分布式文件系统实验报告

专业&班级	姓名	学号	日期
计科2班	陈泓仰	15303009	2018.12.29

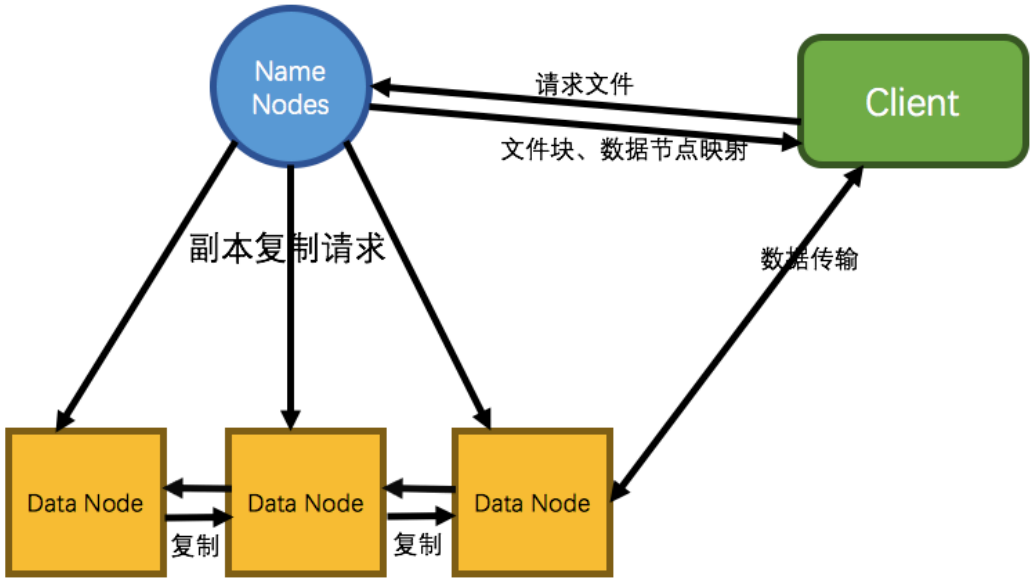
题目要求

- 使用RPC模式作为不同节点之间的通信方式，实现分布式文件系统。
- 要求文件系统具备基本的文件操作模型包括：创建、删除、访问等功能。
- 作为文件系统的客户端要求具有缓存功能即文件信息首先在本地存储搜索，作为缓存的介质可以是内存也可以是磁盘文件
- 为了保证数据的可用性和文件系统性能，数据需要创建多个副本，并且在正常情况下，多个副本不在同一物理机器，多个副本之间能够保持一致性（可选择最终一致性或者瞬时一致性）。
- 支持多用户即多个客户端，文件可以并行读写（即包含文件锁）

解决思路\实现细节

系统架构

- 在这次实验设计的分布式文件系统，模拟非对称文件系统HDFS进行实现。设计架构如下：



- 类似于HDFS，本文件系统也是采用master/slave架构。同样的，一个服务器集群是由一个命名空间节点Namenode和一定数目的数据节点Datanodes组成。Namenode是一个中心服务器，负责管理文件系统的名字空间(namespace)以及客户端对文件的访问。而服务器集群中的Datanode节点负责管理它所在节点上的存储。在本文件系统中，Namenode节点暴露文件系统的名字空间，用户能够向Namenode发出请求，并以文件的形式在Datanodes节点上面存储数

据。

- 在文件系统中，对文件采用分块存储的方式。一个文件被分成一个或多个数据块，这些块存储在一组Datanode上。Namenode执行文件系统的名字空间操作，比如打开、关闭、创建文件或目录。它也负责确定数据块到具体Datanode节点的映射。Datanode负责处理文件系统客户端的数据传输操作（比如读写）。在Namenode的统一调度下进行数据块的创建、删除和访问。

采用文件分片技术

- 为了提高文件访问速度，对大型文件（大于10M的文件）采用文件分片技术，也就是将一个文件分成一个或多个数据块，将这些数据块散列在不同的DataNodes中，由NameNode节点对数据块索引与DataNode编号之间的映射表进行管理。
- 这样做是为了提高并行文件读取的速度，使得不同的client能够同时读取同一块数据。

命名空间数据结构

- 在命名空间节点(NameNode)中维护着整个文件系统的目录树信息以及文件chunk号到数据节点之间的映射表，每当服务端建立时，NameNode节点首先从磁盘中读取这两个信息，并处理为对应的数据结构。根据客户端的文件传输请求来实时更新这一个数据结构、同时进行持久化。
- 目录树信息数据结构，类似于Linux文件系统目录格式，处理为以下形式：
  - 磁盘：以 “. ”作为根目录

```
文件夹名称    该文件夹中所有的子文件夹      该文件夹中所有的子文件
. test_dir,try,test,chenhy,g10 server.py
./test_dir test,test2 trainData.txt
```

  - 内存目录树表：同样以 “. ”作为根目录

```
文件夹名称:{[该文件夹中所有的子文件夹],[该文件夹中所有的子文件]}
.:{[test_dir,try,test,chenhy,g10],[server.py]}
./test_dir:{[test,test2],[trainData.txt]}
```

- 文件chunk号到数据节点之间的映射表
  - 磁盘：以Json文件存储

```
{"dir_name": ".", #目录名称
  "file": [#文件列表
      {"file_name": "server.py", #文件1文件名字
       "file_chunk": [
           {"chunk_name": "chunk0001", #文件chunk号
            "file_server_list": ["1", "2"], #拥有该chunk的DataNodes索引
            "client_list": [], #拥有
            "Version": 0,
            "lock_symbol": 0
           }
       ]
      }
```

```
    ]
}
```

- 内存：以字典为数据结构作为存储

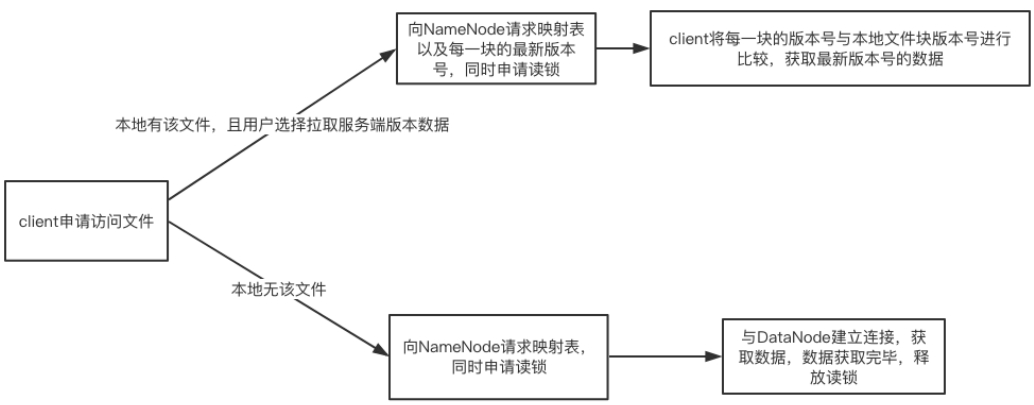
```
".":{"server.py":{"chunk0001":{"file_server_list":
['1','2'],"client_list":[],"Version":0,"lock_symbol":0}}}
```

文件操作的实现

- 由于命名空间与数据分离，也就是没有在服务器中有如同目录树一样的真实文件数据，因此也没有办法直接使用python的os模块来进行比如切换文件目录cd指令、列出文件夹中所有文件及文件夹ls指令，创建文件目录mkdir指令、创建文件mkfile指令、删除文件rm指令。
- 必须自己对目录树数据结构、文件chunk号到数据节点映射表数据结构进行处理，来实现上述指令。下面简述一下对上述指令的实现细节。
- cd：在客户端与服务端中会在程序中共同维护客户端的当前目录位置，当使用cd命令进行切换时，会根据**当前目录位置、该目录的子文件夹名**这两个因素来进行同步改变客户端的当前目录。
- ls：由于在服务端维护了当前客户端的位置，以及目录树文件结构，因此当客户端通过rpc调用服务端下的ls命令时，namenode节点只要在目录树文件结构进行字典键值搜索即可返回结果。
- rm：在本文件系统中，并没有实现数据节点的文件删除，当客户端调用rm命令，并指明文件名称时，namenode节点**只会将该文件名字从目录树数据结构以及映射表结构**中删除（同时将这两个表进行保存），并不会将此文件从本地缓存以及DataNode节点中删去。
- mkdir：客户端通过指定文件夹名称，调用rpc调用服务端下的mkdir命令，服务端首先查看是否存在同名文件夹，若非，则在目录树结构以及映射表数据结构中加入相应的名字，并对这两个表进行保存。
- mkfile：客户端会向NameNode节点发出创建文件的请求，由NameNode节点判定该客户端当前目录下是否具有同名文件，没有的话，则创建成功，并告知客户端创建已成功。

文件访问过程及实现

- 先看流程图帮助理解：



- 类似于HDFS的文件访问过程，客户端首先通过rpc将所要获取的文件名以及对应的chunk号发送给服务器，由服务器将对应的chunk与DataNode节点的映射列表返回给客户端，然后客户端再与DataNode节点建立连接，将数据下载到本地。
- 在本文件系统中，为了减少数据通信开销以及减少大文件数据传输时间，当客户端从DataNode节点中获取了文件之后，会将该文件缓存在本地磁盘中。当客户端向NameNode节点发出数据传输请求时，客户端首先判断本地是否拥有所请求的文件，并询问client是否与服务端数据文件进行同步。NameNode节点除了会将chunk与DataNode的映射表返回到客户端之外，还会返回所请求的文件块版本号，当客户端获取到文件块版本号之后，会将服务端中文件块的版本号与本地文件的版本号进行比较，若服务端中该文件块的版本号比本地文件的版本号更新，则客户端会获取最新版本的文件块，若非，则不会获取。
- 在客户端向NameNode节点获取映射表与版本号时，会由NameNode节点判定该文件块的锁标志位，如果该锁为写锁，则会进行自旋等待，直到写锁释放，则马上读文件加读锁。

文件更新

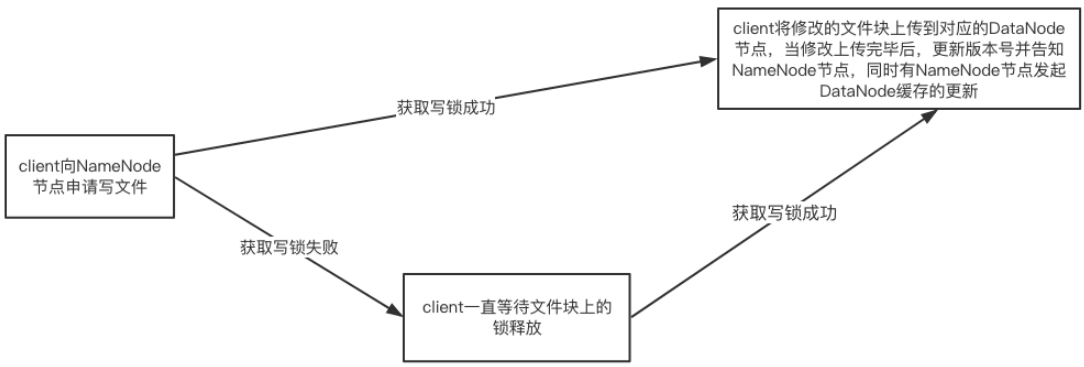
- 首先要说明的一点是，更新的文件必须是由客户端创建的或者是在服务端目录树和映射表中有对应文件记录的文件。
- 客户端上传文件的日志结构
  - 磁盘文件

```
#datanode下的文件名称（Data文件夹是DataNode中存储文件的文件夹）,{chunk编号:["DataNode编号",DataNOde的端口号,本地文件版本号]}
["Data/abcd.txt_dir/abcd.txt", {"chunk0001": ["1", 18862, 0]}]
```

  - 内存结构

```
#datanode下的文件名称（Data文件夹是DataNode中存储文件的文件夹）:{chunk编号:["DataNode编号",DataNOde的端口号,本地文件版本号]}
"Data/abcd.txt_dir/abcd.txt":{"chunk0001": ["1", 18862, 0]}
```

- 客户端更新文件过程
- 流程图



- 客户端可以在本地缓存的文件上进行修改，然后将修改文件同步到服务端。
    - 首先客户端先读取上传文件日志记录，找到DataNode节点及节点对应的端口号，然后向NameNode节点申请写锁，如果该文件块上已经被加了读锁或者写锁，则将等待文件块上的锁释放。
    - 当获取到了写锁，之后，客户端会与DataNode节点建立连接，将更新的文件上传到DataNode节点中，同时更新上传日志记录中的文件块版本号，并告知NameNode节点，释放该文件块上的写锁。
    - 当文件块的更新完成后，NameNode会查询拥有该文件块的DataNode节点，然后由NameNode节点要求DataNode节点中数据的更新。

版本管理与读写锁实现

读锁与版本管理

- 客户端每次获取服务端文件时，都会比较本地文件块与服务端文件块的版本号，若服务端文件块有所更新，则与DataNode节点建立连接，保证客户端所拥有的文件是最新版本。
- 在客户端获取服务端文件时，需要进行加锁。那么为什么选择在客户端向NameNode节点获取映射表和版本号时加锁，而不在客户端与DataNode节点建立数据传输请求之前加锁呢。
- 目的，也是为了保持客户端本地文件块版本与服务端文件块版本的一致，保证客户端本地文件是最新的。
- 设想一个场景，请求读文件的客户端此时拥有所有最新版本的文件块，但是此时客户端仍想与服务端进行文件同步。如果客户端与DataNode节点建立数据传输请求之前加锁，那么在客户端从NameNode获取映射表与版本号列表之后，有其他的客户端向NameNode请求写操作，并成功更新了文件块的版本号，此时请求读操作的客户端无法意识到文件块版本已经出现了更新，则不会获取到当前最新版本的文件块。

写锁与版本管理

- 客户端每次修改文件，并将修改后的文件上传到服务端之后，服务端都会将服务端文件块的版本进行更新，然后将最新版本的文件块进行一次全局的同步。
- 而客户端在修改文件时，在客户端处进行自旋等待，在这里选择在客户端进行自旋等待是为了减少服务端的负载。因此选择由想要进行修改的客户端进行等待

副本的更新与管理

- 为了保持文件块的一致性。Namenode节点在得到客户端的更新文件块之后，会将拥有该文件块的DataNode节点中的所有该文件块进行更新。实现的是最终一致性。

遇到的问题

- 在这次分布式文件系统设计的时候还是存在着一些缺陷需要解决。
    - 首先是当client在上传数据（更新操作）的时候突然发生掉线，那么此时该文件块仍是处于写锁状态，因此当后续客户端想要继续修改或者读取此文件时，则会一直处于申请锁的状态。对于这个问题，其实有一个解决思路，就是加入超时机制，由NameNode节点定期的去搜索映射表中该文件块的锁标志位，当超过一定时限时，将该标志位置为回无锁状态。
    - 另外一个问题就是，当有两个client对同一版本v1的文件块进行修改时，client1先上传了本地的文件块之后，NameNode节点和client1中该文件块的版本号都变为v1 + 1，而此后client2上传该文件块之后，就会出现服务端和client2本地的该文件块的版本号不一致的问

题（服务端为v1+1，而client2为v1）。对于这个问题，我的解决思路是必须在分布式文件系统中使用Paxos算法，来对文件块的更新进行同步。
    - 最后一个问题就是，NameNode节点没有以日志的形式来保存对数据的更新操作，而是通过维护映射表，不断的将这个映射表进行保存，IO次数较多，因此，这也提高了NameNode节点的负载。
- 该文件系统在进行数据的上传时，采用的是同步复制，必须先等待复制操作进行完毕之后，才能进行接下来的操作。

总结

- 在这次分布式文件系统的设计中，通过模仿HDFS的结构来设计分布式文件系统，这个过程让我对HDFS，甚至是GFS的文件架构有了更深一步的理解。清楚了将服务器分为NameNode节点和DataNode节点的目的是使主服务器不会构成瓶颈，也为存储系统的扩展提供了更强的扩展性；清楚了NameNode节点中的映射表设立的目的是为了维护服务端所有文件的操作记录。
- 除此之外，通过对分布式文件系统中读写以及读写锁的实现，更加深刻的认识了一个分布式文件系统中要保证节点间数据的一致性或者是使得多个client节点对文件块版本号的共识是一件非常不易的事情，而达成client节点对文件块版本号的共识，可以有Paxos算法来保证，通过设计这次文件系统也对这个问题认识的更加深刻。