Chordy - a distributed hash table

Xin Ren

October 4, 2017

1 Introduction

In this assignment, a distributed hash table following the Chord scheme is implemented.

2 Detailed Implementation

Following modules are implemented in this homework.

key.erl	Generates a random key for each node and store element. Provides the function to check if a point(key) is between two other points on a ring structure.
node1.erl	The first implementation which only maintains a ring structure; we are able to add nodes in the ring but not add any elements to the store. With stabilization procedure, a new node can find its predecessor and successor, while both of them can update the new node as its predecessor or successor.
storage.erl	Provides functions to initiate a store, add key and value pair to a store, look up value with a key in a store, spilt a store to two parts with a specific range of the keys and the rest, and merge two stores.
node2.erl	The 2nd implementation which implements the store. A client can add elements each of which contains a key and a value by contacting any node in the ring. The node receiving an element checks if the key of the element is between its predecessor's id and its id, if yes, the node saves the element into its store, otherwise, it forwards the element to its successor. A client can also look up an element with its key by contacting any node in the ring. The node receiving a lookup request checks if the key of the element is between its predecessor's id and its id, if yes, the node looks up the element in its store, otherwise, it forwards the request to its successor.
node3.erl	The 3rd implementation which handles node crash. Each node in the ring keep track of a Next node, which is the successor of its successor and monitor its predecessor and successor with Erlang build-in monitor function. When the predecessor of a node crashes, it sets the predecessor node to nil. And when the successor of a node crashes, it sets the successor to the next node and set the next node to nil. The stabilization procedure will guarantee the nil be replaced with correct node and fix the ring structure.

3 Evaluation

Some performance tests have been done towards the 2nd implementation.

A client adds 4000 elements and 10000 elements to a ring with 1, 2, 3, or 4 nodes and looks up all the elements. The time taken by the lookup was evaluated.

	1 node	2 nodes	3 nodes	4 nodes
4000 elements	1139ms	1264ms	1935ms	2262ms
10000 elements	3291ms	4181ms	4930ms	5491ms

As we can see from the table above, the lookup took longer when there were more nodes in the ring. This is because when the number of elements is not big, compare to the time taken by a request sending among nodes, the time taken by searching an element in a store is insignificant. The advantage of chord protocol should be more visible with more elements when the time taken by a request sending among nodes is insignificant by comparing to the time taken by searching an element in a store.

Also during the test, I found that the result was very unstable, for example, when there were 10000 elements distributed to 4 nodes, sometimes the lookup took 5xxx ms, sometimes 3xxx ms, which means the results were influenced by the running environment and not very trustworthy, therefore, further test is not performed.

I also tried to look up all the elements by contacting different nodes in the ring, the result showed that the time taken was different from node to node. The reason for this is that the rand:uniform/1 function we use to generate random key and id can not guarantee that nodes evenly distributed in the ring nor the elements distributed evenly in the ring.

4 Conclusions

Through this assignment, I have learnt chord protocol, how to implement it with erlang, understand that it speeds up the handling of lookup requests by distributing elements to multiple nodes in a ring structure, and the more elements, the more visible its advantage.