Chordy A Distributed Hash Table

jinjia zhang

October 12, 2023

1 Introduction

This assignment is to implement a distributed hash table following the Chord scheme.

I add new features step by step until the final version is implemented. The first task is to implement a key module that generates a random number and checks if a given key is between From and To or equal to To. After that, the first step is to have a ring structure that only handles a growing ring. There is a periodic stabilization procedure that asks what is the predecessor node of the successor.

In the second version, the local storage and related APIs are added to each node. The storage module is first implemented to provide some utility functions. Adding and looking up is quite simple. The node checks if the key is responsible by itself. If it is not, it will send a message to its successor. And a new node should take over part of the responsibility.

In the final, the handling failures process is implemented if one node dies. I make use of the built-in monitor procedure to detect the failures of successors and predecessors.

2 Main problems and solutions

One of the most important problems is to maintain the ring when nodes join and leave. When a node joins, it runs a stabilizing procedure periodically to maintain the ring. If the current node is between the predecessor of the successor and the successor, the node is the new predecessor of the successor. If the current node is not between the predecessor of the successor and the successor, the predecessor of the successor is our new successor.

Another problem was when I did the performance, there was a warning about preventing overlapping partitions. When I add **-connect_all false** flag, the warning disappears.

3 Evaluation

As for the first implementation, I start with three nodes. According to the results, the order in the ring is [1, 15, 20]. This is the expected result.

```
# node 1
erl -name node1@192.168.5.15 -setcookie secret
> Pid = node1:start(1).
> register(node1, Pid).
> Pid ! probe.
# the last one is the current node, the previous one is successor.
# All nodes in the ring: [20,15,1]

# node 2
erl -name node2@192.168.5.15 -setcookie secret
> Pid = node1:start(20, {node1, 'node1@192.168.5.15'}).
> Pid ! probe.
# All nodes in the ring: [15,1,20]

# node3
erl -name node3@192.168.5.15 -setcookie secret
> Pid = node1:start(15, {node1, 'node1@192.168.5.15'}).
> Pid ! probe.
# All nodes in the ring: [1,20,15]
```

As for the second implementation with a local storage, I did some performance tests. According to the result, one machine to handle 4000 elements takes less than four machines that do 1000 elements each. If values are distributed evenly and the number of elements becomes huge, the system with multiple machines performs better than one machine. Because the message passing among multiple machines also costs resources.

```
## one server
./server.sh one

## four servers
./server.sh four
./server.sh stop

## four clients
./client.sh
```

As for the third implementation with handling failure, when the predecessor dies and it is set to nil, someone will sooner or later knock on our

door and present themselves as the possible predecessor. And if the successor dies, we adopt our next node as our successor. Of course, data attached to this node disappears.

There are a lot of improvements that can be done. First, replication is quite crucial to keep data safe when nodes leave. It is also practical to hash names to create unique keys for objects instead of random numbers. Besides, adding virtual nodes is the key to implementing a consistent hash.

4 Conclusions

This assignment was very interesting and practical. Key-value storage is a very important product in the industry. Through this assignment, I learned how to use Erlang to implement a simple Distributed Hash Table and had a deep understanding of Chord scheme. It allowed me to quickly learn other famous open-source projects, such as Redis, which also uses consistent hashing technology.