中山大學

《基于 gRPC 的分布式文件系统实现》

（2023年秋季学期 分布式系统课程设计）

授课教师：	吴维刚
姓　　名：	黄河锦
班　　级：	计科大数据班
学　　号：	21307404
日　　期：	2024.1.9

1 题目要求

设计一个分布式文件系统。该文件系统可以是client-server架构，也可以是P2P非集中式架构。 要求文件系统具有基本的访问、打开、删除、缓存等功能，同时具有一致性、支持多用户特点。 在设计过程中能够体现在分布式课程中学习到的一些机制或者思想，例如Paxos共识、缓存更新机制、访问控制机制、并行扩展等。

2 解决思路

2.1 系统架构选择

首先，我们需要决定是采用`client-server`架构还是`P2P`架构。`client-server`架构通常简单明了，有利于实现管理和数据一致性。`P2P`架构则能够更好地扩展和抵抗故障，但实现一致性更加复杂。鉴于作业要求和Python的易用性，我们选择`client-server`架构。

2.2 文件操作和通信协议

- 使用Python实现所有功能，利用Python的多进程和网络库（如`socket`或`http`）。
- 文件系统节点之间的通信采用RPC模式，可以选择`XML-RPC`或`gRPC`。`gRPC`基于HTTP/2，性能更优，且支持多语言，因此我们选择`gRPC`。

2.3 文件操作类型

实现以下基本文件操作：

- 创建（Create）
- 删除（Delete）
- 访问（Read/Write）

2.4 关键技术

2.4.1 客户端缓存

- 客户端缓存可以使用内存（如Python的字典）或磁盘文件（如SQLite数据库）。
- 缓存策略可采用LRU（Least Recently Used）算法，优化性能。

2.4.2 数据副本和一致性

- 数据副本：创建文件时在不同的物理机器上创建多个副本。
- 一致性：可以选择CAP定理中的一致性和可用性。考虑到可操作性，我们可以选择最终一致性模型来平衡。

2.4.3 多用户支持和文件锁

- 使用文件锁机制来支持并行读写，确保同一时间只有一个用户可以写入特定文件。
- 读写锁（共享锁和排他锁）可以保证读者不会阻塞读者，但写者会阻塞所有其他操作。

2.4.4 访问权限控制

实现基本的访问控制列表（ACL），根据用户身份控制文件访问权限。

2.5 本地测试和验证

使用 python 的 `concurrent.futures` 开发多线程以模拟多用户对分布式文件系统的并发访问。进行相关测试。

在 python 提供的 `unittest` 测试框架中对文件锁的功能进行了测试。

3 实现细节

下面是整个项目的流程图说明：

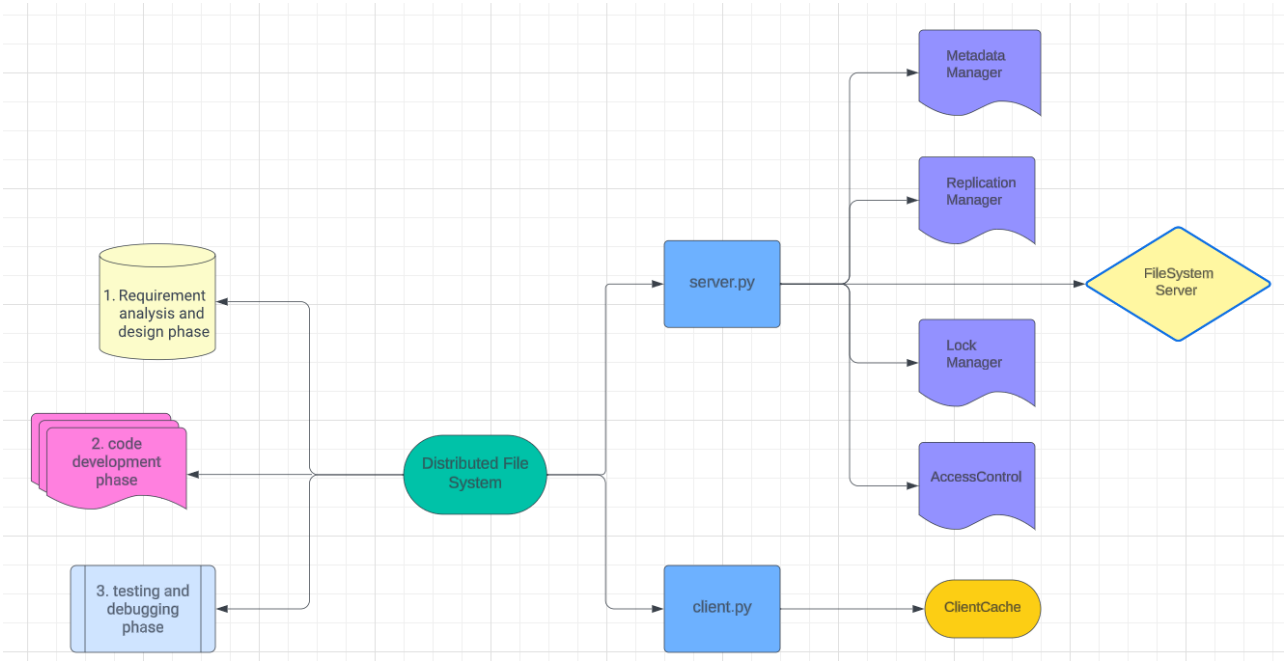


图 1 分布式文件系统项目的流程图

按照Top-Down的设计思想，我们从高层设计开始，逐步细化到具体的实现。现在我们已经有了一个基础的框架，下面我们将分步骤对每项功能进行细化。

其中，服务器端负责文件的存储、元数据管理、副本控制、锁定机制和访问控制，而客户端则负责维护本地缓存。对应的 Top-Down 设计的类如下：

- **服务器端 (`server.py`):**
  - MetadataManager
  - **FileSystemServer**

  - ReplicationManager
  - LockManager
  - AccessControl

- **客户端 (`client.py`):**
  - ClientCache

在这个课程设计中，我们将 FileSystemServer 类作为集成各种文件系统功能的类进行实现，因此其实现在最后给出。首先是对实现文件系统的各种机制的讨论。

3.1 设计文件存储策略

在服务器端，需要一个高效的方式来存储和检索文件。我们可以使用文件系统来存储实际的文件，同时维护一个元数据索引，记录每个文件的信息，如其副本位置、大小、创建时间等。

细化点:

- 创建一个元数据结构。
- 实现文件的读写删除到磁盘。
- 对于每个文件操作，更新元数据。

我们将定义 `MetadataManager` 并将其集成到 `FileSystemServer` 类中，以便在文件创建时创建元数据。完整代码实现如下：

```python
# server.py

import os
import grpc
from concurrent import futures
import file_system_pb2
import file_system_pb2_grpc
import json
import threading
import datetime

# 管理文件的元数据
class MetadataManager:
    def __init__(self):
        self.files_metadata = {}
        self.metadata_lock = threading.Lock()
        self.metadata_file = "metadata.json"
        self.load_metadata()

    def load_metadata(self):
        # 从 json 文件中以字典形式加载元数据
        if os.path.exists(self.metadata_file):
            with open(self.metadata_file, 'r') as f:
                self.files_metadata = json.load(f)

    def save_metadata(self):
```

```
        # 以字典形式将元数据保存为 json 文件
        with open(self.metadata_file, 'w') as f:
            json.dump(self.files_metadata, f, indent=4)

    def create_file_metadata(self, filename, replica_info):
        # 为新文件创建并保存一个元数据
        with self.metadata_lock:
            if filename not in self.files_metadata:
                self.files_metadata[filename] = {
                    'size': 0,
                    'created_at': str(datetime.datetime.now()),
                    'modified_at': str(datetime.datetime.now()),
                    # 记录文件存储的副本信息
                    'replicas': replica_info
                }
                self.save_metadata()
                return True
            else:
                return False

    def get_file_metadata(self, filename):
        # 获取已创建文件的元数据
        with self.metadata_lock:
            return self.files_metadata.get(filename, None)

    def update_file_metadata(self, filename, metadata):
        # 更新已创建文件的元数据
        with self.metadata_lock:
            if filename in self.files_metadata:
                self.files_metadata[filename].update(metadata)
                self.files_metadata[filename]['modified_at'] =
str(datetime.datetime.now())
                self.save_metadata()
                return True
            else:
                return False

    def delete_file_metadata(self, filename):
        # 更新删除已创建文件的元数据
        with self.metadata_lock:
            if filename in self.files_metadata:
                del self.files_metadata[filename]
                self.save_metadata()
                return True
            else:
                return False
```

3.2 实现副本管理和一致性

我们需要确保文件系统中的副本是一致的。这通常意味着实现一个副本控制协议，比如最终一致性模型。

细化点:

- 在写操作发生时，更新所有副本。
- 如果选择最终一致性，可以实现一个后台进程定期同步副本状态。

完整代码实现如下:

```
class ReplicationManager:
    def __init__(self, replica_dirs):
        # 副本所在的目录
        self.replica_dirs = replica_dirs
        for dir in self.replica_dirs:
            if not os.path.exists(dir):
                os.makedirs(dir)

    def get_replica_info(self, filename):
        # 获取副本信息，例如存储副本的路径列表
        replica_paths = [os.path.join(replica_dir, filename) for
replica_dir in self.replica_dirs]
        return replica_paths

    def replicate_data(self, filename, data):
        # 将副本复制到所有节点上，实现一致性
        for replica_dir in self.replica_dirs:
            replica_path = os.path.join(replica_dir, filename)
            with open(replica_path, 'wb') as replica_file:
                replica_file.write(data)
        return True

    def delete_replica(self, filename):
        # 从所有副本节点中删除一个文件
        for replica_dir in self.replica_dirs:
            replica_path = os.path.join(replica_dir, filename)
            if os.path.exists(replica_path):
                os.remove(replica_path)
        return True

    def synchronize_replicas(self, filename):
        # 将文件同步到所有的副本节点中
        # 构建主副本路径
        primary_path = os.path.join(self.replica_dirs[0], filename)
        if os.path.exists(primary_path):
            with open(primary_path, 'rb') as primary_file:
                # 读取主副本文件的数据，并将其复制到所有副本目录中
                data = primary_file.read()
                for replica_dir in self.replica_dirs[1:]:
```

```
                    # 构建副本路径，打开副本文件并写入数据
                    replica_path = os.path.join(replica_dir, filename)
                    with open(replica_path, 'wb') as replica_file:
                        replica_file.write(data)
        return True
```

3.3 文件锁定机制

为了支持多用户并发访问，我们需要在服务器端实现一个文件锁定机制。

细化点:

- 实现锁管理器，支持读写锁。
- 在文件操作前检查并设置锁状态。
- 操作完成后释放锁。

完整代码实现如下:

```
class LockManager:
    def __init__(self):
        # 初始化一个字典来保存文件锁的状态
        self.locks = {}
        self.locks_lock = threading.Lock()  # 保护对locks字典的访问

    def _get_lock(self, filename):
        with self.locks_lock:
            if filename not in self.locks:
                # 对于每个文件，我们使用一个读写锁，允许多个线程同时读取
                self.locks[filename] = threading.RLock()
            return self.locks[filename]

    def acquire_read_lock(self, filename):
        # 获取读锁，如果已经有读锁，则增加计数
        file_lock = self._get_lock(filename)
        file_lock.acquire()

    def acquire_write_lock(self, filename):
        # 获取写锁，如果已经有读锁或写锁，则阻塞直到锁释放
        file_lock = self._get_lock(filename)
        file_lock.acquire()

    def release_lock(self, filename):
        # 释放读或写锁，减少计数
        file_lock = self._get_lock(filename)
        if file_lock._is_owned():  # 检查当前线程是否拥有锁
            file_lock.release()
```

3.4 访问控制机制

访问控制确保用户只能访问他们有权限的文件。

细化点:

- 实现访问控制列表（ACL）。
- 在文件操作前检查用户权限。

完整代码实现如下:

```python
class AccessControl:
    def __init__(self):
        # 初始化一个字典来保存用户对文件的权限
        self.permissions = {'testfile.txt': {'hhj': ['read']}}

    def set_permission(self, user, filename, operation):
        # 设置用户对文件的操作权限
        if filename not in self.permissions:
            self.permissions[filename] = {}
        self.permissions[filename][user] = operation

    def check_permission(self, user, filename, operation):
        # 检查用户是否对文件具有操作权限
        print(self.permissions)
        file_permissions = self.permissions.get(filename, {})
        user_permissions = file_permissions.get(user, [])
        return operation in user_permissions
```

3.5 集成上述功能实现分布式文件系统

实现文件的创建、读取、写入和删除操作。每个操作都应该首先与元数据交互，然后进行实际的文件系统操作。

细化点:

- 在文件**创建**时，先获取文件写锁，选择存储位置并创建文件副本，最后释放写锁。
- 在文件**读取**时，先通过访问控制列表查看用户权限，若有权限则检索并返回文件内容。
- 在文件**写入**时，先获取文件写锁，更新文件内容并维护副本一致性，最后释放写锁。
- 在文件**删除**时，删除所有副本并更新元数据。

完整代码实现如下:

```python
class FileSystemServicer(file_system_pb2_grpc.FileSystemServicer):
    def __init__(self, storage_path="storage"):
        # 定义文件存储的目录
        self.storage_path = storage_path
        if not os.path.exists(self.storage_path):
            os.makedirs(self.storage_path)

        # 1. 文件元数据管理器
        self.metadata_manager = MetadataManager()
```

```
10
11          # 2. 文件副本管理器
12          self.replication_manager = ReplicationManager(
13              ['replica1', 'replica2', 'replica3'])
14
15          # 3. 文件锁定机制
16          self.lock_manager = LockManager()
17
18          # 4. 访问控制列表
19          self.access_control = AccessControl()
20          self.access_control.set_permission('黄河锦', 'testfile.txt',
['read'])
21
22
23      def CreateFile(self, request, context):
24          file_path = os.path.join(self.storage_path, request.filename)
25          # 获取一个写锁
26          self.lock_manager.acquire_write_lock(request.filename)
27          try:
28              if not os.path.exists(file_path):
29                  try:
30                      # 创建一个空文件
31                      open(file_path, 'w').close()
32                      # 将数据扩散到各副本节点
33                      if self.replication_manager.replicate_data
(request.filename, b''):
34                          # 获取副本信息
35                          replica_info = self.replication_manager.
get_replica_info(request.filename)
36                          # 创建文件的元数据，并包括副本信息
37                          self.metadata_manager.create_file_metadata
(request.filename, replica_info)
38                          return file_system_pb2.CreateFileResponse
                                (success=True)
39                      else:
40                          context.set_details('Metadata creation failed.')
41                          context.set_code(grpc.StatusCode.INTERNAL)
42                          return file_system_pb2.CreateFileResponse
                                (success=False)
43                  # 若文件无法正常创建，则抛出I/O异常
44                  except IOError as e:
45                      context.set_details(str(e))
46                      context.set_code(grpc.StatusCode.INTERNAL)
47                      return file_system_pb2.CreateFileResponse
                            (success=False)
48              # 若文件已创建，则不能重复创建，返回false
49              else:
50                  context.set_details('File already exists.')
51                  context.set_code(grpc.StatusCode.ALREADY_EXISTS)
52                  return file_system_pb2.CreateFileResponse(success=False)
```

```
        finally:
            # 确保写锁的释放
            self.lock_manager.release_lock(request.filename)


    def ReadFile(self, request, context):
        user = request.user  # 假设请求中包含了用户信息
        file_path = os.path.join(self.storage_path, request.filename)
        if os.path.exists(file_path):
            # 首先通过ACL检查用户是否有读取文件的权限
            if self.access_control.check_permission(user,
request.filename, 'read'):
                # 若有权限，在读取文件之前需要先获取读锁
                self.lock_manager.acquire_read_lock(request.filename)
                try:
                    with open(file_path, 'rb') as file:
                        data = file.read()
                    return file_system_pb2.ReadFileResponse(data=data)
                except IOError as e:
                    context.set_details(str(e))
                    context.set_code(grpc.StatusCode.INTERNAL)
                    return file_system_pb2.ReadFileResponse(data=b"")
                finally:
                    # 确保最终释放读锁
                    self.lock_manager.release_lock(request.filename)
            else:
                # 用户无读取文件权限
                context.set_details('Permission denied.')
                context.set_code(grpc.StatusCode.PERMISSION_DENIED)
                return file_system_pb2.ReadFileResponse(data=b"")
        else:
            context.set_details('File not found.')
            context.set_code(grpc.StatusCode.NOT_FOUND)
            return file_system_pb2.ReadFileResponse(data=b"")


    def WriteFile(self, request, context):
        file_path = os.path.join(self.storage_path, request.filename)
        # 首先获取一个写锁
        self.lock_manager.acquire_write_lock(request.filename)
        try:
            # 写入数据到主文件
            with open(file_path, 'wb') as file:
                file.write(request.data)
            # 同步数据到各副本节点
            if self.replication_manager.replicate_data(request.filename,
request.data):
                # 获取副本信息
                replica_info = self.replication_manager.get_replica_info
(request.filename)
```

```
                # 更新文件元数据，包括副本信息
                self.metadata_manager.update_file_metadata
(request.filename, {'replicas': replica_info})
                return file_system_pb2.WriteFileResponse(success=True)
        except IOError as e:
            context.set_details(str(e))
            context.set_code(grpc.StatusCode.INTERNAL)
            return file_system_pb2.WriteFileResponse(success=False)
        finally:
            # 确保写锁的释放
            self.lock_manager.release_lock(request.filename)


    def DeleteFile(self, request, context):
        file_path = os.path.join(self.storage_path, request.filename)
        if os.path.exists(file_path):
            try:
                # 删除主文件
                os.remove(file_path)
                # 删除副本文件
                if self.replication_manager.delete_replica
(request.filename):
                    # 更新文件元数据，移除副本信息
                    self.metadata_manager.update_file_metadata
(request.filename, {'replicas': []})
                    return file_system_pb2.DeleteFileResponse
                           (success=True)
            except IOError as e:
                context.set_details(str(e))
                context.set_code(grpc.StatusCode.INTERNAL)
                return file_system_pb2.DeleteFileResponse(success=False)
        else:
            context.set_details('File not found.')
            context.set_code(grpc.StatusCode.NOT_FOUND)
            return file_system_pb2.DeleteFileResponse(success=False)
```

3.6 客户端缓存

客户端缓存可以显著提高读取操作的性能，特别是对于频繁访问的文件。

细化点:

- 实现缓存策略，如LRU缓存。
- 在客户端读取文件时，首先检查本地缓存。
- 在写操作后，根据缓存策略更新缓存。

完整代码实现如下：

```
# client.py

import grpc
```

```python
import file_system_pb2
import file_system_pb2_grpc
from collections import OrderedDict

# 实现 LRU 缓存
class LruCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return None
        else:
            self.cache.move_to_end(key)  # mark as recently used
            return self.cache[key]

    def put(self, key, value):
        self.cache[key] = value
        self.cache.move_to_end(key)
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)  # remove least recently used

# 基于 LRU 缓存实现客户端缓存
class ClientCache:
    def __init__(self, size):
        self.cache = LruCache(size)

    def get_from_cache(self, filename):
        # 如果 cache 中存在文件数据，则从 cache 中读取
        return self.cache.get(filename)

    def update_cache(self, filename, data):
        # 用最新的文件数据更新 cache
        self.cache.put(filename, data)
```

4 运行情况

首先给出 `client.py` 的完整测试代码（可并发访问服务器）：

```python
# 使用客户端缓存
cache = ClientCache(size=3)

# 创建文件
def create_file(stub, filename):
    response = stub.CreateFile(file_system_pb2.CreateFileRequest(filename=filename))
    print("File created:", response.success)
```

```python

# 读取文件
def read_file(stub, filename, user):
    file_data = cache.get_from_cache(filename)
    if file_data is None:
        # 如果文件不在缓存中，则从 server 中读取
        request = file_system_pb2.ReadFileRequest(filename=filename, user=user)
        response = stub.ReadFile(request)
        file_data = response.data
        # 将该文件更新到客户端缓存中
        cache.update_cache(filename, file_data)
    else:
        # 文件在客户端缓存中，直接读取即可
        print(f"Retrieved {filename} from cache.")
    print("File content:", file_data)

# 写入文件
def write_file(stub, filename, data):
    response = stub.WriteFile(file_system_pb2.WriteFileRequest(filename=filename, data=data))
    print("File written:", response.success)

# 删除文件
def delete_file(stub, filename):
    response = stub.DeleteFile(file_system_pb2.DeleteFileRequest(filename=filename))
    print("File deleted:", response.success)

# 运行客户端，进行文件操作
def run():
    with grpc.insecure_channel('127.0.0.1:50053') as channel:
        # 使用 gRPC 进行远程过程调用与 server 通信
        stub = file_system_pb2_grpc.FileSystemStub(channel)

        # 创建一个新的文件，并写入信息
        create_file(stub, "testfile.txt")
        write_file(stub, "testfile.txt", b"Hello SYSU")

        # 读取文件，可以选择不同的用户进行多次读取
        read_file(stub, "testfile.txt", "hhj")
        read_file(stub, "testfile.txt", "黄河锦")

        # 删除文件
        delete_file(stub, "testfile.txt")

if __name__ == '__main__':
    run()
```

4.1 项目整体测试

首先在一个终端中打开服务器：

```
终端:  本地 ×  本地 (2) ×  +  ⌄
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS D:\Desktop\DS_pro> python server.py
Server running on port 50053...
{'testfile.txt': {'hhj': ['read'], '黄河锦': ['read']}}
```

然后在另一个终端打开客户端：

```
终端:  本地 ×  本地 (2) ×  +  ⌄
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS D:\Desktop\DS_pro> python client.py
File created: True
File written: True
File content: b'Hello SYSU'
File deleted: True
PS D:\Desktop\DS_pro>
```

这样，我们便完成了一次客户端对服务器的分布式文件系统的访问。接下来为了进一步展示，我们把 `run` 中的 `delete_file` 文件删除操作暂时去除，观察元数据的存储文件 metadata.json 和各副本的存储情况：

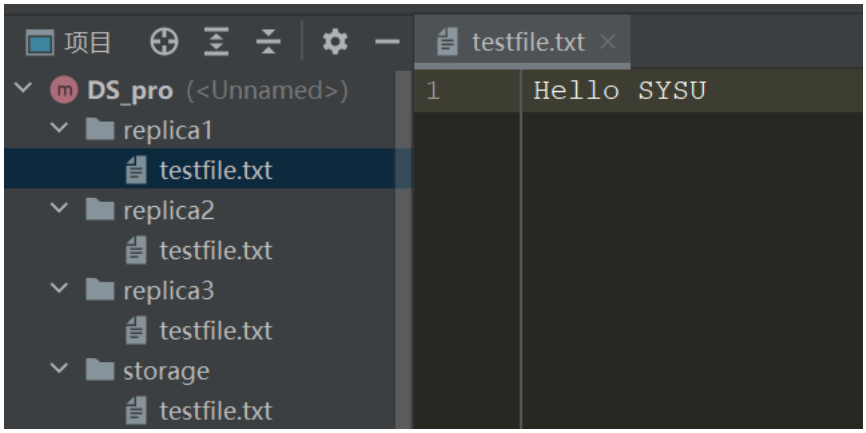
metadata.json 的情况如下：

metadata.json

```json
{
    "testfile.txt": {
        "size": 0,
        "created_at": "2023-12-24 11:44:14.456833",
        "modified_at": "2024-01-09 11:43:45.847519",
        "replicas": [
            "replica1\\testfile.txt",
            "replica2\\testfile.txt",
            "replica3\\testfile.txt"
        ]
    }
}
```

可以看到，文件的元数据管理器 `MetadataManager` 正确地创建了文件的元数据，记录了该文件最初的创建时间，最近一次的修改时间，以及存储在哪些副本的副本信息。

各副本的存储情况如下：



可以看到，我们定义的三个文件副本都正确地存储了文件 testfile.txt ，且数据的一致性通过同步机制得到了保证。

4.2 细化测试

4.2.1 访问控制列表和客户端缓存的测试

从上述实现细节我们可以看到，此时默认的ACL为:`{'testfile.txt': {'hhj': ['read'], '黄河锦': ['read']}}`，因此我们首先对这两个授权用户进行文件读取的测试。具体来说，我们将`run`函数设置如下：

```
def run():
    with grpc.insecure_channel('127.0.0.1:50053') as channel:
        # 使用 gRPC 进行远程过程调用与 server 通信
        stub = file_system_pb2_grpc.FileSystemStub(channel)

        # 创建一个新的文件，并写入信息
        create_file(stub, "testfile.txt")
        write_file(stub, "testfile.txt", b"Hello SYSU")

        # 读取文件，此处选择用户 hhj 和 黄河锦 进行两次读取
        read_file(stub, "testfile.txt", "hhj")
        read_file(stub, "testfile.txt", "黄河锦")

        # 删除文件
        delete_file(stub, "testfile.txt")
```

在客户端执行`run`，运行结果如下：

```
终端:  本地 ×  本地 (2) ×  +
PS D:\Desktop\DS_pro> python client.py
File created: True
File written: True
File content: b'Hello SYSU'
Retrieved testfile.txt from cache.
File content: b'Hello SYSU'
File deleted: True
PS D:\Desktop\DS_pro>
```

可以看到，此时ACL的授权用户可以正常进行文件读取，且第二次的读取使用了客户端的缓存机制，打印出了 **Retrieved testfile.txt from cache** 语句。

接下来，我们更改 `run` 函数的设置，尝试以未授权用户的身份对文件进行读取：

```python
def run():
    with grpc.insecure_channel('127.0.0.1:50053') as channel:
        # ...(同上)

        # 读取文件，此处选择未授权用户 张三 进行读取
        read_file(stub, "testfile.txt", "张三")

        # ...(同上)
```

在客户端执行 `run`，运行结果如下：

```
终端:  本地 ×  本地 (2) ×  +
PS D:\Desktop\DS_pro> python client.py
Traceback (most recent call last):
  File "D:\Desktop\DS_pro\client.py", line 97, in <module>
    run()
  File "D:\Desktop\DS_pro\client.py", line 89, in run
    read_file(stub, "testfile.txt", "张三")
  File "D:\Desktop\DS_pro\client.py", line 56, in read_file
    response = stub.ReadFile(request)
  File "C:\Users\hhj29\AppData\Local\Programs\Python\Python39\lib\site-packages\grpc
    return _end_unary_response_blocking(state, call, False, None)
  File "C:\Users\hhj29\AppData\Local\Programs\Python\Python39\lib\site-packages\grpc
ocking
    raise _InactiveRpcError(state)  # pytype: disable=not-instantiable
grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that terminated with:
        status = StatusCode.PERMISSION_DENIED
        details = "Permission denied."
        debug_error_string = "UNKNOWN:Error received from peer ipv4:127.0.0.1:50053
grpc_status:7, grpc_message:"Permission denied."}"
>
PS D:\Desktop\DS_pro>
```

可以看到，未授权用户张三不能访问目标文件，这说明访问控制机制正确地发挥了作用。

4.2.2 多用户并发访问测试

在模拟多用户访问时，有以下两种不同粒度的方案，先对它们进行简要说明：

1. **开多个终端模拟多用户**

这种方法本质上是使用多进程的方式模拟多用户行为。在实际实现时，需要显式地进行单进程的阻塞以模拟并发场景，且粒度较粗，因此我们不选用这种较为粗糙的方式。

2. **使用多线程模拟多个用户**

这种方法实现了更细粒度的并发访问，且有成熟的 python 线程库支持，因此我们会采用这种方式模拟多用户对分布式文件系统的并发访问。

为方便演示，我们首先对当前用户进行授权，即当前所有用户都可以访问它们创建的文件。首先给出修改后的 `run` 函数的相关代码：

```
import concurrent.futures

# 单个用户的一个完整操作序列，可按需进行解耦
def user_operations(stub, username):
    user_mapping = {
        "hhj": 1,
        "Smith Huang": 2
    }
    default_mapping = 3
    filename = f"testfile_{user_mapping.get(username, default_mapping)}.txt"
    create_file(stub, filename, username)
    content = f"Hello SYSU, {username}"
    write_file(stub, filename, content.encode())
    read_file(stub, filename, username)
    delete_file(stub, filename)

def run_concurrently():
    with grpc.insecure_channel('127.0.0.1:50053') as channel:
        stub = file_system_pb2_grpc.FileSystemStub(channel)
        with concurrent.futures.ThreadPoolExecutor() as executor:
            executor.submit(user_operations, stub, "hhj")
            executor.submit(user_operations, stub, "Smith Huang")
            executor.submit(user_operations, stub, "Zhang San")

if __name__ == '__main__':
    run_concurrently()
```

上面的代码使用了 python 的 `concurrent.futures` 模块，支持 `ThreadPoolExecutor()` 线程池的多线程访问方法。其中，创建文件、写入文件、读取文件并删除文件构成单个用户的一个完整的操作序列。我们模拟 hhj，Smith Huang 和 Zhang San 这三个用户的并发访问，运行结果如下：

```
终端:  本地 ×  本地 (2) ×  +  ⌄
PS D:\Desktop\DS_pro> python client.py
File created by user Smith Huang: True
File created by user Zhang San: True
File created by user hhj: True
File written: True
File written: True
File content: b'Hello SYSU, Smith Huang'
File written: True
File content: b'Hello SYSU, hhj'
File content: b'Hello SYSU, Zhang San'
File deleted: True
File deleted: True
File deleted: True
PS D:\Desktop\DS_pro>
```

可以看到，我们成功实现了三个不同用户对分布式文件系统的并发访问，且运行结果体现了多线程的特点，即先创建文件的用户不一定先读取，它们的操作序列在单个用户来看是有序的，但从全局来看则不然。

- **并发创建和写入同一文件测试**

接下来，我们稍微修改一下上述代码的测试逻辑，让用户 hhj 和 Zhang San 试图创建同一个文件 testfile_1.txt，而用户 Smith Huang 试图创建另一个文件 testfile_2.txt。之后，我们对操作序列进行解耦，分别启动对应三个用户的三个独立的写操作线程，运行结果如下：

```
PS D:\Desktop\DS_pro> python client.py
File testfile_1.txt created by user Zhang San: True
File testfile_2.txt created by user Smith Huang: True
File testfile_1.txt written by user hhj: True
File testfile_2.txt written by user Smith Huang: True
File testfile_2.txt deleted by user Smith Huang: True
File testfile_1.txt deleted by user hhj: True
PS D:\Desktop\DS_pro>
```

可以看到，只有用户 Zhang San 和 Smith Huang 成功进行了两个不同文件的创建，用户 hhj 试图对文件 testfile_1.txt的创建操作失败；此外，只有 hhj 和 Zhang San 成功进行了写入，用户 Zhang San 试图对文件 testfile_1.txt的写入操作失败。上述实验结果说明了对同一个文件的并发创建或写入操作都是冲突（互斥）的，这与我们的实验预期相符。

- **并发读取同一文件测试**

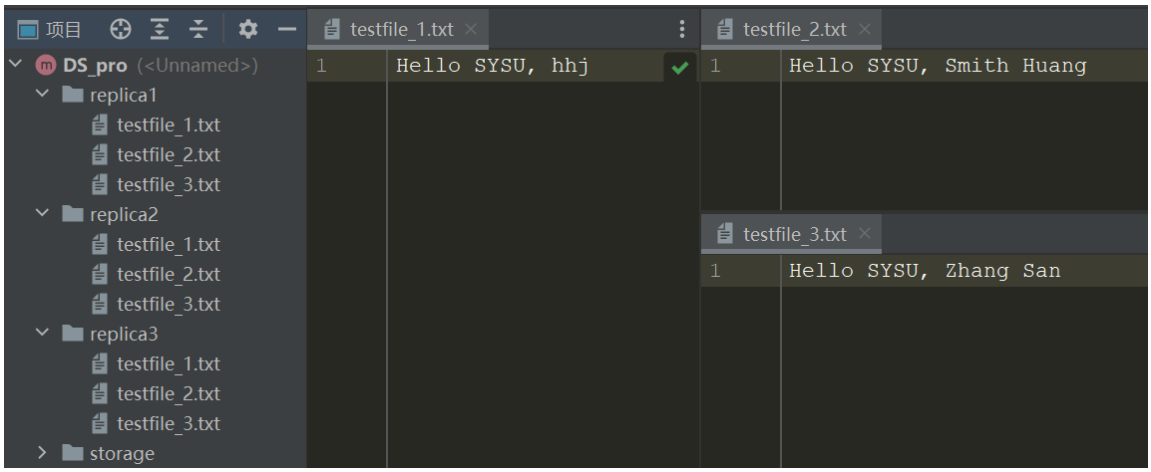
创建文件的逻辑和上面一样。我们继续对操作序列进行解耦，让用户 hhj 和 Zhang San 试图并发读取同一文件 testfile_1.txt，用户 Smith Huang 试图读取文件 testfile_2.txt，运行结果如下：

```
终端:  本地 ×  本地 (2) ×  本地 (3) ×  +
PS D:\Desktop\DS_pro> python client.py
File testfile_1.txt created by user Zhang San: True
File testfile_2.txt created by user Smith Huang: True
File testfile_1.txt written by user Zhang San: True
File testfile_2.txt written by user Smith Huang: True
File testfile_1.txt read by hhj, content: b'Hello SYSU, Zhang San'
File testfile_2.txt read by Smith Huang, content: b'Hello SYSU, Smith Huang'
File testfile_1.txt read by Zhang San, content: b'Hello SYSU, Zhang San'
File testfile_1.txt deleted by user hhj: True
File testfile_2.txt deleted by user Smith Huang: True
```

可以看到，文件被 Zhang San 和 Smith Huang 创建并写入，但最先读取到的用户却是 hhj，且三个用户均能正确读取目标文件，这说明对同一文件或不同文件的并发读都是允许的，这和我们的实验逻辑也是相符的。

- **副本一致性**

此外，副本的数据一致性可以注释掉 `delete_file` 后直接在文件目录下进行查看：



可以看到，三个不同的副本节点存储的文件数据信息是相同的。

4.2.3 unittest 测试框架

除了上述使用 python 的 `concurrent.futures` 模块模拟多线程的测试方法，我还写了一个 test.lock.py 文件用于更为精细和自动化地测试文件锁机制的功能，主要测试点包括对文件的并发读和并发写。`unittest` 是 Python 中的一个测试框架，可以根据需求编写和运行单元测试。

由于篇幅限制，此处缺省 test_lock.py 的代码展示（已放在源代码中）。两个测试函数分别为 `test_concurrent_readers` 和 `test_write_requires_exclusivity`。直接给出测试结果：

```
终端:  本地 ×  本地 (2) ×  本地 (3) ×  +
PS D:\Desktop\DS_pro> python .\test_lock.py
..
----------------------------------------------------------------------
Ran 2 tests in 1.219s

OK
```

可以看到，我们设计的分布式文件系统可以通过该测试。

5 遇到的问题

本项目的代码开发阶段总体上进展还算顺利，在理清分布式文件系统需要实现的机制后，选择相应的策略进行细化实现即可。在进入测试与调试阶段后，我在刚开始的时候遇到了如何模拟多用户行为的问题。后来经过查阅网上的资料，发现可以使用 python 的 `concurrent.futures` 模块模拟多线程进行相关测试。

实际上我尝试了多种测试方法，最后决定选用这种多线程的方式，主要是考虑到其具有细粒度和可按需动态编程的灵活性。此外，在模拟多用户的多线程并发测试中，我发现线程之间存在的竞争条件可能会导致文件创建和删除的异常。因此，我引入了文件锁机制，确保了线程安全。并在 `unittest` 测试框架中对文件锁的功能进行了测试，可以正确通过相应的测试点。

6 总结

在现代计算环境中，分布式文件系统是至关重要的组件，它允许跨多台计算机存储和访问数据，从而提供了高可靠性、可伸缩性和并发访问。本课程设计项目的目标是设计并实现一个基本的分布式文件系统，支持文件的创建、读取、写入和删除操作，并能够处理并发请求。

在**需求分析和设计阶段**，我首先确定了系统要支持的基本操作：创建（CreateFile）、读取（ReadFile）、写入（WriteFile）和删除（DeleteFile）文件。并设计了系统的基本架构，包括文件服务器（FileSystemServicer），元数据管理、复制管理和锁管理等组件。

在**代码开发阶段**，我使用Python语言和gRPC框架进行了编程实现。我设计了元数据结构，以便于管理文件信息；实现了文件操作的逻辑，确保了文件数据的一致性和可访问性；集成了锁管理机制，以处理并发访问和修改文件的情况。

在**测试与调试阶段**，我尤其关注多用户下的并发访问场景，利用 python 提供的多线程机制对不同的读写情形均进行了相应的测试。此外，我还学习了 unittest 并编写了相应的测试框架用于文件锁机制的测试。

通过本项目，我成功地实现了一个基本的分布式文件系统，该系统能够处理基本的文件操作和并发请求。尽管系统还不够成熟，不能用于生产环境，但它提供了一个学习和实践分布式系统概念的有效平台。在这个过程中，我了解到设计一个健壮的分布式系统需要细致的规划和对并发编程的深刻理解。我还学到了如何使用单元测试来验证系统的正确性和稳定性。

总的来说，这个课程设计锻炼了我的软件设计和编程实践的能力，也为我未来在计算机科学和软件工程领域关于分布式系统的学习和研究打下了坚实的基础，使我受益匪浅。