CS-451 PROJECT

Distributed, Scalable, Consistent and Fault tolerant Key-Value Store

FINAL REPORT

Team:

1. **140050020 (Govind Lahoti)**
2. **140050046 (Vishwanadh Rapolu)**
3. **140070001 (Sohum Dhar)**

Design

Basic Idea

The underlying idea has been borrowed from Chord with slight modification. Let k be the replication factor. 2^m be the total number of nodes in the Chord. We will use a consistent hash function h and derive k consistent hash functions using h, $h_i(x) = (h(x) + i)\%m$. Each node will be responsible for some part of keyspace (as in chord) for each hash function (each dimension). The node id here will be a vector of k-dimension.

Node Join: If a new node (say N) joins, it will do so by contacting some node (say M). M will become leader for handling the joining. It will appropriately assign a node-id (k dimensional vector) to N. N will then set-up heartbeat channels with k-successors (each corresponding to one dimension i.e, hash function). Also it starts the procedure to construct the finger tables. The remaining nodes update the dimensions for each Key-space so that the invariant "k replicas are present along k successive nodes and along different dimensions and the dimensions are in the order of node ids in the chord" is maintained. Here the heartbeat channels doesn't change between the remaining nodes doesn't change.

Node Leave: The node leaving the system voluntarily inform its successors. The corresponding data will be transferred to each of its successors (just like chord) and heartbeat channels are setup. The dimensions for the existing Key-spaces are updated.

Key Lookup: When a user adds a key, it asks in say node N. The node N finds the hash in fixed direction(least hash) and finds the key as in basic chord along this dimension. Since for sequential consistency, the protocol used is Local Read, read never blocks so it returns the value immediately.

Key Insert: When a user adds a key, it asks in say node N. As in lookup we go to a node that has the key. This node initiates the write phase of Local Read Protocol. On a write request w(x,v), a message w(x,v) to write the value is sent through a total ordered broadcast channel and blocks. For a key always a node which has the key and its predecessor doesn't have the key always starts the write for that key and sends to other replicas in a FIFO order. Since always a single node is sending the request and FIFO, total order is ensured here.

Key Merging: Key merging is done as in chord but along each dimension. If a node fails or leaves and let say a keyspace K is present in dimension j. Then now the next alive process should contain the key space in dimension j. So the successor shifts K from j+1 dimension to j and shift all other key spaces to the previous dimension. The 2nd successor which has k in j+2 dimension now shifts K to j+1 and shifts other key spaces to the respective previous dimensions. A node has a successor which doesn't have K and we

create the new replica in that successor. Similar shifts of Key spaces happen during joins also.

Failure Detection and Handling: Each node will have k different successor, one in each dimension. Each node maintains a heartbeat channel each one of them (i.e. a k-clique) i.e a node has a heartbeat channel for each node that has a key partition in any of its dimensions. If a node (say N_i) fails, the other $k-1$ nodes will know because of the heartbeat channels. Node leaves or fails:- Consider a partition in keyspace $K_{Ni,j}$ corresponding to the jth dimension of node Ni. The other t nodes(corresponding to the partition in the other dimensions $\neq j$) then initiate the protocol to transfer the keys in the partition to the successor node of N_p which is has the keys of the partition and its successor doesn't have them. This is done for all partitions in all dimensions j.Basic Chord just transfers the keys to its successors where in here the keys are transferred to the node which previously doesn't have the keys.

Load Balancing: We have a particular key stored on k nodes (k is the replication factor). We use the chord lookup algorithm along a fixed direction but not the same node for all keys, independent of the node chosen by the user. Further, since we change the lookup node each time for different keys, the requests(load) for each keyspace gets distributed uniformly amongst the all nodes. Replicas are only used for backup.

Scalability: The read and write throughput will increase linearly with number of nodes. Our main concern would be delays due to disk read and write. The network delays are much less compared to them. Since our design has ensured equal load at each node, so the length of requests queue (for disk read and write) at each node will be inversely proportional to number of nodes in the system, the read and write throughput improve linearly with number of nodes.

For any set of N nodes and K keys, with high probability , when an $(N + 1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node.In an *N*-node network, each node maintains information about only $O(log\ N)$ other nodes, and a lookup requires $O(log\ N)$ messages.

Testcases:

Test1 :- Create of replica as a node leaves or gets killed.
Setup :- replication factor = 2. Create 5 nodes along with master node. Create 2 read user which randomly read a value. Create 2 write users which randomly writes a key-value. Now kill a node. Wait for some time and exit.
Result:- k-replicas for each key are present.

Test2 :- Replying to a request in the process of creating a replica
Setup :- replication factor = 2. Create 5 nodes along with master node. Create 2 read user which randomly read a value. Create 2 write users which randomly writes a key-value. Now kill 3 nodes. Wait for some time and exit.
Result:- Request are handled even if k-replicas are not present. The nodes have the key space correctly split among them.

Test3 :- Throughput and load balancing
Setup :- 100 user send look up requests to random nodes continuously. We calculate the number of requests served by the system per second.
To simulate the practical behavior of time taken when reading from slow disc, we add a sleep time of 0.01 sec for each request at any node.

The experimental results are as follow:

Number of nodes	Number of requests per second
1	94
2	187
4	322

Test4:- Chord Protocols
Setup:- replication factor = 2. Create 5 nodes along with master node. Create 2 read user which randomly read a value. Create 2 write users which randomly writes a key-value. Now kill a node. Wait for some time and exit.
Result:- Finger tables are correctly updated. The division of key spaces is correct.

Test5:- Sequential Consistency
Setup:- create 6 nodes. Create 10 users which write for a same key and then read the value. Based on the values obtained create a happened before relationship graph and check if there are any cycles
Result:- No cycles detected. Total Order in writes is followed.

GIT REPO :- https://github.com/govindlahoti/Distributed-Key-V alue-Store