**MP3 Report**

**Design:**

Our system do not uses master to support file operations and process joins and we are using gossip to sync filelist and memberlist.

For synching memberlist, whenever a machine detects some change, its sends its memberlist to two random machines in the system and these machines selection 2 more random machines and in this way the memberlist is propagated. On receiving the failure message, it compares the received member list with its local memberlist and if its finds an entry with more recent heartbeat counter, its updates the entry in the memberlist. Local timestamp for the updated entries are also updated based on local time.

For synching file list, we are using gossip. When a machine detects some change in its filelist it sends gossip messages to two random machines, which then forwards it in the same way. When a machine receives gossip message for syncing filelist it updates compares the filelist with its own filelist and if the updates the entries in the filelist for which the heartbeat counter is higher in the received gossip message filelist. It updates the local timestamp for the all the updated entries in the filelist.

Failure detection is based on gossip too, when a machine receives a gossip message from another machine it increments its heartbeat counter and update the local timestamp for the remote machine entry in its local memberlist. When a machine don’t get a gossip heartbeat message for Tfail time, we change the state of the remote machine in the memberlist to “toBeDeleted” and machine waits for sometime to make sure every machine in the group have set this file “toBeDeleted” and then it deletes the file, it deletes it from the filelist.

**Message:** We are using Google protocol buffers for our messages.

Our system uses following messages:

|  |  |
| --- | --- |
| TCP Messages | UDP Messages |
| **Join** – used by processes to join | **SyncMemberList** – used to sync memberlist between processes using gossip |
| **Put** – used to put file in SDFS | **SyncFileList** – used to sync filelist between processes using gossip |
| **Get** – used to retrieve file from SDFS |  |
| **Delete** – used to delete file from SDFS |  |
| **ReadyToPut** – used to tell client that server is ready to receive the file |  |
| **ReadyToGet** – used to tell the client that server is ready to transmit the file |  |

**File Operations:**

**Put:**

When a machine1 wants to put file in SDFS, it randomly selects a good machine machine2 and send a put message. Machine2 upon receiving the put message will prepares the local file server to receive file from machine1 and machine2 will then send a readyToPut message to machine1 to tell machine1 that its ready to receive the file. Machine1 on receiving the readyToPut message sends file to machine2. Once the file is stored on Machine2, the replication thread will detect the new file and will replicate it.

**Get:**

If a machine wants to get a file, it can simply check its filelist and if the file is present in the filelist it will retrieve file from the machine having the primary replica, if machine having primary replica is down, it will retrieve file from the secondary replicas. As the files are immutable, we just need to retrieve a single copy.

**Delete:**

If a file needs to be deleted, the machine wants to delete the file from SDFS checks its filelist for the machines having the replicas of this file and sends delete request to these machines.

**Replication Strategy:** Our replication doesn’t depend upon the master. Any machine if can replicate the file, if it observes that file is listed in the filelist but have less replicas than required. We have a scanning thread for replication and after a specified delay, its scans the local filelist and check if any file have less than the required replicas and than replicate any files which have less than required replicas. Our design makes sure that at anytime system have correct number of replicas, in case there are less machines up in the system than the number of replicas it will replicate create replicas limited by the number of working machines.

**Measurements:**

1. **Re-replication time and bandwidth upon a failure (measure for 1GB file)**
   1. Re-replication time:
   2. Re-replication bandwidth:
2. **Time between master failure and new master being reinstated**

Our design does not need master, so we are not doing master election. It can support operations without master.

1. **Times to read and write one file of size 1 MB, 10MB, 100MB, 1GB, under no failure**

|  |  |  |
| --- | --- | --- |
| File size | Read time | Write time |
| 1 MB | 15.2 ms | 43.8 ms |
| 10 MB | 154.6 ms | 286.6 ms |
| 100 MB | 1083.6 ms | 2645.2 ms |
| 1 GB | 12941.4 ms | 43937.6 ms |

Graph 1

Graph 2

1. Time to store entire Wikipedia corpus into SDFS with 4 machines

We are using two replicas, and it takes 197.805 seconds i.e. 3.29 minutes. We test it on laptop because EWS do not allow us to store large Wikipedia dump.

**Extra Credit Work**

File operations in our system are not done through the master. Every machine is running three server processes, one UDP server process for gossip, one TCP server process for Google protocol buffers message and another TCP server process for sending and receiving files.

Every machine executes file requests by itself. When user wants to do some file operations, the local machine sends requests to appropriate remote machines and then do the file operations. For example, if we want to put the file into SDFS, the local machine can put the machine can choose any machine and sends file to that machine using our put message system (explained above). This machine will store the file and the replica manager on this machine will check if the file has required number of replicas, if SDFS doesn’t have the required number of replicas of the file, this process will select random machines from the filelist and will put the replicas of the files in these machines (again using our put message system). Get operations are simpler, as the files are immutable, we just need to retrieve on copy for the machine having the primary replica. Every machine can handle the get operation. For delete operation, we need