Project 4

-Hamsika Rammohan(903667610)

-Nihar Kalsekar(903729147)

**Overview**

In this project, we had to implement a distributed key-value store. GRPC is used between the nodes to communicate. The system is scalable, available and resilient to temporary node failure. The manager is always running and takes care of registering storage nodes/clients. The client can add/edit data and the system will return the most recent data.

Here is a diagram showing a rough flow of the application:

Diagram

Description automatically generated

**System Components:**

**GT Manager:**

The manager file used the following data structures:

A screenshot of a computer

Description automatically generated with medium confidence

1. Priority Queue: The priority queue stores the list of storage servers available. When the put function is called, the priority queue pops out the number of storage nodes needed based on the replication factors and inserts the key there. After this, it increases the priority and adds the storage nodes back to the queue
2. Lookup Table: This is a map of maps that stores the client id, key and the list of storage nodes where the key is present.

The manager has the following RPC calls when it acts like a server:

Text

Description automatically generated

1. Register Node: The storage sends a grpc call to the manager to register it.
2. InitClient: This grpc call is from the client to the manager to intiliaze the client
3. CM\_Put\_Key: This call is from the client to the manager to insert the key in the storage node
4. CM\_Get\_Key: This call is from the client to get the value of a particular key from the storage node.

We have used the concept of **nested RPC Calls** in our design. The manager acts both as a server and client for the get/put grpc calls. When a client send a get grpc call to the manager, the manager sends another grpc call to the storage node, which then sends a reply and based on that the manager sends a response to the client. Similar is the case with the put call.

**GT Storage Node**

The storage node has the following data structures:

Text

Description automatically generated

The stored\_values is a map specific to each storage node that stores the key-value mapping.

GRPC Calls:

The following are the grpc call made when Storage Node acts as a server:

Text

Description automatically generated

1. Put\_Key: The manager sends a GRPC call to the storage node to put a key in it’s hashmap
2. Get\_Key: The manager sends a GRPC call to the storage node to get a value of a key

The storage node registers itself with the manager first before these put/get key functions come into play.

**Client API Calls**:

1. Register\_Client: This API call is sent to the manager to register the client with the manager, the client id is inserted into the lookup table data structure(map of maps)
2. Get Key: The client sends a grpc request to the manager with the key whose value it wants. The manager then checks the lookup table to find the list of servers that have the value and makes a grpc call to the storage nodes. The storage node returns the value to the manager who in turn returns the value to the client.
3. Put Key: The client sends a grpc request to the manager with the key- valuepair . The manager then checks the lookup table to find the list of servers(storage nodes) with the most capacity and pops them out on the basis of the replication factor. The manager then sends a grpc request to the storage node to insert the key-value pair in their stored\_value hashmap

**Design Principles**:

1. Data Partitioning:

We do not store data in a single node and have multiple nodes available to store data. There might be times when there is a node failure and our implementation is able to handle it. We use data partitioning to distribute the loads evenly across multiple nodes. We have implemented this load balancing using priority queue. When a node is created, it registes itself with the manager while sharing its ID.

1. Data Replication:

There maybe instances where there is node failure/node is down. To deal with such cases, we have data replication enabled. Whenever there is a put key request, the manager pops out the node with most capacity from the priority queue and then inserts the key/value pair into that node. The priority of the storage node is updated and added back to the queue.

1. Data consistency:

Data consistency is the pillar of our implementation. Any updates to the key-value nodes are done real-time. Once a client sends a request to put key, even if the key exists, the change is made across all the nodes.

**Design Tradeoffs**

We have focused more on ‘Data consistency’ and node faulire. We update all the storage nodes real time and this results in slower speed. We could have just updated the lookup table and periodlly updated the nodes if we had focused on faster response.

**Implementation Issues:**

1. We could not automate the creation of all the storage due to lack of time and scripting knowledge, hence we manually create the storage nodes.
2. Our design includes a lookup table(map) which has the format map<string,map<string, set<strings>>>. The outermost key field refers to the client Id. Hence we have to input the Client Id in and The Put\_Key and the Get\_Key functions as an argument. (We could certainly do away with that in an improved design)
3. Our design takes a long time to run as we are not breaking the loop during get calls

Screenshot

Graphical user interface, text, application, chat or text message

Description automatically generated