云同步文件系统-设计文档

陶天骅 2017010255 陈张萌 2017013678

概述

目前的云盘等云同步方式有不少不便之处。当一个用户需要同时在多设备上利用云同步功能进行文件协作时，需要下载云盘软件，再手动将文件进行上传、下载，当文件在某一个设备有修改时，还需要对云盘当中旧的文件版本进行删除并将修改后的文件重新上传，非常繁琐且不便。

针对这个问题，Apple公司开发了iCloud，支持所有的苹果设备进行自动的文件共享：在同一个Apple账号登陆的任何一个设备，对文件进行修改，都会在联网状态时上传到服务器，可以被同一账号登陆的其他设备访问。

我们也注意到了这种需求，因此基于**Linux系统下的FUSE**开发了一个支持自动同步的用户态文件系统：

- 分为服务器端和客户端，支持多个客户端之间的文件自动同步
- 文件的同步在客户端文件系统的后台线程进行，不会对前台的文件修改产生影响
- 对一系列的文件操作进行合并，减少传输带宽
- 灵活部署，对以下三种使用场景都可以支持：
  - 两台设备（服务器-客户端）
  - 多台设备（服务器*1-客户端*N）

- 断网状态（断网状态下支持继续操作，连接网络后可以同步断网时操作）

需求分析

下面进行需求分析。由于时间限制暂时没有设计与实现多用户，假设所有使用该文件系统的设备都属于同一用户。

功能需求

- **用户修改本地文件**

名称	用户修改本地文件
角色	用户
描述	用户用户对本地文件进行修改后，将操作序列保存在本地
触发条件	用户修改本地文件
前置条件	服务器已经配置完成
后置条件	系统将用户操作日志保存在本地
正常流程	1.0 用户修改本地文件 1. 用户对本地文件进行修改 2. 系统将用户进行的操作以日志形式存储在本地
可选流程	---
异常	---
假设	---

- **客户端将本地文件修改上传至服务器**

名称	客户端将本地文件上传
角色	用户
描述	用户用户对本地文件进行修改后，自动同步至服务器
触发条件	每隔一段时间自动进行
前置条件	服务器已经配置完成
后置条件	用户对本地文件的修改被存储进服务器中
正常流程	2.0 客户端将本地文件修改上传至服务器 1. 启动线程，检查本地是否有未同步的修改（参见2.0 E1） 2. 与服务器建立连接（参见2.0 E2） 3. 将本地修改日志上传至服务器 4. 上传成功后更新本地的dirty data信息 5. 线程结束

可选流程	---
异常	2.0 E1 本地没有未同步的修改 1. 该线程结束 2.0 E2 连接服务器失败 1. 该线程结束，如果有未同步的信息可以等到下次线程启动时统一进行同步
假设	---

- **客户端从服务器拉取最新的文件信息**

名称	从服务器拉取最新文件
角色	用户
描述	客户端自动将服务器的最新文件信息拉去到本地
触发条件	每隔一段时间自动进行
前置条件	服务器已经配置完成
后置条件	将服务器中未同步到本地的文件同步过来
正常流程	3.0 从服务器拉取最新文件 1. 启动线程，连接服务器（参见3.0 E1） 2. 将服务器中未同步到本地的文件同步到本地 3. 更新本地记录的服务器更新时间戳
可选流程	---
异常	3.0 E1 与服务器建立连接失败 1. 本轮拉取结束，等待下一轮线程启动后再进行连接与同步
假设	---

非功能需求

- **对文件同步的延迟进行处理**

精简了Log结构，只保留必要信息，最大程度减少冗余信息，利用网络带宽。

- **文件修改冲突**

冲突问题有不少复杂的情况需要一一进行解决。

例如，用户在不同客户端同时对文件进行修改产生冲突：在两个客户端均断网的情况下，对同一个文件进行了修改；当两个客户端均进行联网后，对这个文件的修改就会产生一定的冲突。

针对以上所示的类似冲突情况需要进行处理。

- **一致性**

一致性对于一个文件系统来说是正确性的保证。在同步文件系统当中，也需要保证每次操作的原子性，尤其是涉及到网络传输的时候，更应该多加注意：例如一系列操作传输了一半，这时发生了断网该如何处理等等。

- **断网状态下不影响使用与联网后同步**

虽然期望它具有云同步功能，但是文件系统首先需要在单客户端的断网状态下可以正常使用，也就是说至少是在本地断网状态下可以像一个普通文件系统这样工作，而且当客户端联网之后，在断网期间进行的修改仍然能够被同步至服务器。

设计思路

在设计上，由于希望文件系统的同步功能不要影响到前台正常的文件操作，采用多线程的方式，需要进行同步时就申请后台线程。这样存在一个好处就是可以很好地把“文件系统”部分的功能和“同步”部分的功能分离开，方便组织代码。

文件系统部分

客户端和服务器的共同之处是，它们在作为“普通文件系统”时的表现是一致的。因此先使用FileOperation类来实现一个普通的文件系统。无论是对客户端还是服务器端，都使用日志功能记录操作序列。

而传输与同步这样在客户端和服务器端表现不同的功能，需要分别进行实现。

服务器端

服务器端实现一个共享队列SharedQueue，用于存储服务器端文件的操作序列，以及防止冲突修改。

每一条Log都有一个时间戳。当客户端需要拉取服务器端的新修改文件信息进行更新时，记录操作时的log时间戳，这样下次拉取更新时只要从上次更新的位置开始进行更新即可。

服务器对客户端提供以下功能：PUSH（上传本地修改序列日志）、PULL（拉取服务器端新的修改序列日志）、上传文件、从服务器端下载文件。

服务器端还负责解决文件修改冲突的问题。如果有两个客户端同时对一个文件进行了修改，默认的冲突消除策略是同时保存两份副本，对其中一份进行重命名（或删除其中一份）。

客户端

客户端利用多线程机制来实现自动PUSH和PULL。

一个线程池用于进行自动的PUSH，以一定的时间间隔（默认为5s）对本地文件进行的修改日志以及对应的文件进行上传。

另一个线程池用于进行自动PULL，以一定的时间间隔（默认为10s）对服务器端新修改过的文件。

模块、接口及代码框架

在上面的设计思路下，本部分设计了同步文件系统的各个模块、接口以及功能，并介绍了最终的代码框架。

Log表项

对于文件系统执行的操作，会使用日志方式进行记录，同时操作记录也会作为服务器与客户端之间进行同步的协议。

以下是每个日志表项包含的内容、含义以及实例：

TOKEN	**---**	**ARGVS**
表示该项操作的编号，1...18	操作含义	操作涉及的参数。不同的操作附带的其他参数不同，需要根据具体情况进行解析。
1	WRITE_DONE	path
2	RENAME	from, to
3	MKDIR	path, mode
4	RMDIR	path
.....	.....	.....

在设计了Log表项之后，文件系统的历史操作可以被很好地记录，也方便了服务器与客户端之间信息的同步。

普通文件系统接口

如果不考虑文件内容同步的功能，无论对于客户端还是服务器，我们都希望它们能表现得像一个普通的文件系统。因此对于文件读写、删改、重命名、修改权限等等，我们都使文件系统的行为就像一个普通文件系统一样。相关的函数接口等都遵循FUSE框架给出的接口。

服务器对客户端提供的网络传输接口

服务器支持客户端的四个行为：PUSH 、PULL、上传文件、从服务器端下载文件。

- **PUSH**

push操作要求提供本地进行的新修改的序列，包括增加的文件以及文件夹、删除的文件以及文件夹、发生过重命名的文件夹。

```
/*
newFiles:新增文件集合
deleteFiles:删除文件集合
newDirs:新增文件夹集合
deleteDirs:删除文件夹集合
renameDirs:重命名的文件夹集合
*/
void NetworkAgent::pushToServer(
  std::set<std::string> &newFiles,
  std::set<std::string> &deleteFiles,
  std::set<std::string> &newDirs,
  std::set<std::string> &deleteDirs,
  std::set<std::pair<std::string,std::string>>
&renameDirs) {
//将四个set转化为四个字符串
//当对某个文件夹进行重命名时，需要将里面涉及到的所有文件都进行路径
修改
//......
//记录当前操作的token为PUSH，方便服务器端解析
//......
//发送字符串信息
//......
//对涉及到的文件进行上传
    for (const auto &path: newF) {
        uploadFile(server_fd, path);
    }
}
```

- **PULL**

pull操作需要提供该客户端上次从服务器进行同步的位置，此次同步就只需要从上一次同步的位置开始进行同步即可。

```
/*
last_sync:上一次从服务器pull时的位置（服务器上存储的log的id）
*/
void NetworkAgent::pullfromServer(uint64_t last_sync) {
    int server_fd = *connections.begin();
    Operation_t token = PULL;
    sendRaw(server_fd, (char*) &token, sizeof(token));
    sendRaw(server_fd, (char*) &last_sync,,
sizeof(last_sync));
  //从服务器下载相应文件
  //......
}
```

- **UploadFile**

文件上传操作要求提供服务器的socket id，以及需要上传的文件的路径。

```
/*
remote_fd:服务器的socket id
path:文件在本地存储的相对路径
*/
void NetworkAgent::uploadFile(int remote_fd, const
std::string &path) {
  //......
}
```

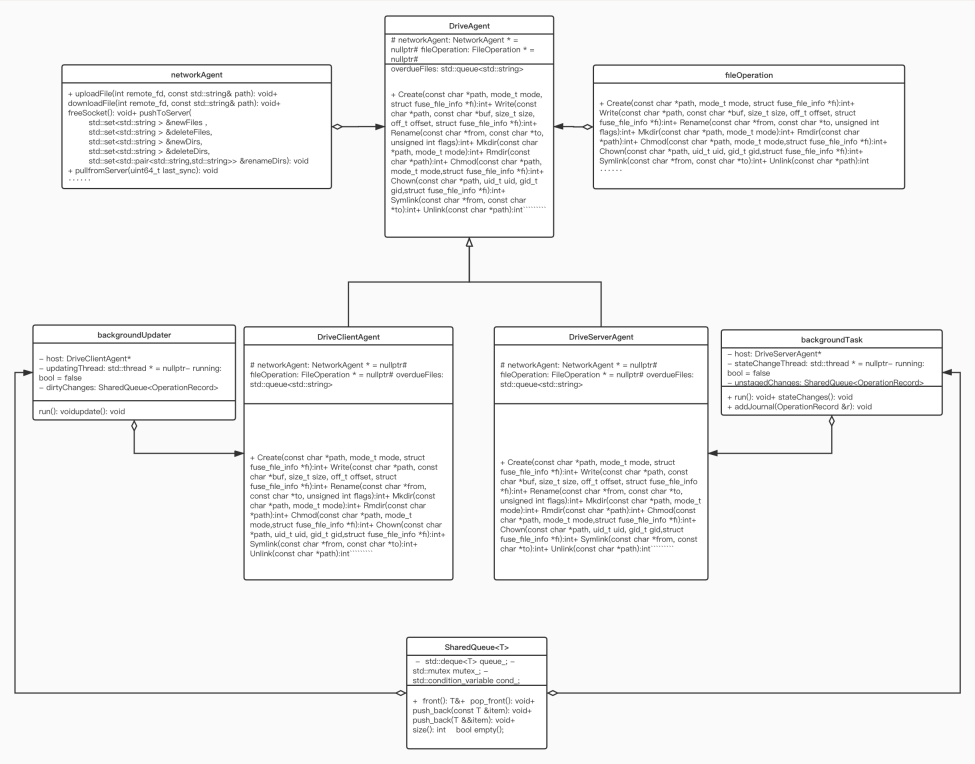
- **DownloadFile**

文件下载要求提供服务器的

```
/*
remote_fd:服务器的socket id
path:文件在服务器存储的相对路径
*/
void NetworkAgent::downloadFile(int remote_fd, const
std::string &path) {
  //......
}
```

代码框架

综上所述，文件系统设计代码框架如下图所示。



开发难点

- 关于环境配置

FUSE在Ubuntu20.04版本下安装是非常方便的，可以使用apt-get在命令行实现安装。但是在Ubuntu18.04版本下就需要手动下载安装包并且解压，配置环境需要花较多时间。而且不同版本的FUSE似乎不能方便地通用，在最初的环境配置上的确花了较多时间。最终发现最方便的解决方式还是直接下载一个20.04版本的Ubuntu操作系统安装虚拟机，然后使用命令行安装FUSE。

- 关于远程协作

尽管有各类会议软件可以对远程协作实现支持，而且两位开发者均熟悉git，但是还是遇到了分工不均以及沟通不便的情况。不过换一种角度，远程协作的方式对于协作开发也是一个比较好的锻炼。

- 关于框架设计

由于需要考虑各种异常情况，文件系统的功能逻辑较为复杂。为了同步需要设计日志系统与应用网络通信协议。

- 关于代码编写

FUSE调试困难，难以获得中间变量；用户态文件系统需要了解各种系统调用。我们在开发过程中没有使用其他库，总代码量在3000行左右(含空行和注释)。

- 并发实现困难

由于服务器和多客户端连接，多线程的调试较为混乱；在进行同步和网络操作时，不能阻塞前台的正常文件操作；要实现 thread-safe 的队列。

进一步改进

以上实现的仅仅是一个具有初步同步以及冲突保护功能的文件系统，它能实现基本的文件自动云同步，但是仍然有着很大的改进空间，主要在于以下几个方面：

- 线程安全

目前的系统在互斥方面存在一定的问题需要解决。

例如，在文件系统测试时均使用较小的文件进行本地测试，一来文件较小，二来本机同时作为服务器和客户端时文件传输较快。但是当部署到真实的服务器进行测试时，会出现问题：由于网络传输速率有限，有可能在上一轮进行的大文件传输还没有结束时（自然也就没有更新客户端的dirty page），下一轮文件传输就会开始，这样就会有文件重复传输等等问题。

再例如，当某一个客户端在断网时进行了更新，同时服务器端数据也由其他客户端进行了更新，那么该客户端重新联网时，是先将本地文件传送到服务器还是先将服务器的最新版本同步到本地？如果产生冲突应该如何进行处理？

以上等等问题都需要查阅资料，进行进一步的解决。例如我们可以查找iCloud在这方面的相关设计文档。

- 部署至服务器

如上一节所说，目前的测试是将本机作为服务器进行的。本机作为服务器有诸多方便之处，例如服务器和客户端的网络文件传输并不会真的进行网络传输，因而速度较快；再例如“服务器”较为安全，不必考虑受到攻击等情况。

但是在真实的应用场景当中自然还是需要将部署至真实的服务器。这样面临的问题就是：一是尽可能提升性能，二是考虑服务器安全性，考虑可能的攻击等情况。

- 多用户功能，以及信息加密

目前的文件系统实现的功能是假设所有设备都属于同一用户；但是在更加真实的场景当中，需要允许多用户功能。我们不清楚iCloud具体如何实现账号登陆功能，因此暂且没有实现。

与之相关的是文件内容的加密。一方面是服务器端和客户端在进行文件传输的过程需要进行一定的加密，另一方面是不同用户的文件内容也需要进行一定的隔离。

- 文件系统一致性

文件系统如果希望保持稳定运行，需要更多的一致性保证。目前仅仅考虑了简单的冲突、断网情况，对于较复杂的一致性问题还没有解决，也无法保证某几个操作正确时，文件系统一定能保持正确。

例如在目前的实现当中，文件系统存在的重要问题就是，客户端会先从服务器端拉取Log，再根据Log对文件进行下载。

当然，用户是不能直接访问Log的，而如果文件系统没有把服务器端文件下载完成，用户也看不到有这样一个文件，因此用户打开一个“不存在的文件”的风险可以暂时不考虑。

但是仍然存在其他的问题，一个简单的例子是：假设在某次客户端进行了一次PULL操作，但是还没有来得及下载文件。此时服务器端把某一个文件删除了，但是删除操作的Log还没有被同步到客户端；这样就会出现客户端试图下载一个服务器端已经存在的文件的情况。

关于一致性还有很多复杂的场景需要加以考虑。

- **性能进一步优化**

从目前来看，服务器和客户端之间的沟通以来网络传输，而两者之间的通信与网络传输是后台线程周期性进行的，即使没有进行文件更新也保持着文件传输。这样从实现的角度来看有很多优势，但是与此同时这样可能会带来服务器与客户端之间的确认信息传输过于频繁、真正的更改传输不够及时、大文件更改频繁占用大量带宽，以及本地的添加完之后马上删去的临时文件也进行传输等等问题。

我们已经针对迅速被删除的临时文件作出了优化，在将一系列的文件系统操作Log传输出去之前，在本地会先合并能够合并的操作。

而针对其他的问题，可能需要在同步方式上做出一定修改，例如大文件进行改动时再从服务器拉取、以及需要进行读取/修改某个文件时再单独从下载某文件。不过这对服务器功能提出的要求似乎更为复杂了。

分工

陶天骅：设计框架，填充函数。

陈张萌：填充函数，写文档.