Design Document

Utkarsh Kajaria: 5205243 *Shruti Mahale: 5160832*

We have implement a simple distributed file system in which multiple clients can share files together. The file system contains one coordinator node and several server nodes. Basically, the coordinator is also a server node, but it is also a control point to form quorum and implement concurrency.

Thrift Files:

1. Node.thrift: Contains one thrift struct. It is "node" which is data structure of the nodes in the File System. A node contains IP, port, and its ID which is randomly assigned to it by coordinator.

2. ActualNode.thrift: This thrift file contains the thrift call functions the Coordinator and server nodes makes to each other.

Components in File System:

1. SNServer: This is coordinator server file. Apart from read and write quorum formation and handling read and write requests, it also handles eventual consistency. Also, concurrency is handled by coordinator node using locks.

   We have parameterized SNServer to get Nr, Nw, NTotal from command line.

2. NodeServer: This is the node server file.

   Whenever, you invoke SNServer and NodeServer, a new directory in created inside root/export/scratch folder where all the files are written and read from.

   We have parameterized SNServer to get Port number.

3. NodeHandler: This is the handler file for both Coordinator and Node Server. It contains most of the methods relevant to read and write operations.

4. Client: This is the client code. There are four types of client based on their workload. Read heavy, Write heavy and one with equal number of consequent reads and writes and fourth one provides a way to fetch the state of the node at any given time i.e it fetches all the files on a particular node at any point of time. This provides a UI for knowing which files are present in a particular node.

   We have parameterized the Client to get op (as defined below) and number of operations (numberOfOps) the client will perform. op indicates the kind of client that you want.

   If op=0, then it is a read heavy client. (90% of numberOfOps are reads and rest 10% are writes)
   If op=1, then it is a write heavy client. (90% writes of numberOfOps and rest 10% are reads)

If op=2, then it is a client having equal read-write load. (50% reads and 50% writes)
If op=3, then it is a client to fetch the state of the file system.

It also prints 'Total Read Time' and 'Total Write Time'

Read Operation:

Client.java is the client file. Client requests that it wants to read a file called filename.1 from the file system.

The name of the file is filename followed by the version number. For example, the 5th version of file Apple will be Apple.5

So, when the Client wants to request a read operation on file Apple, first, it makes an RPC call to the coordinator to give it a random node (giveRandomNode). At coordinator, there is a list of node which are currently alive called activeList. This list not only involves Node Servers but also the coordinator itself. The coordinator chooses a node randomly from the activelist and returns it back to the client. The client then calls requestRead on this random node asking to read the file Apple if it exists on that node. Once the request reaches the NodeServer, it calls the coordinator. The coordinator forms a read quorum. The read quorum is formed by picking up 'Nr' number of random nodes from the activelist. Now, the coordinator calls a function called getNewestVersion which gives us the most recent copy of the requested file present on the quorum. In readNewestVersion, the coordinator makes a RPC call to all the nodes in the read quorum and fetches the max version of the file. This max version is stored and the node in which this most recent file is present is also stored. Then, coordinator makes an RPC call to this node which has max version of file and reads the file. A data structure called 'request' is returned which contains the content of the file and status flag which indicates if it was a successful read or not. The data structure is returned back to the client and thus the read call ends.

Write Operation:

When the Client wants to request a write operation on file Apple, first, it makes an RPC call to the coordinator to give it a random node (giveRandomNode). The coordinator chooses a node randomly from the activelist and returns it back to the client. The client then calls requestWrite on this random node asking to write the file Apple on that node. Once the request reaches the NodeServer, it calls the coordinator. The coordinator forms a write quorum Nw. The write quorum is formed by picking up 'Nw' number of random nodes from the activelist. Now, the coordinator calls a function called getHighestVersion which gives us the most recent copy of the requested file present on the quorum. In getHighestVersion, the coordinator makes a RPC call to all the nodes in the write quorum and fetches the max version of the file. This max version is stored. The new file to be written should be named as filename.(maxversion+1). Then, coordinator makes an RPC call to all the node in the write quorum, thereby writing the most recent file in all nodes write quorum. A data structure called 'request' is returned which contains the empty content and status flag which

indicates if it was a successful write or not. The data structure is returned back to the client and thus the write call ends.

After every write operation, we print out the files present on that node. (UI for files on each node)

Concurrency:

Concurrency is implemented at coordinator by using read lock and write lock. We have used Java locks to implement concurrency which gives us read-read concurrency but, no currency in read-write or write-write.

Consistency:

Consistency is maintained at the operation level and not file level. All write operations are sequential.

All read-write operations are sequential. All reads are concurrent.

Eventual Consistency:

Coordinator handles the eventual consistency to be applied on the file system. Once the coordinator server code is invoked, it calls the replicate method periodically. It check if the coordinator has no operations in its queue, then it should write the files written in the write operation to all the other non-quorum nodes. For every file, the list of non-quorum nodes for the latest write operation is saved and we assume that the coordinator has access to this.

Directory Structure at each Node Server to read/write files:

All the VMs run on distributed systems and hence there is an underlying consistency which exists on all of them. Hence, we whenever we write a file on one VM it is replicated on other VM's as well. But, since the scope of this project involves implementing your own consistency model, we had to find out a directory that will be local to the node and won't replicate files across all servers.

One such path is in root directory. Inside root, if we go to export/scratch, then you can create new files and directory which are present only on that VM and not others. Thus, when a node server starts, we create a new directory inside the root/export/scratch called "kajar_mahal". All write and read operations are

Hard cording required:

SNServer's IP and Port Number should be hardcoded in SNServer file.