**CORRECTNESS**

**Hash Function** –The hash function has been designed in this fashion –

It converts a file name to a number by adding up the ASCII values of all the characters in the file name, we then perform a Modulo7 operation on that sum to find out which one among the seven servers would the file reside in. The other two servers are determined by adding 1 and 2 respectively and performing a Modulo 7 on that number again.

**Servers Available**: S0, S1, S2, S3, S4, S5, S6

Example –

**File Name** – ‘ABC’

**Hash Value Calculation** –

ASCII of A – 65

ASCII of B – 66

ASCII of C – 67

Sum of ASCII – 65 + 66 + 67 = 198

Hash Value – 198 % 7 = 2, (2 + 1) % 7 = 3, (2 + 2) % 7 = 4

**Selected Servers** – S2, S3, S4

I have used file names **‘A’ through ‘J’** for the project. And as per this design we will have the following servers chosen by the hash function, hence these files will only be present in these 3 servers.

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Server 1** | **Server 2** | **Server 3** |
| A | S2 | S3 | S4 |
| B | S3 | S4 | S5 |
| C | S4 | S5 | S6 |
| D | S5 | S6 | S0 |
| E | S6 | S0 | S1 |
| F | S0 | S1 | S2 |
| G | S1 | S2 | S3 |
| H | S2 | S3 | S4 |
| I | S3 | S4 | S5 |
| J | S4 | S5 | S6 |

**Voting Protocols –** To perform any write operation we need to ensure that the request reaches at least two servers and both the servers vote in favor of the write operation, and we allow the write with at most one server being unreachable.

I have simulated the unavailability of the servers S2 and S3. With disruptions for whatever reason on Server S2 and S3 we will not be able to write to write to file A, G and H. Since 2 among the 3 servers are down and we won’t have sufficient votes to write to these files.

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Server 1** | **Server 2** | **Server 3** |
| A | S2 | S3 | S4 |
| B | S3 | S4 | S5 |
| C | S4 | S5 | S6 |
| D | S5 | S6 | S0 |
| E | S6 | S0 | S1 |
| F | S0 | S1 | S2 |
| G | S1 | S2 | S3 |
| H | S2 | S3 | S4 |
| I | S3 | S4 | S5 |
| J | S4 | S5 | S6 |

**Safety** –

**Read** –

Read requests are sent to only one among the three servers and even if one of them are reachable we would be able to achieve consistent read as long as consistent writes happened. But in case of a server failure, or a partition it is possible that we are not able to achieve consistent read results since we might have reached a server in the partition which does not have majority servers for that object, or when a server recovered and the file has not yet been synced, since writes have been skipped on that server while it was performed on the other servers.

**Write** –

Writes are consistent and it is ensured that the writes which are permitted are done in the same order on all the servers. I have used a Priority Blocking Queue for each file on each server, on which write jobs are placed based on the logical time stamp of the client which has made the write request, ties broken using their client ids. So, in case multiple clients have made a write request to the same file at different time, jobs are processed in the order in which they came in, in case there are multiple concurrent write requests from different clients the requests are arranged on the basis of their client ids. Now when processing each job request from the front of the queue, the server sends a Vote message to the replica servers after putting a lock on the file so that no other requests for a write to the file are permitted, it waits for an acknowledgement of the vote from the replica servers – If Vote is Cast by the Replica Server – The vote count increases to 1/2 and hence we execute the critical section and write to file, send a release message and remove the job and remove the lock on the file, If Vote Denied by the replica server – The vote would be denied by the server if there is some other job for the same file that is currently running, and the server which had sent the vote request will wait and release all locks and resend a vote message again after some time until the same is accepted by that server and the same write request is at the head of the queue in the replica server allowing it to cast its vote.

Thus, at a time each server will allow only one lock for each object in that server, that is only one write request would be entertained, and only if a minimum of the majority (2 of 3 servers) vote for the request are received, then the request would be processed. All these guarantee that all write requests are processed in the same sequence in all the servers allowing us to achieve consistent copies of objects on all the servers.

However, in the event of a failure of a server or a server being unreachable, it is possible that the copies become inconsistent in that server since there might be incoming write requests when the server was down and those were processed during the time the server was down. However, it is possible to bring the server which failed to be at par with the other servers, as we can take a delta by maintaining version numbers and logs and eventually when the server recovers we can bring it in sync with the other servers by sending requests to check the version numbers and updating the delta to the object on the failed server. Thus, we can again achieve consistent write results with a small tweak, which is beyond the scope of this project.

If the networks are partitioned the partition with at least 2 servers will be able to process the request. Also, it is possible that no writes are processed if 2 out of 3 servers are down or the server is partitioned in such a manner that no partition has a majority of the server for an object.

**Liveness**

All requests are ordered on the basis of client timestamps and ties broken by client ids in a queue and are processed from the top of the queue eventually receiving majority votes from replica servers or getting discarded in the event of not receiving a majority after waiting for a timeout, thereby ensuring that there is no starvation. There is no possible scenario for a deadlock as well since to process a request a server needs to lock an object and requests for vote for the job from the replica object servers – it can either process the request on receiving majority of votes, or if it does not receive majority then release the lock and request to vote again later. This ensures that resources are not locked by any two events at the same time. Also, inability to receive majority for longer time in case of a failure (of 2 out 3 servers) we can discard the job by waiting for a timeout. Thereby handling all the possible cases of a deadlock.