University of Tromsø

INF-3200 Distributed Systems Fundamentals

Assignment 1

October 4, 2015

1 Introduction

In this assignment a distributed key value store is implemented. The key value store will support storing and retrieving of data. The system implemented will use some of Chord features. Chord is a Peer-to-peer service for lookup and storing of key value pairs. In this report i will discuss message passing, join, finger table, consistent hashing and some features that could be implemented.

1.1 Requirements

The code consist of two parts. A frontend server, and the nodes. The frontend sever is responsible for testing the backend nodes. The frontend will only have access to a list of backend nodes. The frontend will be responsible for forwarding put and get requests to the backend nodes.

Backend nodes are the focus point of this assignment. The backend should support storage of data in memory, and have a monitoring tool.

1.2 Technical background

1.2.1 Chord

Chord is a Peer-to-peer lookup service. Chord will handle the mapping of keys to nodes. Chord needs to only know about one other node to work. Additional routing information can be implemented to support a more efficient lookup service. Chord will have to know its successor routing information. When a key is requested from the system, the node will pass that key to its successor. Complexity is therefore O(N), where N is the number of nodes. In a worst case scenarion, the key will have to go be sent to all nodes.

Chord uses Consistent Hashing to decide which node is responsible for which keys. With consistent hashing, each node and key will get an identifier. Consistent hashing will provide roughly the same balance of keys on each node. Consistent hashing will order the nodes in a circle where each key maps to its successor node in the circle.

Chord also handles node joins, and leaves. A joining node will only have to know about one node currently part of the system. When a node joins the system it will have to find its successor, and update its predecessor. Every key that the new node is now responsible will also have to be moved.

Finger tables are used to improve the performance of the chord protocol. The finger table consists of m fingers that hold routing information of different nodes. The mapping works as follows: $(n + 2^{i-1}) \bmod 2^m$ The n is the identifier of the node, i is the finger entry, and m is the number of finger tables

2 Design

The program is designed after the chord protocol.

2.1 Frontend

The frontend sever will be responsible for communicating with the backend nodes. It is acutally not required to use the frontend. If we already know the name of one node, it is possible to just connect to that node. The fronend will contain a list of all nodes. When the frontend recieves a request it will forward this to a random backend node.

Test code is supplied to test the frontend and the backend for correctness

2.2 Backend

The Backend is responsible for storing and retrieving data. The backend nodes uses five functions to communicate with each other. Get successor, get value, update predecessor, update finger table and put value. Since the approach in this solution is the O(N) approach, each node will send a message to its successor if it decides that it can not answer the request.

2.2.1 Node join

A node is only required to have information about one other node on the network. It uses this node as a hook, and the hook node will send out a request to find the successor of the node joining. When the new node has recieved it successor, it will send and update message to its predecessor that his successor needs to be updated.

During a node join, the node will also initailize its identifier, finger table and send out update messages to the other nodes.

2.2.2 consistent hashing

To decide where each node should be placed in the circle, a consistent hash function is used. In this assignment SHA-1 will be used to hash the routing information to a node. The identifier of the node is then calculated with the function $(hashedvalue)2^m$. 2^m will decide the number of identifier that are used in the ring.

2.2.3 finger table

The finger table will be initialized when a node joins the system. It will send a get successor(finger) for each finger in the table. The first finger in the node will always be the successor of that node. A finger table can for example look like this. Here the node identifier is 27, and it it using 2^5 identifiers:

Table 1: Finger table for node with identifer: 27 and hostname: node 1

finger	hostname	NodeIdentifier
28	node2	29
29	node 2	29
31	node 20	31
3	node 3	1
11	node 1	27

From Table 1, we can also see that a finger can map to itself. In this example the node 27 already knows that keys in the range 11-27 is his responsibility. During a node join, the finger tables of all other nodes needs to be validated. The new node could change the finger mapping from previous nodes such that it is wrong. The newly joined node will then send an update finger table message to all other nodes. The message contains the nodes predeccessors hostname. All nodes that have fingers mapping to the predeccessor will have to update its finger.

2.2.4 Get and put

Get and put messages are recieved from the frontend. These are then hashed, and send ahead to the successor.

3 Implementation

Implemented using python on the uvrocks cluster. The startup script will create a hostfile with randoms nodes. The hostfile is then used to boot random nodes. Fronend will also use this hostfile, when sending out requests.

3.1 Threading

In this program multithreading was not utilized. This caused multiple problems with corner cases since one node can only have one connection active at a time. A good amount of if tests are then used to handle these corner cases, and to make sure a message is not passed to the node that first sent out the message. Threading each connection would help solve these complicated if tests.

4 Discussion

4.1 Node join

During a node join a lot of messages are passed through the network. First a get successor message is passed, which can end up passing N nodes. Then a find fingers function is started which will potentially send out $m * N$ messages. Where m is the number of fingers. Lastly a update finger message is passed. This message will be espcially dominating with few nodes on the network. Since we might have to update every finger for every node. The node join implemented here in this solution, should be improved by utiliziing the fingertable

Figure 1 is an example from a node where we initalized the system with 10 nodes. During an initalizing a lot of messages will be passed around in the system. With only 10 nodes there are 37 messages passing this node. The more nodes the system consists of the more messages will need to be handled when a node joins. The program might run into scale issues when running on hundereds of nodes. This program has only been tested with 40 nodes, which worked fine.

4.2 load redistribution

This implementation does not handle load redistribution after a node joins. Data should be redestributed to the correct node after a node join. If the test script is run before all nodes have joined the network , the result will be wrong.

4.3 Finger table

Fingertables has not been utilized in this implementation. They are implemented, but some of the code will have to be rewritten to utilize them.

Figure 1: Initializing 10 nodes

```
compute-1-2.local - - [04/Oct/2015 13:20:19] "GET_SUCCESSOR 8581429 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:19] "GET_SUCCESSOR 8254843 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 8559424 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064721 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064723 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064727 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064735 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064751 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064783 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064847 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5064975 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5065231 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5065743 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5066767 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5068815 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5072911 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5081103 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5097487 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5130255 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5195791 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5326863 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 5589007 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 6113295 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 7161871 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 9259023 HTTP/1.1" 200 -
compute-3-3.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 34475237 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "UPDATE_PREDECESSOR compute-1-2 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 9309412 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "GET_SUCCESSOR 17698020 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:20] "UPDATE_FINGER_TABLE compute-1-2 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:21] "UPDATE_PREDECESSOR compute-1-2 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:21] "GET_SUCCESSOR 7730755 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:21] "GET_SUCCESSOR 11925059 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:21] "GET_SUCCESSOR 20313667 HTTP/1.1" 200 -
compute-1-2.local - - [04/Oct/2015 13:20:21] "UPDATE_FINGER_TABLE compute-1-2 HTTP/1.1" 200 -
```

Table 2: Size of data storage, in number of characters

node	size
1	3318
2	93
3	804
4	930
5	8954
6	206
7	333
8	4983
9	3923
10	3106

4.4 Evaluation

Tests done with 10 nodes, and 1000 PUT's and GET's.

From Table 2 we can see that for some of the nodes there is roughly the same amount of data stored. There are however big differences when we compare the storage. For eksample between node 5 and 2. These differences happens when the hashing functions gives nodes identifiers with close or far apart neighbours. If a node has successor that is very close, it will be used as storage less than the others.

5 Conclusion

In this report a function distributed hash value store has been described. There are some features and optimization that could be done to improve this solution. Some of these has been discussed in the report. The distributed system has been tested with up to 40 nodes and 1000 PUT and GET operation. The system launches with random nodes each time

References

[1] Chord: A scalable peer-to-peer lookup service for internet applications.
SIGCOMM Comput. Commun. Rev. 31,4. 149- 160. DOI=10.1145/964723.383071