CS451: Project Report
Kyklos: A modified Chord DHT

Anmol Arora(130050027) , Pranjal Khare(130050028), Ritish Goyal(130050086)

Apr 26 2017

1 Introduction

To build a scalable, fault tolerant distributed Key value store similar to Cassandra in Go.

2 Underlying design

We plan to build a scalable distributed key value store using the concepts of Chord DHT and modifying it to support data replication and fault tolerance in lines to what is implemented in Cassandra. We are using following design choices for supporting the required features:

- Chord-DHT for consistent hashing : Consistent hashing function takes care of uniformly dividing the keys and nodes on the available keyspace.
- Random N node replication : Random N nodes on the chord ring act as replicas(where N is the replication factor).
- Heartbeats: Every node sends heartbeats to its successor periodically to check if its alive. When failure is detected, the successor lists are updated immediately and finger tables are updated over time. We are assuming that no more than two successive nodes on ring fail simultaneously, since we are maintaining two successor lists. This can be extended to higher order lists to ensure stronger failure detections.
- 2PC Protocol : 2PC protocol is used to enable atomic write transactions, with random backoffs to redo aborted transactions.
- Load Balancing: The read operations are rooted to one of the replicas randomly. Since all the replicas are consistent, read from any one replica is sufficient. Randomly choosing replicas ensures load balancing.

3 Design Details

3.1 Data maintained at a node

Each node stores information about its two successors and a predecessor. Each node also has a datastore for storing key-value pairs (HashTable) and a finger table. The finger table enries are

fixed periodically as per the chord protocol. There is a temporary datastore which is used for 2PC protocol. It keeps the key-value pair till the commit message is sent by the coordinator

3.2 Addition of node

A node should already know any one of the nodes already a part of the ring(called as host). New node asks host to find out the successor of its id. The node then initializes its finger table entries, sets its successor node and lets its successor know of its presence, so that it can set its predecessor node. The successor also sends the relevant keyspace to this new node. The new node then contacts other nodes who's finger tables entries might be affected by its addition. Stabilize and fix fingers functions ensure that evenutally the predecessors and fingers tables will be consistent.

3.3 Replication

Every write operation is replicated in N random nodes on the chord ring chosen by hashing the 'key_i' values (i ϵ 1..N), where N is the replication factor. This in turn creates N copies of the same key-value pair across the ring. Any write operation will update all 'key_i' values. This is done through 2PC protocol to ensure that all replicas are consistent.

3.4 Load Balancing

A read is load balanced by getting the value from a 'key_i' hashed node using a random value of i (i ϵ 1..N). This favours the speed of read operations as read is done on one node only.

3.5 Failure Detection and Handling

Failure detection is done through heartbeat mechanism between a node and its successor. We assume the failure of maximum one node at a time. A node periodically sends a heartbeat to its successor to check if it's still alive. If a reply to the heartbeat is not received, the node assumes the successor has failed and asks its next successor for the information about its successor and also asks it to update its predecessor since its predecessor has failed. This way, the keyspaces of the nodes are updated to remove the failed node.

3.6 Node leaving at will

The node transfers the keys and sets the successors and respective predecessors correctly.

3.7 Handling Requests

Whenever client wants to give read or write requests, it can contact any node in the system. This node will act as leader(in case of write operation) and now handle the request.

3.7.1 Favouring reads over writes

We are assuming that the majority of the requests will be read requests and hence our system is optimized to serve reads faster compared to writes. We are following Read One/Write All architecture, whereby all replicas are synchronized at all times, hence read request can be handled by any node and write request makes the update at all nodes.

Read: The interface node, which receives the client read request, selects a random values between 1 and N. It appends this node to the key and calculates the hash.This new_key-pair would exist in stable system, since write operations ensure write happens on all key_i (i ϵ 1..N). Since every node maintains the same value, multitude of read requests can be distributed across replicated nodes to achieve load balancing. In case an key not found error is found, this is most likely due to the said node being deleted. In such case the client will try the same procedure, but with another value for i, hence routing this request to another replica.

Write: Every write request updates value at every replica. Hence all replicas are consistent. However this causes the writes to take longer time. To ensure that write operations happens on all or nonr(atomicity) 2PC commit protocol is used.

3.8 Consistency

Concurrent read and writes on different keys are allowed to happen at the same time. If there are two writes on the same key, they are done sequentially .Since every node performs the writes in the same order sequential consistency is maintained.

3.8.1 2-Phase Commit Protocol for write operations on replicas

For handling a write operation (a series of writes for replication), we need to make it atomic i.e. if the transaction fails mid-way we need to rollback the changes. In order to do so we take help of a 2 phase commit protocol. The node which is contacted by the client becomes the leader node and is responsible for taking consensus from the N nodes in case of a write operation to check if they canCommit (which they check by writing in the temporary datastore). If all the nodes send a canCommit, then the leader sends a Commit message to the nodes and actual data is written to the datastore. Else an abort message is sent and no data is written. In case two write operations are performed on the same key, such requests will conflict at the nodes. In such cases the node rejects both the transactions and they get aborted. In this case, the respective leader nodes wait for a random time and then the operations are tried again (random back-off).

4 Demonstrating The Features

4.1 Replication with consistency

Sequence of operations:

```
create 4 nodes
put(1, 2) on any node
get(1) will give value 2
put(1, 3) on any node
get(1) should now return value 3
now kill 2 nodes which store some replicas of (1, 3) pair
get(1) will still return the correct value 3
```

5 GitHub Link

https://github.com/mewtomat/Distributed-Project

6 References

- Cassandra Paper : dl.acm.org/citation.cfm?id=1773922
- Chord-DHT :

  https://piazza-resources.s3.amazonaws.com/ixed3rf8vn32lc/j0arlmter316v8/DHTChord.pdf?AWSAccessKeyId=AKIAIEDNRLJ4AZKBW6HA&Expires=1490028497&Signature=YVlnGFLqD0qEblBrtJAYVajymdo%3D

- Cassandra Overview : http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/ gettingStartedCassandraIntro.html