**SDFS Document**

14302010037 Chengzu Ou

**1. Abstract**

This project intends to build a distributed file system called *Simple Distributed File System(SDFS)* which support some basic operations, such as *open* & *create* a file or directory. In this document, we consider the architecture and some tricky details of *Client*, *NameNode* and *DataNode*, both holistically and in detail.

**2. Background**

Nowadays, more and more people choose to store their files in some cloud servers. It is safe, fast and convenient. However, for cloud storage service providers, with the growth of the users, it is not easy to serve such a large amount of client at the same time. There are also a lot of problems in traditional file system, such as low fault-tolerance and low react speed. Thus, we need a more powerful and reliable file system which should also be distributed.

There are already plenty of distributed file system in the market, such as *HDFS*. In this project, we do not intend to build a so sophisticated file system that can compare to any one of them. Instead, we are going to learn the basic concept of distributed file system by building one.

**3. Goals**

The main goal of this project is to build a distributed file system that support basic operations. In the meantime, we are going to try our best to improve the performance and fault-tolerant of the system.

**4. Architecture**

*SDFS* has a master/slave architecture. The system consists of a single *NameNode*, a master server that manages the file system namespace, and a number of *DataNodes*, which manage storage attached to the nodes that they run on. According to the requirement, a file is split into one or more blocks and these blocks are stored in a set of *DataNodes*. The *NameNode* executes file system namespace operations like *open*, *create*, *close*, and *mkdir*. It also determines the mapping of blocks to *DataNodes*. The *DataNodes* are responsible for serving read and write requests from the file system’s clients. The *DataNodes* also perform block creation, deletion, and replication upon instruction from the NameNode.

**5. Implementation**

5.1 Execution Flow of *open*, *create* and *mkdir*

5.1.1 *open*

Step1: *Client* sends *open* request to *NameNode*.

Step2: *NameNode* checks *DirNode* to get entry of the file.

Step3: *DirNode* returns the entry of the file which contains the *ID* of the *FileNode*.

Step4: *NameNode* asks *FileNode* for the information of the *BlockInfos*.

Step5: *NameNode* gives *Client* the information of the file blocks.

5.1.2 *create*

Step1: *Client* sends create request to *NameNode*.

Step2: *NameNode* asks *DirNode* to check if the file name conflict with the current file.

Step3: *NameNode* creates a *FileNode* if the name does not conflict with current file and assign a free block of the *DataNode* to this *FileNode*.

Step4: *NameNode* gives *Client* the information of the newly assigned file blocks.

Step5: *Client* sents a request to specific *DataNode* according to the block info given by *NameNode* and open a output stream to get ready to write data.

5.1.3 *mkdir*

Step1: *Client* sends *mkdir* request to *NameNode*.

Step2: *NameNode* asks *DirNode* to check if the directory name conflict with the current directory.

Step3: *NameNode* creates a *DirNode* if the name does not conflict with current directory.

Step4: *NameNode* gives *Client* some acknowledgements.

5.2 Execution Flow of *read &* *close*

Step1: *Client* sends some requests to specific *DataNodes* according to the block infos given by *NameNode* and open a input stream to get ready to read the data.

Step2: *Client* reads data from *DataNodes* and rearrange them into a complete file.

Step3: *Client* sends *close* request to *NameNode*.

Step4: *NameNode* updates the matadata of the file by writing them to *FileNode*.

5.3 Execution Flow of *write &* *close*

Step1: *Client* sends a request to specific *DataNode* according to the block info given by *NameNode* and open a output stream to get ready to write data.

Step2: *Client* writes data to the *DataNodes*.

Step3: If the block is full, *Client* sends a *addBlock* request to *NameNode* to require a new block for the file.

Step4: *NameNode* adds a *BlockInfo* to the specific *FileNode* and assigns a free block in *DataNode* to the *BlockInfo*.

Step5: *NameNode* gives *Client* the block information.

Step6: *Client* continues on Step1 until no data more to be written.

Step7: *Client* sends *close* request to *NameNode*.

Step8: *NameNode* update the matadata of the file by writing them to *FileNode*.

**6. Difficulties**

- How to dynamically assign new block in the *DataNodes* to *FileNodes*?

- How to manage the replication of some blocks?

- How improve the performance of the system?

**7. Bottleneck**

- When writing data into a file, it is insufficient to sent request to *NameNode* once a block is full and wait for *NameNode* to return a new block.

- It is also insufficient to update namespace every time we make some changes. Perhaps we can use log and update the whole namespace only when not too busy.