Document 2

14302010037 Ou Chengzu

Design of RPC

For `Client`, there are two `Stub`s, `NameNodeStub` and `DataNodeStub`, to communicate with the `NameNode` and `DataNode` respectively.

For the `NameNode`, I use `NameNodeServer` to act like a Server running continually listening on `NAME_NODE_PORT` to handle client requests. So does the `DataNode`.

For each of the server, it listens on a specific port. As it accepts a new client request, it starts a new Thread to handle that client's requests.

I use a self-defined class called `NameNodeRequest` to wrap the request message sent from `Client` to `NameNode` server and a class called `NameNodeResponse` to wrap the response. For `DataNode`, it is basically the same idea.

There are some necessary fields in each request and response object to store the message. To make the package passed among host as small as possible, I use the idea of the `union` structure in C to design the class. That is, some fields such as string and integer are shared by different kinds of Request/Response.

To handle `Exceptions`, I also add some exception fields in response class. If the server throws some exceptions when doing the business, it will add the exception into the response object and when client receive such a response object with exception inside, it will throw the exception to let the client use know what went wrong. All the exceptions are strictly used under the protocol given.

Every time the client sends a request, the server will always send back a response to acknowledge the client that it has received the request and also let the client know whether the request is successfully done or not. If the request is meant to ask for some return value, such as opening some file, the response will also include the return value.

To handle the different types of the request and corresponding response, I use `enum Type`. On server sides, when they receive a request, they will firstly check the request type using a switch statement and send different kinds of request to different kinds of methods in kernel logic, which is NameNode/DataNode, to get their return value and finally send back response. The client just sends the request and wait for the response. It blocks if it does not get a response.

Design of Cache

I use a class called `CacheBlockInfo` to store the cached blocks information such as whether it has been recently used and whether it has been changed. I use a class called `CacheSystem` to manage the cached blocks, including adding a new cached block and removing an old cached block.

I use one pointer to implement LRU algorithm. Whenever a cached block is used, it will set its corresponding `CacheBlockInfo flag` to one. Whenever reading a block is missed, it will read it from server and cache it. When the cache is full and we need to remove a block, it will get a block to remove according to LRU algorithm and check whether the block is changed. If the block is changed, it will write it to server. Then, it is save to remove this block from cache.

For `SDFSFileChannel`, the cache is where it read or write data and how the cache work is hidden from upside applications.

Problems I Met

At first, I had trouble handling the exceptions. But then I realize that I can just put the exceptions into the response package and sent to the client.

Since the `NameNode` and the `DataNode` cannot directly communicate with each other, I kind of not know how to add a new block, because when the client sends this request to `NameNode`, `NameNode` does not even know which block is free and which is not. How can it send back a free `LocatedBlock` to client? I thought it must at least ask some `DataNode` for some free block list. But after consulting the TA, I realize that I can just store the block list information in the `NameNode` and the `DataNode` can be as simple as just pure reading and writing. So I add a `HashMap<InetAddress, ArrayList<Integer>>` to `NameNode` and use it to map the specific `DataNode` to the list of its busy blocks. Through this map, I can easily figure out which block is free and which is not in a certain `DataNode` without communicate with it. Every time I assign a new block to a file, I would add its block number to the certain list in this `HashMap`.

Changes of the Architecture

I read the Java Doc of `FileChannel` and I found that its `fileSize` and position is `long` rather than `int`. However, the skeleton code given uses `int` to specify the `fileSize` in the fields while use `long` to specify `size` in the argument of `truncate()` method. I feel that maybe I should follow the official Java Doc. So I change the `fileSize` in `SDFSFileChannel` from `int` to `long`.

When I first try to run the program, I keep get error messages. I wonder why. Then I realize that the `NAME_NODE_PORT` given is the same as `DATA_NODE_PORT`. Since I have to test it on the same machine using the same localhost, I change one of them to avoid interruption.

Since the system will be deployed on different machines, I think it is better to clearly separate three parts in the code level. So I use `Module` in IntelliJ IDEA to

organize the `NameNode`, `DataNode` and `Client`. Since the three parts has some shared classes, namely some entities transported between them, I also create a mutual module called `share` and add some module dependencies.

I think port information should be defined in the protocol so that both client and server can have an access to it. Thus, I move `NAME_NODE_`PORT and DATA`_NODE_PORT` to its corresponding protocol interfaces.

Extra Work

I haven't done any extra work yet. But I guess maybe I should try to make the system run faster and make a user-friendly client interface which means I have to add some protocols such as `ListDir` (to list all the file and directory under a specific directory) or something.