SDFS-3 Final Document

14302010037 Ou Chengzu

Introduction

Simple Distributed File System is a toy distributed file system written in Java. Its architecture is mainly based on HDFS which is a commercial distributed file system widely used in the industry. SDFS serves as a course lab and is used for students to let them have a deep understanding of the distributed file system and have a basic idea of how the real system works.

System Design Overview

The system is divided into three independent part: Client, NameNode and DataNode. Client is running on the user machine while NameNode and DataNode are on another two different machines. The three part communicate with socket.

Client provide to user a simple API to operate on the file system including opening, closing or creating a file and make a directory. Client also provide a FileChannel to let user read or write on a certain file.

NameNode is used to manage the metadata of the file system including the whole file tree and file data location and its access permission.

DataNode is used to manage the data of the file system.

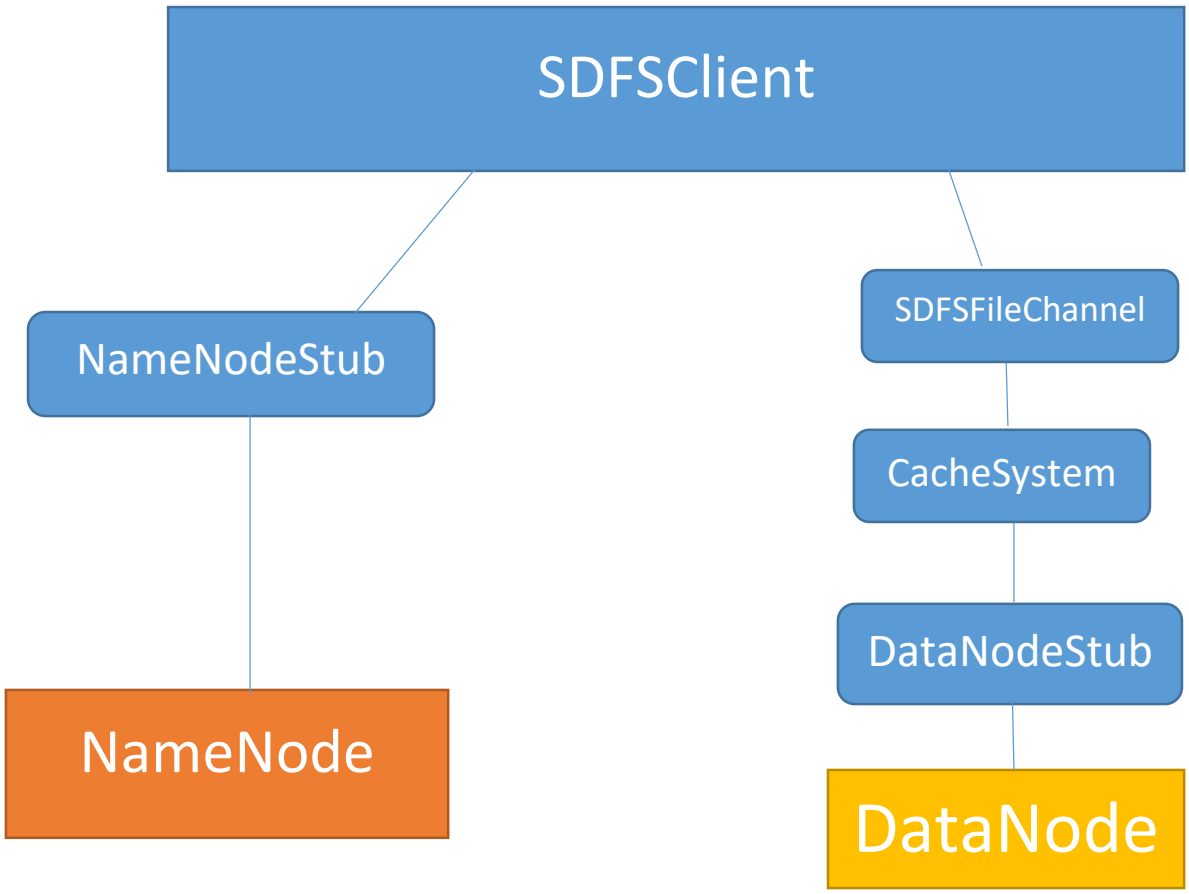
Client Design

Client communicate with NameNode through NameNodeStub to open/close a file.

After opening a file, NameNode will send the file metadata to Client. Client use the metadata to construct a SDFSFileChannel and use it to read/write data.

SDFSFileChannel directly read/write data from CacheSystem and the CacheSystem communicate with DataNode through DataNodeStub.

The CacheSystem helps to reduce the data transferred between Client and DataNode.



NameNode Design

NameNode is actually a server that listens on a certain port to get request from Client or DataNode using NameNodeServer.

The core of NameNode is composed of four components: DataBlockManager, OpenedFileNodeManager, DiskFlusher, and Logger.

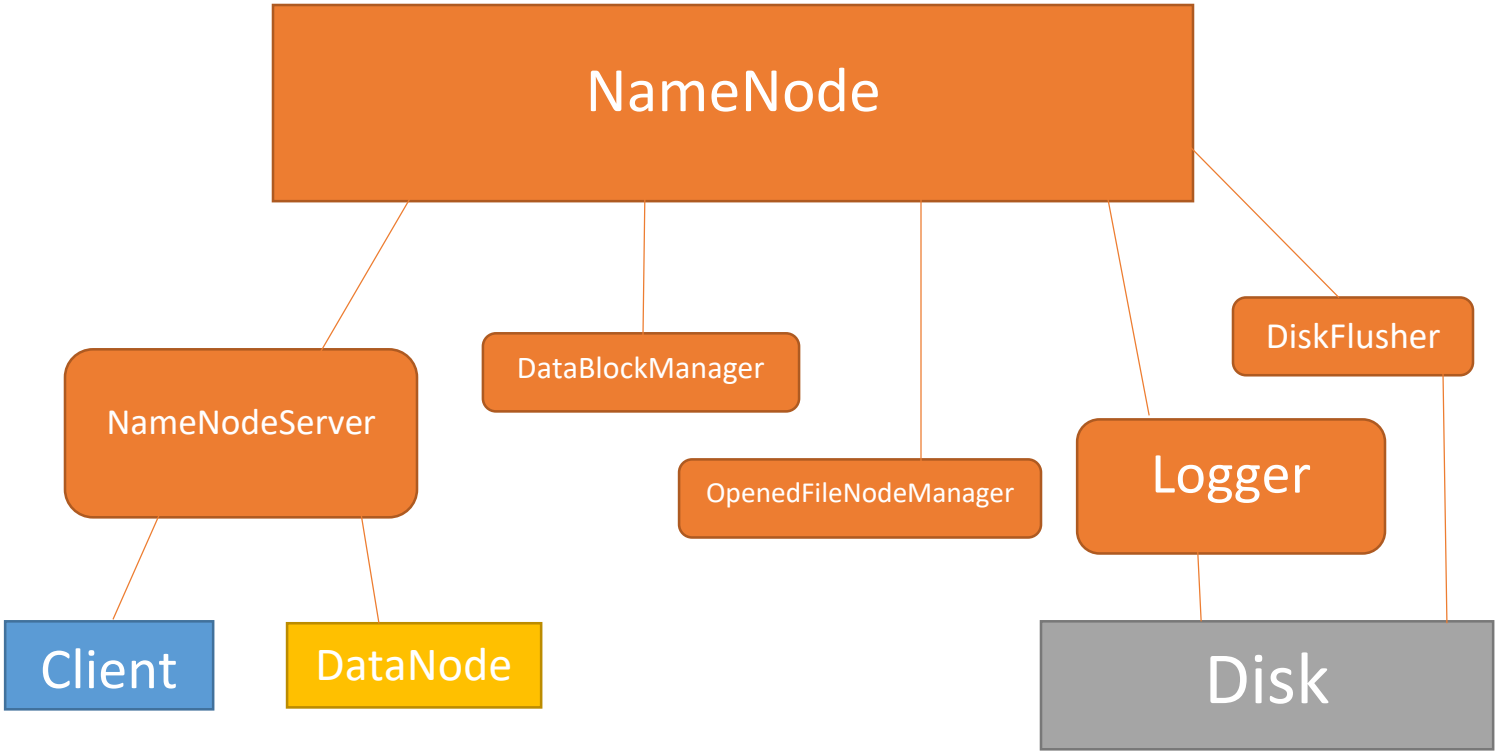
DataBlockManager is used to manager the Block IDs of the free blocks on (each) DataNode. It uses a HashMap that map the block id to its reference count. When NameNode initializes, it will read the entire file tree to record all the currently occupied block ids and set its reference count to one. When a user opens a file, it will add the reference count of the ids of all the blocks of the file by one. When a user closes a file, it subtracts it by one. When a id's reference count reaches zero, it means the block is now available and can be assigned to some other files in the future. Each time the NameNode ask for a new block id, it will search the map to return the least id with a zero reference count.

OpenedFileNodeManager is used to manage currently opened file. It uses two HashMaps for read-only and read-write opening file and to map its Token to correspondingly OpenedFileNode. When opening a file, the NameNode will make a deep copy of the file node and wrap the original file node and its copy into a new object called OpenedFileNode. Each time the Client or DataNode asks to read or write metadata of some opened file, it will search the HashMap by token and do the operation on the copy of the file node. For read-only opening, it makes sure that the metadata of the file is saved at the time of opening and the future changes of the file will not affect the currently opened read-only file. For read-write opening, it makes sure that the modification of the file node will not affect the file tree until it successfully closed. When closing a read-write file, it will replace the metadata of the file node in the file tree with the newer version of the data stored in the opened file node in the HashMap.

DiskFlusher is used to flush the file tree into the disk once after given internal seconds. It will work with Logger to make sure that when it is time to flush, there is no unfinished transaction, namely the transaction that has already START but not COMMIT or ABORT. Each time after a flush, it will write a CHECK_POINT log into the log file through Logger.

Logger is used to read and write log. During the regular working time, it serves as a log writer that write log into a log file. When the system is crashed and need to

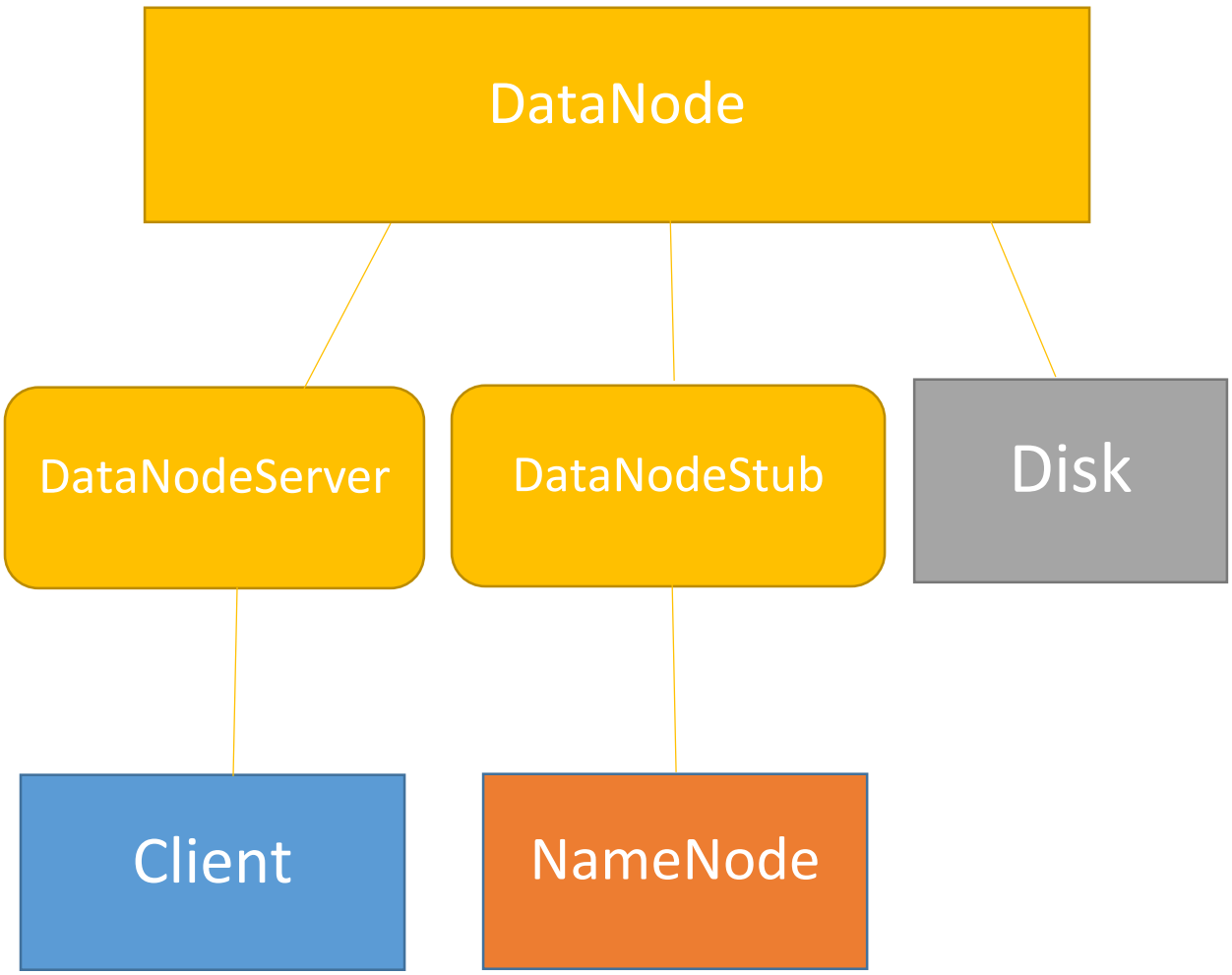
restart, the Logger will read the log file previously written on the disk and try to restore the exact states just before the crash. Since we have done the persistence work for the file tree and it will read the file tree stored on the disk when restarting the system, we do not need to build the entire file tree from the beginning and could only redo the logs after the last CHECK_POINT. But note that some other information of the system such as currently opened file node tokens will not be stored on the disk, we still need to restore that information by reading the logs before the last CHECK_POINT.



DataNode Design

DataNode uses DataNodeServer to listen on a certain port to handle request from Client and uses DataNodeStub to communicate with NameNode to get access permission information.

DataNode directly read/write data from/to disk.



Thread-Safe Design

The biggest thread-safe requirement is on NameNode. And to make it tread-safe, we only need to make sure all the operation on the shared data fields of each Object are atomic. To reach this goal, we first need to figure out what exactly data fields are shared by two or more thread. According to my design of NameNode above, we can see that each component of NameNode has some data fields that are shared by threads. Let us go through them one by one.

In DataBlockManager, we have a HashMap that map block id to its reference count that is shared by threads. We use a lock to make sure each operation that involves the HashMap is atomic. Note that every operation on the HashMap involves modifying it, thus read-write lock does not significantly improve its performance compared to normal lock.

In DiskFlusher, there is no shared field since only one thread is doing the flush job.

In Logger, there are three fields that maybe shared: next log id, currently uncommitted transaction count, and the object output stream that write log to disk. For the first two integer fields, we use AtomicInteger to make sure the operation on it is atomic. For the object output stream, we use a log. Note that although we need to read log, we do not need to use read-write lock to improve the performance because reading the log only happens when initialize the Logger and during initializing, there will be no write request and we do not need to lock it up.

In OpenedFileNodeManager, there are two HashMap that are shared by threads. Since sometimes we only need to read them and do not have to modify it, we can use read-write locks to improve the performance because two threads can simultaneously read the same field.

In NameNode, the only the biggest thread-safe requirement is the file tree. To maximize the performance, we choose to set the lock on the node level. That is, we have a read-write lock for each file node and directory node. When we have to read or write a node, we use lock to prevent two thread simultaneously modifying the same node. Otherwise, reading simultaneously is allowed.

Multi-Client Solution

Firstly, my RPC design allows multiple requests handling on both NameNode and DataNode.

Secondly, just as I have described in NameNode Design above, I use OpenedFileNodeManager to manage opened file node and this includes prevent two clients open read-write the same file node. This is achieved by maintain two HashMaps that map currently opened file node token to its OpenedFileNode object. When a new request asks to open a read-write file, it will check the HashMap of read-write file to make sure it is not opening by other user.

Thirdly, as I have mentioned above (see NameNode Design), I use OpenedFileNode to keep the opened file node untouched and the file tree untouched too.

Changes of the Test Code

I do not use RMI. So I have to change the setup code to match my design.

I keep get StackOverFlowException when running LogTest, so I change the number of the directories made from 256 to 11.

In my design, I write a START log no matter whether an action is done or not. So I remove the last line of LogTest which test whether the log is empty when some exceptions occurred when performing some actions.

All the changes do not modify the behavior of the test.

I also get a StressTest from my roommate to test the performance of my system. It turns out my system do pretty well ☺