Chordy: a distributed hash table

Chuan Su

October 9, 2018

1 Introduction

A peer-to-peer distributed hash table stores key-value pairs by mapping keys onto nodes. A node will store the values for all the keys for which it is responsible. Chord protocol specifies how keys are assigned to nodes and further how values are retrieved by first locating the responsible node for the key.

In this report we will present our implementation to Chord protocal and discuss the main problems we encountered.

2 Consistence Hashing

Chord protocol uses a variant of consistence hashing to assign keys to Chord nodes. SHA1 hashing function is used to generate unique keys. It is remarkable that Chord apply the same hashing function to both `Node Id` and `Key` so that the nodes and keys are mapped to the same space, which is the foundation of Chord Protocol.

In our implementation we instead used a simple random number generator to generate "unique" keys for nodes and keys. Even though no hashing function is applied the Chord Consistence hashing princile is followed - the unique keys generated for both nodes and keys are within the same number range, 0 to 1000,000,000,0 in our case.

3 The Ring, or just Predecessor and Successor?

The Ring is more or less the structure of a double-end queue where tail node points to the head node. What makes *Chord ring* special is that it has *self-stabilize* capabilities, in which a node is able to find its `predecessor` and `successor`.

Self-stabilizing is achived through:

1. Request(): node `N` asks its successor for its predecessor `P` and decides whether `P` should be `N's` successor.
2. Notify(): notifies `N's` successor `P` of its existence, so it can change its predecessor to `N`.
3. UpdateStore(): Split responsibilites by handing over part of the data (key-value pairs) in its local store.

4 Node Responsibilites

Each node in the ring has its own responsibility for certain keys. In our implementation, one node is taking care of all keys that fall into its partial closing range ,that is, from (but not including) its predecessor to (and including) itself.

Responsibilities of a node concerning both `key-value` pair insertion and key lookup.

4.1 Adding an element

Each node maitains a storage that stores all the `key-value` pairs that it is responsible for. If the key of the pair to be inserted to the ring is out of the node responsibility the node will simple pass the `key-value` pair onto its `successor`.

4.2 Lookup procedure

`Lookup procedure` behaves quite similar, which also performs responsibilty check. If the key falls into its range (from the identifier of `predecessor` to itself) it will perform a store lookup to retrieve the value. Otherwise it will pass along the query to its `sucessor`.

5 Performance Evaluation

At first test we have only one node in the ring and added 4000 elements to the ring. The time it took to lookup each of the 4000 elements is 148389 ms.

```
P = test:start(1, nil).
test:evaluate(4000, P).
```

At our next test we have 4 nodes in the ring with same amount of elements added to the ring. The time is took to lookup in the `DHT` was 54564 ms.

```
P = test:start(4, nil).
test:evaluate(4000, P).
```

We can see the ring with one node took almost 3 times longer than the 4 nodes.

6 Conclusion

Our implementation of Chord protocol follows the steps described above.

In my opinion, `Consistence Hashing` in Chord schema where nodes and keys are mapped to the same range provides the foundation for a distributed hash table, which allows Chord to distribute responsibilites across nodes and further perform lookups for a given key identifier.