大作业：分布式文件系统

姓名：陈浚铭
班级：20级网络空间安全
学号：20337021

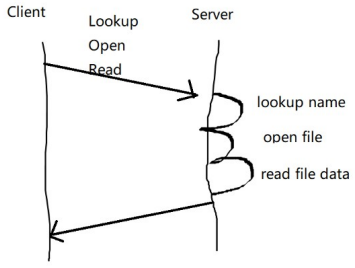
1. 问题描述

对于分布式文件系统，多个客户端通过连接服务器来读/写文件系统的文件，当中会遇到一致性问题，包括缓存一致性(Cache coherence and consistency)，最终一致性问题（eventual consistency），以及读者写者问题(Reader writer's problem)的同步问题(synchronization)等。这是我们在分布式文件系统需要解决的问题。通过使用 grpc 以及 c++语言，实现了 client-server 架构的分布式文件系统。当中参考了 Distributed Systems Principles and Paradigms 这本书关于一致性于副本，同步，分布式文件系统这些重点知识的章。**我并没有参考老师给的参考实现，代码都是我自己写的。**

2. 解决方案

2.1 介绍

我实现的分布式文件系统是 Client-Server 架构，当中 Server 是储存了文件系统，并且负责处理 Client 的读，写等等的请求。通过使用 RPC 的方法，来处理客服端的请求，如下图：



有了 RPC 作为通信协议为基础，我们需要考虑到多个客户端读/写文件的机制，也就是解决读者写者同步问题。我们实现的文件系统是可以**每一瞬间**让多个读者同时读取同一个文件的，而**每一瞬间**只能够有一个写者写文件。对于读锁，我们使用 lease 的方法， 客户端在要读取文件的时候，尝试获取(acquire)读锁，在获取了以后，在给定一个时段以后，会把锁释放(release)了；而对于写锁，写者会尝试获取写锁，而在他写完文件后，会释放写锁。我是使用 pthread 的 rwlock 来实现处理读写问题的机制，当中服务器有查找表 std::map<std::string, pthread_rwlock_t> rwlock_map。注意到我们是**对于每一个文件实现锁机制。**

另外，我们还实现了一个无效通知（invalidation notice)的查找表 std::map<int, std::map<std::string, bool> > invalidationNotice，来记录**每一个客服端**之前所读取的文件的缓存是否最新还是过时，而为了识别每一个客服端，我们在客服端第一次连接上服务器的时候就需要给客服端一个唯一标识符 id，并使用 client_list 来储存他们的 id。

最后，我们还实现了文件缓存，以加快读取文件的速度

，减小客服端需要等待连接服务器的延迟及网络带宽的代价。我们实现的文件缓存是基于FIFO队列的。

2.2 cache机制和一致性

在这个文件系统， 我们是使用文件缓存作为**客服端驱动副本**(client-initiated replica)。每当我们需要读取文件的时候， 我们先从文件看看是否有缓存了该文件。如果是，那么我们向服务器请求一个invalidation notice。如果我们得到的信息是文件缓存的内容是invalid，那么我们需要从服务器请求文件的新内容，并且更新文件缓存的为新的文件内容；如果信息是文件缓存的内容是valid，那么我们可以直接从文件缓存读取。如果文件不在文件缓存，那么我们直接从服务器读取文件内容。

在客户端读文件的伪代码如下:

```
function read_file(filename, clientID)
  if filename not in cache
    buffer <- Call RPC readFile(filename)
              {
                acquireReadLock
                return read_file_data(filename)
              }
    print buffer
    sleep(10s)
    Insert_to_cache(filename, buffer)
    Call RPC releaseReadLock(filename)
  else
   valid <- Call RPC RequestInvalidation(filename, clientID)
   if valid == true
     Call RPC AcquireReadLock(filename)
     buffer <- Cache_read(filename)
     print buffer
     sleep(10s)
     Call RPC releaseReadLock(filename)
   else
    buffer <- Call RPC readFile(filename)
              {
                acquireReadLock
                return read_file_data(filename)
              }
    print(buffer)
    sleep(10s)
    update_Cache(filename, buffer)
    Call RPC releaseReadLock(filename)
```

因此，我能够是实现每一次读文件的时候，保证获取的内容是正确的。注意到我使用的协议是 update propagation 的方法来更新缓存(invalidation protocol),也就是让客户端自己主动去 pull request 更新信息(invalidation notice)。Invalidation protocol 的优点在于它使用小的网络带宽。 注意到使用 update propagation 的方法比较适用于当读/写比例的低，而我们假设这个项目是符合这个条件的。 上述。代码除了保证获取的文件内容是最新之外，同时也保证客户端于客户端之间符合缓冲一致性 cache coherence。

之后，我们需要实现写操作，对于写的伪代码跟读的伪代码框格相似。写操作在写完文件内容的时候，会更新服务器中该文件的内容，告诉服务器曾经该文件为 invalid，并同时在写进去自己的 cache，从而实现了 cache write through。

写操作的伪代码如下：

```
function write_file(filename, clientID)
  if filename not in cache
    buffer <- Call RPC get_file_data_to_write(filename, clientID)
              {
                acquireWriteLock
                for valid in InvalidationNotice:
                  valid = 0
                InvalidationNotice[clientID] = 1

                return read_file_data(filename)
              }
    print buffer
    writeBuffer <- input
    Call RPC releaseWriteLockAndWriteToFile(filename, writeBuffer)
    Insert_to_cache(filename, writeBuffer)
  else
   valid <- Call RPC RequestInvalidation(filename, clientID)
   if valid == true
     Call RPC AcquireWriteLock(filename)
     buffer <- Cache_Read(filename)
     print buffer
     writeBuffer <- input
     Call RPC releaseWriteLockAndWriteToFile(filename, writeBuffer)
   else
    buffer <- Call RPC get_file_data_to_write(filename, clientID)
              {
                acquireWriteLock
                for valid in InvalidationNotice:
                  valid = 0
                InvalidationNotice[clientID] = 1

                return read_file_data(filename)
              }
    print(buffer)
    writeBuffer <- input

    Call RPC releaseWriteLockAndWriteToFile(filename, writeBuffer)
    update_Cache(filename, writeBuffer)
```

最后，注意通过这个读写机制是符合**最终一致性**的。最终一致性能够容许高程度的不一致性。如果对于一个数据项(data item)在很久时间后没有进行更新，那么最终（eventually),所有对于该数据项的访问都会变得一致。在我实现的读机制，如果我们通过无效通知，知道文件缓存是旧（stale)的，那么我们就再从服务器读取新的文件内容，更新文件缓存，并我能够解决冲突(conflict resolution)。注意到我们这个解决冲突方法是非常简单的

因为我们再**同一时间**只是有一个写者，因此我们的读操作总是读取最后写者的写的文件内容，而在实际的分布式系统当中，可能需要考虑同时有多个写者的问题。

3. 实验代码

我以 **root** 作为文件系统的 root 文件夹

dfs.proto

需要实现的 RPC

```
service DFS
{
    rpc connectServer(ClientAddress) returns(ClientID) {}

    // check if valid
    rpc pullInvalidation(FileRequest) returns(Invalidation) {}

    // read file add it cache and acquire lock set valid
    rpc readFile(FileRequest) returns(FileHandle) {}
    // read lock
    rpc acquireReadLock(Filename) returns(Response) {}
    rpc releaseReadLock(Filename) returns(Response) {}

    // get write lock and file data set valid, set others invalid
    rpc getFileDataToWrite(FileRequest) returns(FileHandle) {}

    // get write lock and set valid other invalid
    rpc acquireWriteLock(FileRequest) returns(Response) {}
    // write into cache and server
    rpc writeFileAndReleaseLock(FileHandle) returns(Response) {}
    rpc createFile(FileHandle) returns(Response) {}
    rpc deleteFile(Filename) returns(Response) {}
    rpc list(DirectoryPath) returns (FileList) {}
}
```

重要的 message 类型

```
message FileRequest
{
    int32 id = 1;
    string fname = 2;
}
message ClientID
{
    int32 id = 1;
}

message Invalidation
{
        int32 is_valid = 1;
}

message Response
{
        int32 success = 1;
}
```

```
message Filename
{
        string fname = 1;
}

message FileHandle
{
        string fname = 1;
        string buffer = 2;
        int32 access = 3;
}

message ClientAddress
{
        string ip_address = 1;
        int32 port = 2;
}
```

FileRequest: 分别提供文件名和客服端标识符的信息

ClientID: 客户端标识符，在连接服务器的时候，会返回该值到客户端

Invalidation: 无效通知

Response: 响应码： 它的值可以是 FAIL = 0, 代表失败，SUCCESS = 1 代表成功

Filename: 文件名（在**不需要提供识别符**的时候使用）

FileHandle: 文件名，文件内容，以及响应码（跟 response 一样）

ClientAddress: 我们没有在这里使用到，但是可以用来提供客户端的 ip 地址和端口

dfs_server.cc

我们实现了 DFS_Server 类，它有元素：

```
std::map<std::string, pthread_rwlock_t> rwlock_map; // 文件锁查找表
int client_address_list[5];  // client 的端口表 （我们没有使用到
int client_id_list[5];     // client标识符的列表
std::map<int, std::map<std::string, bool> > invalidationNotice; // 无效通知查找表
int num_of_clients;  // 客服端的数量
```

DFS_Server 类的 RPC 函数：

连接服务器

```
Status connectServer(ServerContext* context, const ClientAddress *client_addr, ClientID *clientID) override
{
  if(num_of_clients == MAX_CLIENTS)
  {
      std::cout << "Connect to server failure, already reached max clients\n";
      clientID->set_id(-1);
  }
  else
  {
      client_address_list[num_of_clients] = client_addr->port();

      // client id
      int unique_id = num_of_clients+1;
      client_id_list[num_of_clients] = unique_id;
      clientID->set_id(unique_id);

      invalidationNotice[unique_id] = std::map<std::string, bool>();
      num_of_clients++;
  }

  return Status::OK;
}
```

请求 Invalidation notice

```
Status pullInvalidation(ServerContext *context, const FileRequest *fileRequest, Invalidation *invalidation) override
{
  int is_valid = invalidationNotice[fileRequest->id()][fileRequest->fname()];
  invalidation->set_is_valid(is_valid);

  return Status::OK;
}
```

列举文件系统的文件

```
Status list(ServerContext *context, const DirectoryPath *directoryPath, FileList *fileList) override
{

    struct dirent *pDirent;
    DIR *pDir;
    DIR *dr;
    pDir = opendir(filesystem_rootpath);
    std::string file_list;
    while((pDirent = readdir(pDir)) != NULL)
    {
      file_list += pDirent->d_name;
      file_list += ' ';
    }
    closedir(pDir);
    fileList->set_filelist(file_list);
    return Status::OK;
}
```

获取写锁并读取文件内容以写：

```
Status getFileDataToWrite(ServerContext* context, const FileRequest *fileRequest, FileHandle *fileHandle) override
{
  std::string filePath = convertFilePath(fileRequest->fname());
  auto search = rwlock_map.find(fileRequest->fname());
  if( search != rwlock_map.end())
  {

      int rc;
      int count =0;
      std::cout <<"Entered thread, getting write lock for file\n";

      // attempt to get write lock
  retry_write_lock:
      rc = pthread_rwlock_trywrlock(&rwlock_map[fileRequest->fname()]);

      if(rc == EBUSY)
      {
        if(count >= 5)
        {
          fileHandle->set_access(FAIL);
          return Status::OK;
        }
        std::cout << "Lock is busy, do sth in the meantime.\n";
        ++count;
        sleep(1);
        goto retry_write_lock;
      } else if(rc != 0)
      {
        std::cout << "Error: \n";
      }

      // invalidate files for all clientID
      for(auto it = invalidationNotice.begin(); it != invalidationNotice.end(); it++)
      {
        if(it->second.find(fileRequest->fname()) != it->second.end())
        {
          it->second[fileRequest->fname()] = 0;
        }
      }
      invalidationNotice[fileRequest->id()][fileRequest->fname()] = 1;
      // read file data
      ifstream fileToRead(filePath);
      std::stringstream strStream;
      strStream << fileToRead.rdbuf();
      fileToRead.close();
      fileHandle->set_fname(fileRequest->fname());
      fileHandle->set_buffer(strStream.str());
      fileHandle->set_access(SUCCESS);

  }
  else
  {
    fileHandle->set_access(FAIL_GET_FILE);
  }
  return Status::OK;
}
```

另外还有读文件函数 readFile, 代码很类似，不同在于它修改对应 filename 和 clientID 的 InvalidationNotice 项为 1，也就是 valid, 而且它是获取读锁。

之后，还有两个函数 acquireWriteLock() 和 acquireReadLock(), 这两个函数只是分别获取写和读锁。

写文件以及释放写锁的函数：我们在调用这个 RPC 的时候，

会以文件名和写入的文件内容作为输入。

```
Status writeFileAndReleaseLock(ServerContext* context, const FileHandle *fileHandle, Response *response) override
{
  std::string writeFilePath = convertFilePath(fileHandle->fname());
  ofstream writeFile(writeFilePath);
  writeFile << fileHandle->buffer();
  writeFile.close();

  // release write lock
  auto search = rwlock_map.find(fileHandle->fname());

  if(search != rwlock_map.end())
  {
    int rc = pthread_rwlock_unlock(&rwlock_map[fileHandle->fname()]);

    if(rc != 0)
      std::cout <<"Error: "<< rc<<std::endl;
    // pthread_rwlock_unlock(&rwlock_map[std::string("poem.txt")]);
    std::cout << "Unlock the write lock for file \n";

    response->set_success(1);
  }
  return Status::OK;
}
```

另外还有 releaseReadLock 函数，它的作用就是释放读锁。

删除和创建文件的函数：

```
Status deleteFile(ServerContext* context, const Filename *filename, Response *response) override
{
  std::string filePath = convertFilePath(filename->fname());
  auto search = rwlock_map.find(filename->fname());
  if( search != rwlock_map.end())
  {
    std::cout <<"Entered thread, getting write lock for file\n";
    int rc = pthread_rwlock_trywrlock(&rwlock_map[filename->fname()]);
    if(rc == EBUSY)
    {
      std::cout << "Failed to get write thread.\n";
      response->set_success(FAIL);
      return Status::OK;
    }
    remove(filePath.c_str());
    std::cout << "Unlock the write lock for file.\n";
    pthread_rwlock_unlock(&rwlock_map[filename->fname()]);
    std::cout << "rwlock destroyed for file.\n";
    pthread_rwlock_destroy(&rwlock_map[filename->fname()]);
    response->set_success(SUCCESS);
  }
  else
  {
    response->set_success(FAIL_GET_FILE);
  }
  return Status::OK;
}
Status createFile(ServerContext* context, const FileHandle *fileHandle, Response *response) override
{
  char filePath[50];
  strcpy(filePath,"/home/jimchan/Desktop/myDFS/root/");

  strcat(filePath, fileHandle->fname().c_str());


    std::ofstream outfile(filePath);
    outfile << fileHandle->buffer();
    response->set_success(SUCCESS);
    rwlock_map[fileHandle->fname()] =  PTHREAD_RWLOCK_INITIALIZER;
    outfile.close();
    return Status::OK;

}
```

dfs_client.cc

首先实现了 FileCacheQueue 这个类，他就是我们的文件缓存使用 FIFO 队列机制，我在这里不显示代码因为不是重点。但是重要的函数有 readFileCache(尝试读取对应文件名的内容), replaceFileCache(更新文件名缓存的内容), enqueue 和 dequeue (用来插入 cache 项)

之后，我们实现了 DFSClient 类，用来调用 DFS_Server 的 RPC 函数。当中在 **private 访问修饰符中的函数**直接调用 RPC，而 **public 访问修饰符的函数调用 private 的函数**。

最后，在 main 函数，我们对于进一步以命令行的方式作为输入输出，调用 DFSClient 的函数。当中对于读/写的代码就对应了上面的伪代码，也就是实现了**最终一致性和缓存一致性**的重要的代码。

读文件：

```
// input filename
std::cout << "Enter filename to read: ";

char input_filename[10];
scanf("%s", input_filename);
std::string filename(input_filename);
// attempt to get file from cache
char outBuffer[5000];
enum QUERY_TYPE cache_result = fCache.readFileCache(filename, outBuffer);
if(cache_result == HIT)
{
  if(client.pullInvalidation(filename, unique_id) == 1) // valid
  {
    if(client.acquireReadLock(filename) == SUCCESS)
    {
      std::cout << "File data (cache hit):\n";
      std::cout << outBuffer;
      std::cout << std::endl;
      sleep(5);
      client.releaseReadLock(filename);
    }
  }
  else // cache is invalid, refresh cache
  {
    std::cout << "# Cache is dirty.\n";
    FileHandle fileHandle;
    if(client.readFile(filename, unique_id, &fileHandle) == SUCCESS)
    {
      std::cout << std::endl;
      sleep(5);
      client.releaseReadLock(filename);
      fCache.replaceFileCache(filename, const_cast<char *>(fileHandle.buffer().c_str()));
    }
  }
}
else // not in cache
{
  FileHandle fileHandle;
  if(client.readFile(filename, unique_id, &fileHandle)==SUCCESS)

    sleep(5);
    client.releaseReadLock(filename);

    // write to cache (adding entry)
    if(fCache.isFull())
      fCache.dequeue();
    // add entry
    file_cache entry;
    set_fileCache(&entry, fileHandle.fname(),const_cast<char *>(fileHandle.buffer().c_str()));
    entry.is_valid = 1;
    fCache.enqueue(entry);

}
```

写文件：

```
std::cout << "Enter filename to write: ";
std::string filename;
std::cin.ignore();
std::cin >> filename;

char outBuffer[5000];
enum QUERY_TYPE cache_result = fCache.readFileCache(filename, outBuffer);
if(cache_result == HIT)
{
  if(client.pullInvalidation(filename, unique_id) == 1) // valid
  {
    if(client.acquireWriteLock(filename, unique_id) == SUCCESS)
    {
      std::cout << "File data (cache hit):\n";
      std::cout << outBuffer;
      std::cout << std::endl;
      char writeBuffer[5000];
      std::cout << "Enter text to write to file(no longer than 5000 chars):\n";
      std::cin.ignore();
      scanf("%4999[^\n]", writeBuffer);
      client.writeFileAndReleaseLock(filename, std::string(writeBuffer));
      fCache.replaceFileCache(filename, writeBuffer);
    }
  }
  else // cache is invalid, refresh cache
  {
    FileHandle fileHandle;
    if(client.getFileDataToWrite(filename, unique_id) == SUCCESS)
    {
      char writeBuffer[5000];
      std::cout << "Enter text to write to file:\n";
      std::cin.ignore();
      scanf("%4999[^\n]", writeBuffer);
      client.writeFileAndReleaseLock(filename, std::string(writeBuffer));
      fCache.replaceFileCache(filename, writeBuffer);
    }
  }
}
else // not in cache
{
  FileHandle fileHandle;
  if(client.getFileDataToWrite(filename, unique_id))
  {
    char writeBuffer[5000];
    std::cout << "Enter text to write to file:\n";
    std::cin.ignore();
    scanf("%4999[^\n]", writeBuffer);
    client.writeFileAndReleaseLock(filename, std::string(writeBuffer));
    // write to cache (adding entry)
    if(fCache.isFull())
      fCache.dequeue();
    // add entry
    file_cache entry;
    set_fileCache(&entry, filename, writeBuffer);
    fCache.enqueue(entry);
  }
}
```

4. 实验结果

文件系统功能：展示了 list, delete file, create file, read file 的功能

缓冲一致性：在这个视频展示了在 client1 读取文件 poem2.txt，然后在 client2 写文件 poem2.txt，最后再在 client1 读取 poem2.txt 新的文件内容

多用户互动：展示了多用户读取同一个文件，以及一个用户先写了一个文件，然后其他用户去读取该文件的内容

5. 遇到的难问题以及解决方案

在实现文件系统的时候，我考虑到如何实现缓存一致性的问题以及同步的问题，当中想到使用查找表来实现对于每一个用户读过的文件的无效通知以及对应每一个文件的读写锁。另外，在实现文件缓存，想到操作系统使用的 FIFO 页替换算法作为灵感来实现。注意到我实现的文件缓存可能更真实有差别。

之后，对于服务器和客服端之间的通信，我想到如何让客户端获取无效通知。我第一个想法是让服务器主动去发送给所有客户端无效通知，也就是 multicasting。但是服务器是被动的角色。经过跟老师的讨论如何解决问题后，得到了两个解决方案，分别是带外和带内的方法。带外的方法就是在 RPC 以外的端口，创建另一个端口，负责应用层通信，该端口把客服端当作服务器，把服务器当作客服端，

从而可以让服务器主动跟客服端通信，并传递无效通知。带内的方法就是在使用 RPC 通信中，传递无效通知。我是用后者的方法，我们先读取文件缓存，然后通过请求无效通知，知道缓存中的文件内容是否最新。这种机制是 pull-based protocol， 也就是 client-based protocol。当 read-to-update 比例低的时候，pull-based protocol 是很有效率的。该机制的一个缺点是，在缓存缺失的情况，响应时间就会增加。

6. 未来扩展：

- 实现中心化的同步方案