**Software Design Document**

**Introduction:**

This software is designed to demonstrate the algorithm that allows for multiple servers and multiple clients to communicate with each other for the purpose of storing, replicating and retrieving data.

**High Level Components**

1. Server:

This entity is responsible for processing the read, insert and update requests from the client. It also forwards the request to the servers where the clients wants the data to be replicated. It waits on the successful replies from the replicated servers after locking the data in the memory and then sends a success answer back to the client upon successfully unlocking the data in the memory.

1. Client:

The client is a command line tool that takes the command from the user to read/Insert/update an object. It uses a hash function applied on the data to pick the primary server and two backup servers when the user wants to insert or update. There needs to be at least one backup server available for client to issue the request. The client then requests the server to update/insert data and includes the backup servers where it wants the primary server to replicate data. For read, it allows the user to read from any of the available servers where the data hashes to.

1. Object:

This is the data unit in the software. It is composed of a key and value pair. The client hashes the key of the data to find the primary and backup servers. The value for our purposes is the string.

1. Memory:

For our purposes, memory is simply a cache memory. Not only does it store the object but also provides the context memory (metadata) for the object on which the primary server operates. The metadata allows for matching the requests made by the primary server to the replies generated by the backup servers and ensuring the correct number of replies are received before committing the data

**Class diagram**



**Sequence Diagram**



**Explanation of the Algorithm**

Safety

All objects are hashed to a primary server which is responsible for updating the backup servers specified by the client. So all clients will contact the primary server for their update and therefore mutual exclusion is achieved that is it cannot be that any client will contact the backup server for updating the object if the primary server is up. This will preserve the order of the updates since the primary server does not serve two requests concurrently.

In the event of crash of the primary server, the client program simply assumes the next in line hashed server is the primary server. That is if S1 is the primary server and it crashes, S2 becomes the new primary server. This happens simultaneously at all clients as they all detect TCP links failure with the primary server.

The program can be optimized by using a integer as a Key(which is done in the program) so that a simple Mod function is required to select the primary server.

Liveliness

The hash memory is protected with the lock. Once an object to be updated is taken out of the hash memory the hash memory is unlocked. The object is then locked. So the server is free to serve any other insert/read/update requests on other objects.

**Problems**

There are some problems with the algorithm. Mainly are few assumptions that can be violated. For example the primary server crashing in the middle of communication with the backup servers. In this scenario some backup servers may be updated and some not. Also, there is a no way at this point to elect a new primary server and informing that to the client that made the request.

Also the roll back mechanism is not implemented. Such as when the primary server already updates some back up servers and then one of the backup servers returns an error. In this case the server has to roll back on all the backup servers that were successfully updated.

Also relying on the TCP for detecting unresponsiveness is again a problem. It may be that server TCP mechanism is working but the application is not. In that case requests to the primary server will not be completed.

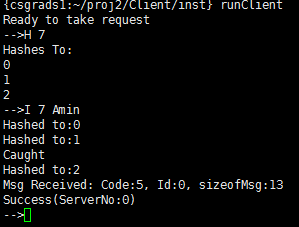
In terms of performance, the primary server can become the bottleneck for the updates/Inserts.

**Test Data**

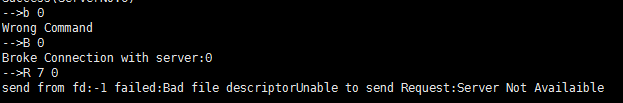
**Scenario # 1 (Crashing of a Server)**

-7 Servers

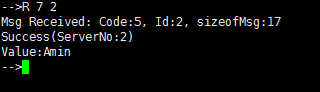
Insert 7 Amin

****

Crashing Server 0 using break command(b 0) and then reading failed(This just severs the link, exiting server 0 will have the same effect)

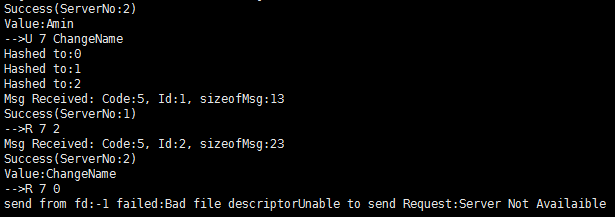


Reading from backup server 2 succeeds.



Update now happens at server1

Backup server is server 2 which is read from successfully



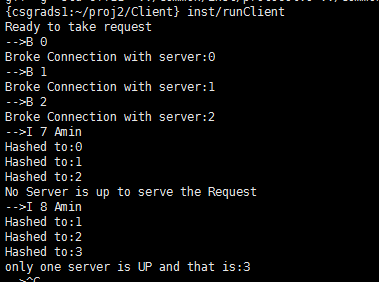
**Scenario # 2 Partitioning**

2 clients, 7 servers

client 1 served by servers 0,1,2

client 2 served by 3,4,5,6

Client #1



Client #2

