CS425: Distributed Systems – MP3 Report

Gopalakrishna Holla V (hollava2)
Alok Tiagi (tiagi2)

Design and algorithm

Key value store design:

The key value store at each machine is a hash table. This key value store supports the four operations, insert, lookup, update and join locally. The value field is a string. So any serializable object can be stored in the key value store.

Routing of keys:

Our setup uses a hashing approach. Machine IDs are hashed using a consistent hashing mechanism and arranged in a clockwise ring (in ascending order of their hash values). Each machine joins with a network ID that is a concatenation of the IP of the machine and its join timestamp. This network ID is hashed and the hash is entered into its entry in its membership table. The machine then gossips its membership list as usual. A member receiving this gossiped membership list adds all the details including the node hash to its own list. In this way, every machine knows the hash (and hence the ring position) of every other machine.

Keys are hashed using the same consistent hashing mechanism and stored at the first machine on the ring with a hash value greater than the hash value of the key (on the ring in the clockwise direction, not necessarily higher in value).

When a command string is received from a client, the receiving machine hashes the key first and figures out which machine the request has to go to. If the receiving machine is the owner of the key, it performs the operation locally. If it is not, then it acts as a client and opens a connection to the right machine (which it knows through its membership list) and forwards the command string. The second machine then performs the same steps (and since it is the right machine, performs the operation) and returns the result. The result is then forwarded to the client.

Consistent hashing mechanism:

We use SHA1 for consistent hashing. The network IDs and the keys are hashed using SHA1 and the first 8 bits are used as the final hash value

Join and leave:

When a new machine joins the system, owing to the gossip mechanism, each existing member gets information about the new joinee. Whenever a new member is added to the membership list, each machine examines the hash value in the entry of the new member to figure out whether the new machine that joined now becomes an immediate predecessor to it. If it does not, then there is nothing to do for the machine. If it does, then the machine must transfer some of its keys to the new machine. All the keys in the local key value store are examined to see whether they now belong to the new machine. If any such keys are found, a well-formed insert command is created for each key and the key is deleted from the local store. Then the machine acts as a client and opens a connection to the new joinee and sends over the commands to the new machine that performs the inserts locally.

When a machine decides to leave, it creates well-formed insert commands for each key in its store, waits until everyone has deleted it from their membership lists, then acts as a client and sends the commands to its successor on the ring before exiting. The successor inserts the keys locally.

Why hashing:

We chose to use the hashing approach because of the easy reorganization of keys when a machine joins or leaves, lesser complexity of gossip messages and ease of overall implementation. In the case of hashing approach, each machine at most needs to propagate its hash value to the other members in the system as opposed to the segmented approach where all the machines need to agree upon the number of segments and the distribution of responsibility for the segments, which becomes all the more complex during joins and leaves. The segmented approach probably gives more control over load balancing but at the cost of possible heavy bandwidth usage during reorganization and higher complexity of implementation.

Use of MP1

We used MP1 to track the operations (and any associated exceptions) during the routing and execution of the insert, lookup, update and delete commands.

Handling of stale gossip information:

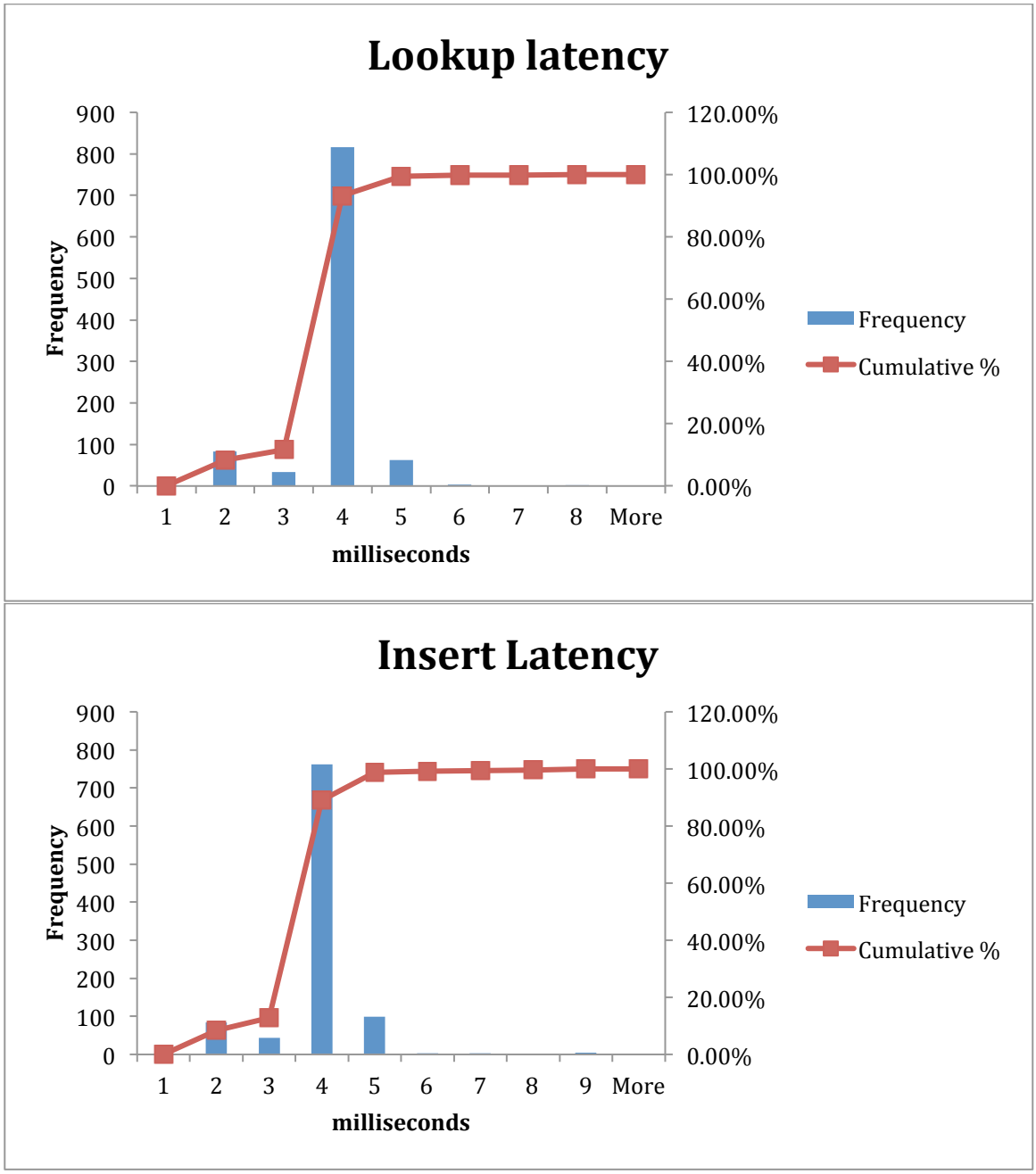
Our design naturally handles all stale information.

Join: When a machine does not have information about a new joinee and routes the key to the wrong machine (this machine can only be the successor of the new joinee), we have two cases. If the receiving machine (i.e the successor) has the new joinee is its membership list already, it just reroutes the command (so there is a total of 3 hops in this case). If it does not have the new joinee in its list yet, then it inserts the key locally. When it receives information about the new joinee, this key will be sent to the new joinee during the reorganization of keys.

Leave: Since a leaving machine sends all its keys only after it is sure that everyone has deleted it from their membership lists, there will be no keys routed to the leaving machine once the machine starts sending its keys

(Graph and discussion for lookup latency when data structure is empty is in the Extra credit section)

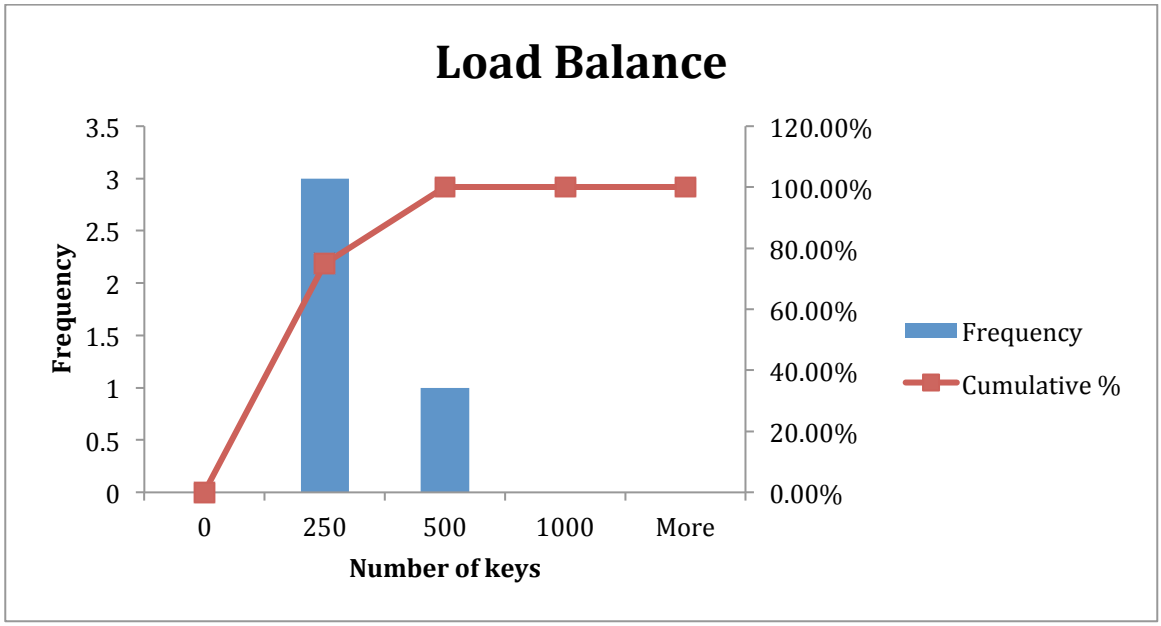
Extra Credit Section

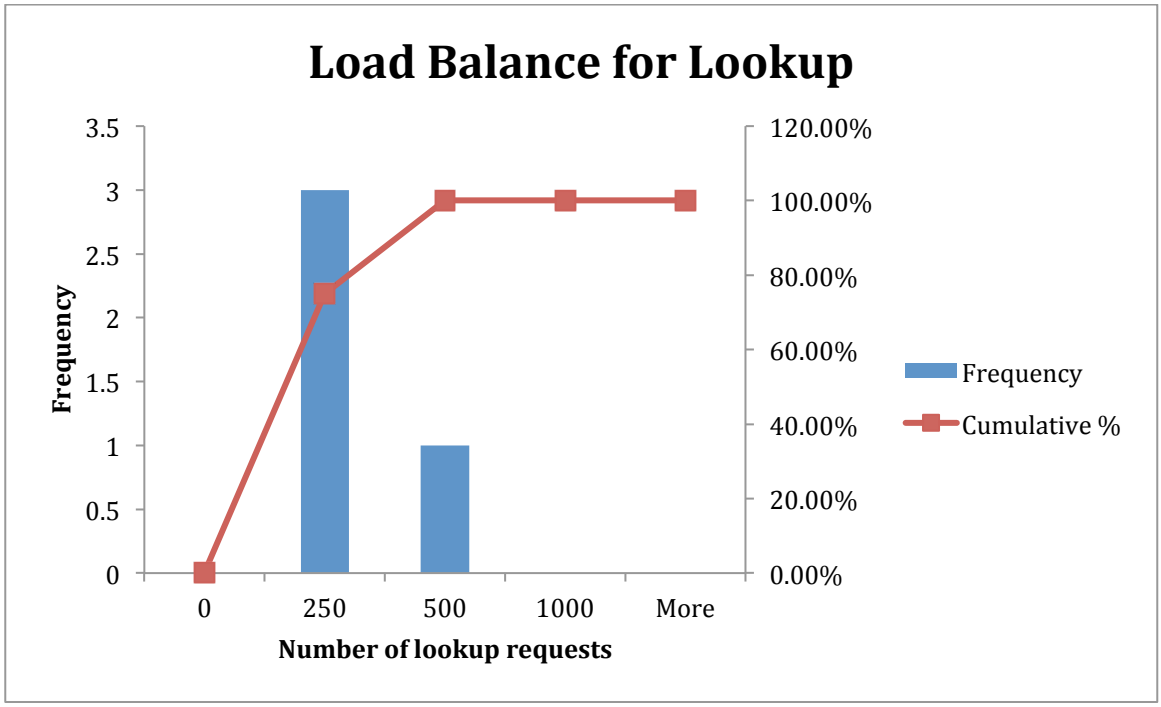


How we performed the experiment: We wrote a script on the client side that picked a random key and performed an insert, lookup and delete on the key (there were no other keys on the 4 servers). A1000 random keys were generated, inserted, looked up and deleted. We plotted the latency of insert/lookup against number of keys that exhibited the selected latency. The graphs above are for a system of 4 servers.

What we gather from the experiment: Both insert and lookup have a latency of about 4 milliseconds most of the time. We found that regardless of the operation being performed, the latency is the same. Additionally we repeated the same experiment on 2

and 3 server systems and found no significant changes in the behavior of the system. This may be due to the fact that, no matter how many nodes are in the system, any operation takes at most one hop from the server that originally receives the command (since we are directly routing the commands)





How we performed the experiment: We wrote a script on the client side that picked 1000 random keys and inserted them into a system of 4 servers. We then took readings for the number of keys on each server. After that we had another script generate a 1000 random keys and try to look it up on the servers (some of the lookups returned no results, but to

discover the non-existence of the key, it still has to be routed to the correct machine and so still counts as load). We recorded the number of lookup requests at each server. We repeated the same experiment five times and plotted the number of keys/requests that reached each server.

What we gather from the experiment: The keys are fairly well distributed with close to 250 keys in each node. But there were cases when the distribution was skewed. This is because the hashing approach offers little control over load balancing. But given a system of N servers, where N is large, the hashing approach might offer comparable load balance as compared to the segmented approach.

Another observation is that the load balance of storage and load balance of lookup are strikingly similar. This can be understood by viewing the load balance of storage as the number of 'insert' requests received. When we look at it this way, we can see that no matter what the operation is (insert, update, delete, lookup), the load balance of requests remains the same given a random set of keys. This is because there is always only a single server responsible for the operation and there is at most 1 hop to perform the operation (because of the design)