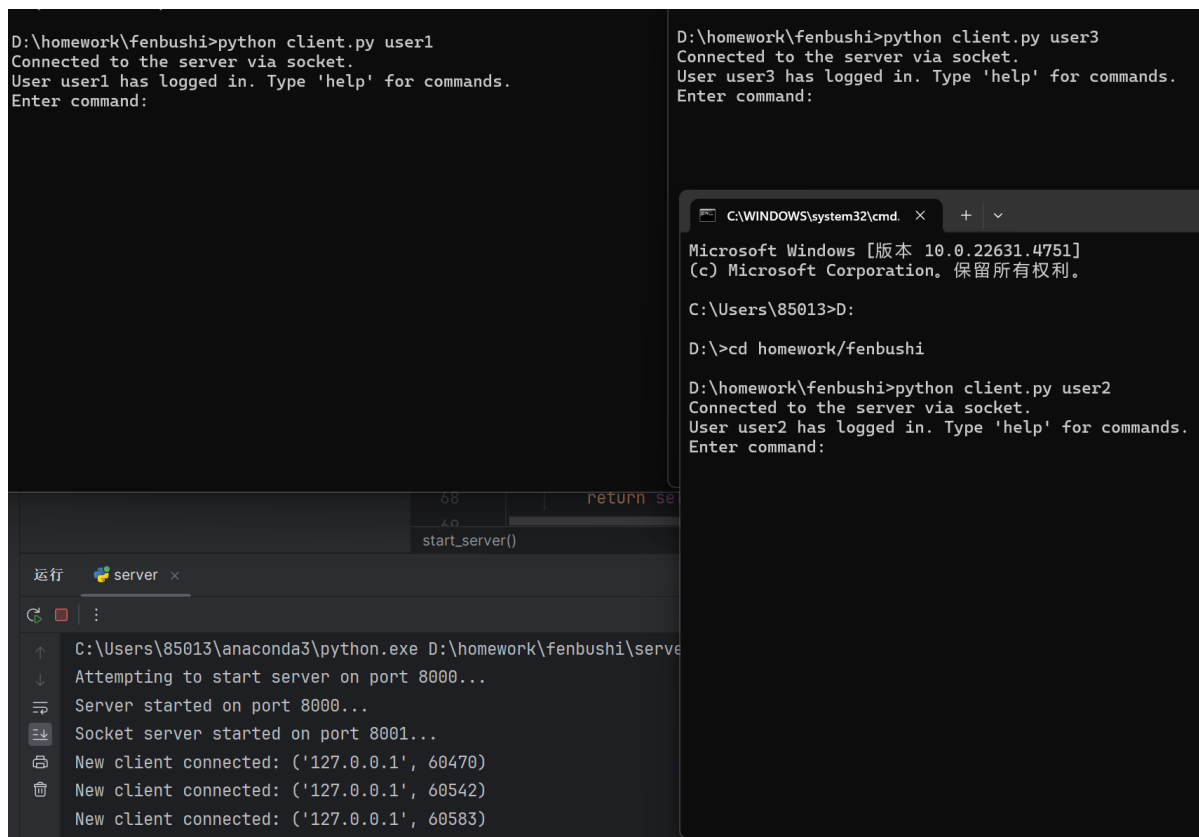
```

(6) 程序入口

```
if __name__ == "__main__":
    # 检查命令行参数是否正确
    if len(sys.argv) != 2 or sys.argv[1].lower() == "help":
        print("Usage: python client1.py <user_id>")
        print_help()
        sys.exit(1)
    # 否则，从命令行参数中获取用户 ID，调用 main_loop 函数启动客户端
    user_id = sys.argv[1]
    main_loop(user_id)
```

四、实验结果演示

首先启动服务器，再打开多个终端模仿多个客户端登录。这边打开三个客户端进行模拟，用户ID分别为 user1、user2、user3：



The screenshot displays a terminal window with two main sections. The top section shows the execution of a Python script to start a server and three separate client sessions. The bottom section shows the output of the server script, which logs the server's status and the connections of the three clients.

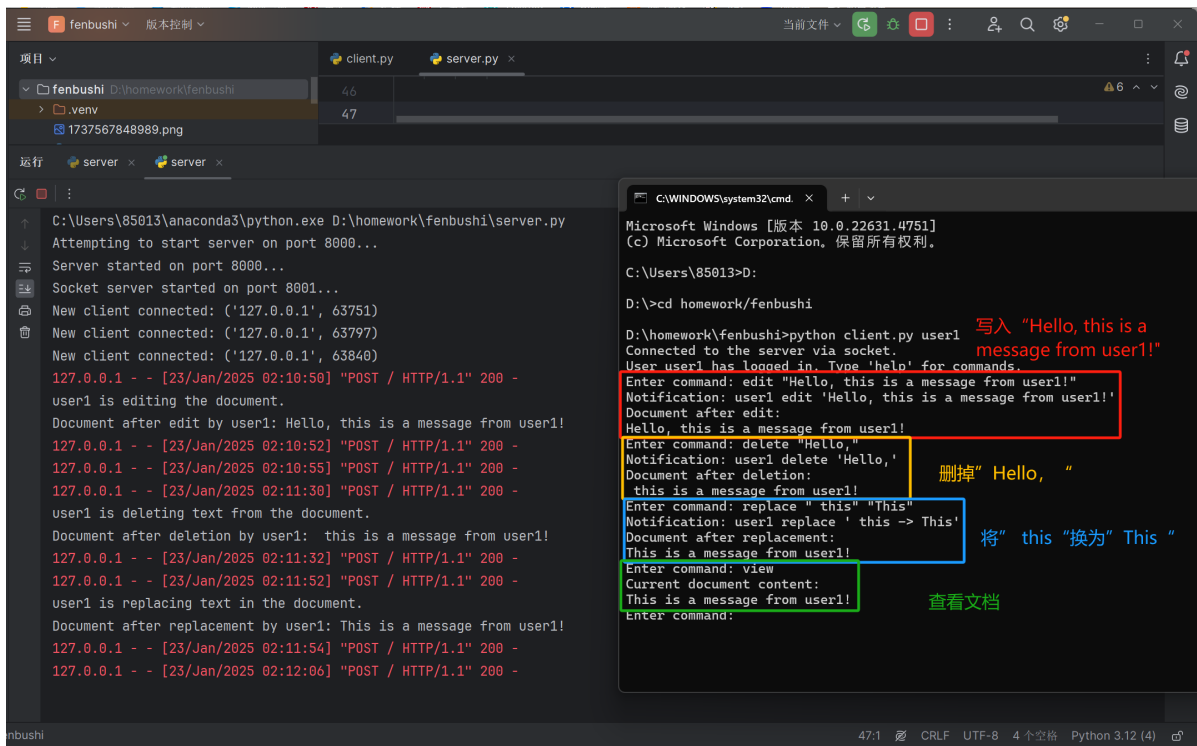
```
D:\homework\fenbushi>python client.py user1
Connected to the server via socket.
User user1 has logged in. Type 'help' for commands.
Enter command:

D:\homework\fenbushi>python client.py user3
Connected to the server via socket.
User user3 has logged in. Type 'help' for commands.
Enter command:

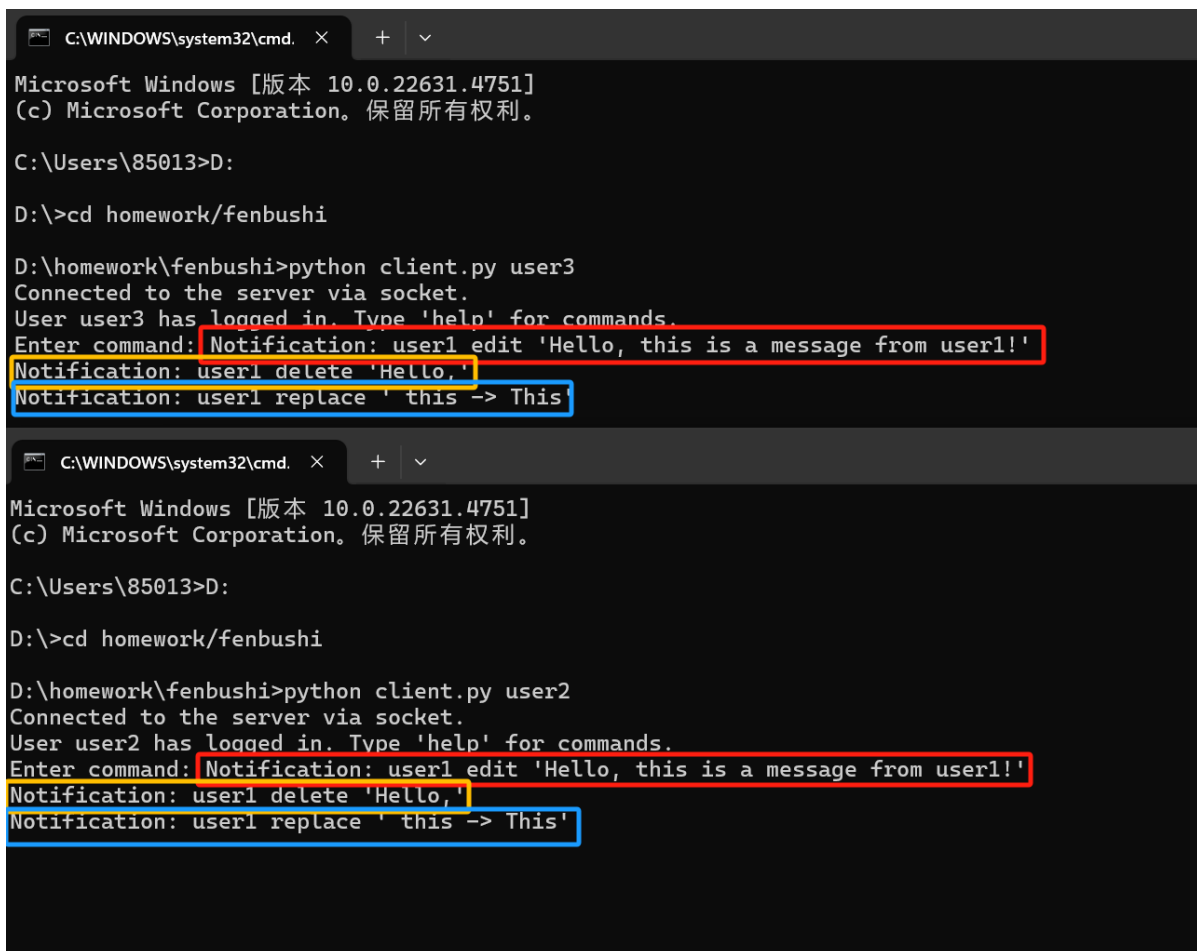
D:\homework\fenbushi>python client.py user2
Connected to the server via socket.
User user2 has logged in. Type 'help' for commands.
Enter command:
```

```
运行 server x
C:\Users\85013\anaconda3\python.exe D:\homework\fenbushi\server.py
Attempting to start server on port 8000...
Server started on port 8000...
Socket server started on port 8001...
New client connected: ('127.0.0.1', 60470)
New client connected: ('127.0.0.1', 60542)
New client connected: ('127.0.0.1', 60583)
```

其中一个客户进行编辑、删除、替换、查看文档



可以发现，在其它客户端可以看到user1对文档进行的更改：



我们在客户端2和客户端3也可以进行编辑，且数据会同步更新：

1. 支持多人同时在线编辑文档

系统通过 XML-RPC 允许多个客户端（用户）连接到服务器并进行文档编辑操作。每个客户端都可以发送编辑、删除和替换操作到服务器，服务器将这些更改应用到共享文档中，并通过套接字通知所有客户端。这满足了多人同时在线编辑文档的要求。

2. 通信方式选择 RPC

系统使用 XML-RPC 作为通信方式，这是远程过程调用（RPC）的一种实现。客户端通过 XML-RPC 调用服务器上的方法来编辑文档，这满足了使用 RPC 的要求。

3. 具备分布式系统互斥协议

系统通过 `threading.Lock()` 实现了锁机制，确保在编辑文档时，只有一个客户端可以修改文档，从而避免了并发写入的问题。

4. 支持至少一种系统一致性

通过锁机制，系统确保了文档的一致性。每次只有一个客户端可以修改文档，其他客户端在修改完成后会收到通知，从而保证了所有客户端看到的文档状态是一致的。

5. 具备一定的容错能力

系统在客户端和服务端都进行了异常处理，例如在发送通知失败时打印错误信息，并且语句都用 `try` 和 `except`，当有遇到运行错误不会强制退出，在用户输入错误时也会有提示，这表明系统具有一定的容错能力。

当然，这个代码仍有许多不足之处，比如当客户对文档进行修改时，通知信息是发给包括自己的所有客户，而不是发给除自己之外的所有客户；三个编辑方法过于基础，不够全面等等。

经过这次实验，我对分布式系统有了更深一步的理解，了解了分布式系统在日常生活中的有如此普遍的应用，以及它在人类现代生活的重要作用。同时我也对 RPC 的原理和应用有了进一步的了解。

至此，实验完成。