UNIVERSITY OF TROMSØ

INF-3200: Distributed Systems

Christian Reehorst, Dani Beckmann, Nishant Verma

October 5th, 2015

1 Introduction

In this paper, we describe our choice of architecture supporting a distributed hash table (DHT) made to run on a cluster. Making an architecture that reflects our personal flair is important, whilst still staying true to the proven and documented ways of other DHT designs. We have implemented the CHORD architecture where each node is organized as a node in a circular linked list as part of a decentralized structured system that fulfills the requirement outlines of the assignment described in the next subsection.

1.1 Requirements

Given a front-end script to handle PUT and GET requests, we are to provide a back-end solution that stores and retrieves key values to and from random storage nodes. The data (keys) will be distributed between the nodes according to the chosen architecture with a minimum of three nodes. The data is only to be stored in the memory of the dedicated nodes. Lastly we are to implement a monitoring tool that visualizes how the storage distribution is between the nodes. This report will complement and describe the work we have done. Following this chapter are some of the fundamental knowledge needed for this assignment.

2 Technical Background

We used basic hash techniques and socket programming to distribute data on different nodes. A general idea of circular single linked list is required as we keep propagating to the next node. The point of a circular linked list is to skip all of the "if next is not None" logic.

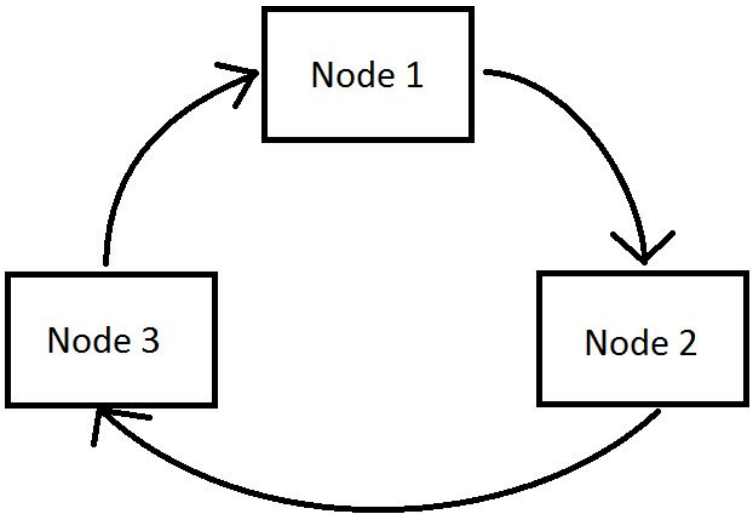
The PUT and GET requests from the front-end uses the HTTP protocol. The module is always implemented on the client side. An instance of the HTTP protocol represents one transaction with a HTTP server. It is instantiated passing a host and an optional port number. Next we will be going over Big Oh notations.

2.1 Big Oh notation

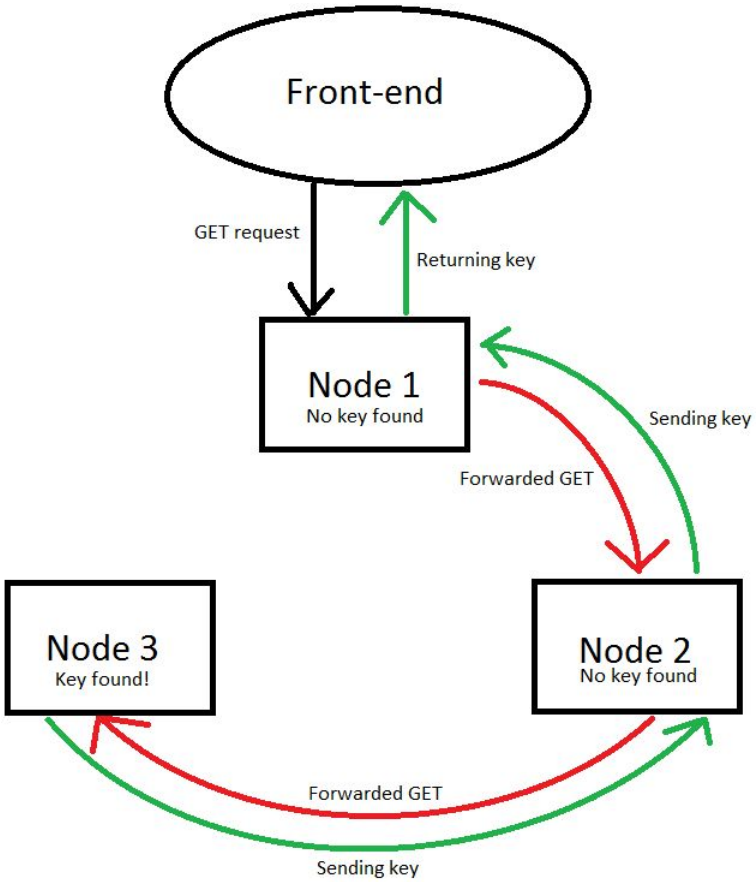
Big Oh notation gives the upper bound on the time complexity of the algorithm. So we can expect the algorithm to not behave worse than this. Our approach for finding a value corresponding to a key may result in looking up all of the nodes in worst case scenario, so we can argue that time complexity our algorithm is O(n), where n is the number of nodes.

3 Design

Our architecture is a decentralized structured network of nodes, using consistent hashing to split the load. Upon a PUT request, the keys are categorized from 0 to 2 evenly, and then distributed to the nodes for storage. We take hash of the key and do a remainder(%) operation to decide the node number on which it is going to be stored. A GET request, from the front-end, asks a random node if it has the desired key and returns it if it has. If it doesn't, it creates a connection to the succeeding node and forwards the request, which returns it if it has it, or creates a connection to the next one, and so on until the key is found. This means that the nodes only know about its succeeding node, but does not have an open connection to it. It only opens one when necessary.



The figure above visualizes how the nodes communicate within our architecture. When a GET request (which is to a random node, as mentioned) is sent by the front-end, and the key is not immediately found, this node then acts as the prime connection between front-end and back-end. This means that whenever the key is eventually found, the key is transported back through the architecture the same way the request came. A connection to the front-end from the node holding the key that was not asked for it, is not made.



This figure depicts how the communication described above maps out. This communication is possible because every node also listens to whoever creates a connection to them. In the next chapter we will talk about our experiences regarding the overall implementation of the pre-code and our solution.

4 Implementation

As the pre-code given this assignment was all in python, converting the entirety of the code to a different language, or doing a fifty-fifty split seemed like more trouble that it was worth, hence all deliverable code is written in python. Debugging through print-statements was troublesome at first because print statements from the node scripts was being cast aside. This was a bit of a starting roadblock for us, but changing the startup.sh file to allow this made aworld of change. Our back end node is designed as client and server. It accepts a TCP connection on a certain port and forwards the request to the neighbour node if needed. The neighbour node is just the next node from the list of nodes except the last node, in which the first node in the list acts as the neighbour node. So we have a circular single linked list where each back end node is the node of the Linked list.

To add a key value pair we created a put request to the random node using hostname. The back-end node then processes the request and decides if it is the correct node for the data to be stored using our hash function. If not, it forwards the request to the neighbour node and so on. Now, same behaviour can be observed when a query is made. We make a GET request to a random node which responds back with the data if it has it, otherwise it gets the

data from the neighbour and respond back. This goes on until we find the data. In the next chapter, we will be discussing the advantages and disadvantages of our design.

5 Discussion

Our chosen architecture, we feel, is well balanced. It is a step above the simplest of architectures, yet still less complex than the more advanced ones. It has a clear logic, and is easy to rebuild or replicate. The realistic approach is much due to this being a school assignment, and doesn't serve any other purpose but for us to learn.

Performance-wise it has some issues. A lot of the time querying for keys is spent communicating (unless you get lucky and the node you contact has the key you are looking for). Also, the architecture doesn't support nodes leaving or joining the network. If one node goes down, for whatever reason, the whole system does too. We've designed the system to support 3 nodes, and for a structured network, that's fine, but for a larger system, the lookup of keys could potentially be quite lengthy. Implementing a finger table would be an obvious solution to this.

Load balancing is an important feature which a simple DHT should support. When inserting a new value it should insert the data in the node in which it has less data. When a new node joins the DHT, most of the insertion should take place in this one. Also we can consider moving some data from nodes with more data to the ones with less data.

Fault tolerance deals with the situation in which one of backend node is dead. In case one of the node stops responding we should have a fallback-mechanism in which we can look for the data in a node which back up the failed node.

Our system does not support both of these features. Also we have a bug in our implementation. If a non -existent key is queried our nodes keep forwarding the request in an infinite loop. A solution to the problem may be that the starting node keeps track of the request and if it encounters this request again it should send an error message. A timeout operation could also be considered, albeit less elegant.

Adding and leaving of node is not supported. To support this, we need to maintain our circular list structure. So, we update our list whenever a node joins or leaves. The next subsection covers the evaluation of our design.

5.1 Evaluation

The test cases run successfully for three nodes and we can say that it will work on n nodes as well.The code scales linearly as we have argued above and thus the time taken to find key in back end system will scale with increase the number of nodes.

6 Conclusion

Given a pre-coded front-end script for sending and receiving key values, we were to design a back-end system to support these functions through a distributed storage over a cluster of nodes. This report has described our chosen architecture for a distributed hash storage system, looking at strengths and weaknesses of it. With consistent hashing, the data is distributed evenly among our storage nodes, providing a well-balanced and solidly structured architecture.

For the group, overcoming programming language barriers in this setting has been our biggest task. None of us has worked on python or shell-scripts before, and finding the ins and outs the given pre-code to later modify and manipulate was a tough challenge. In the end, the system works well given certain parameters, and we are happy with our result.

7 References

[1] https://docs.python.org/2/library/httplib.html Python HTTP protocol

[2] http://cslibrary.stanford.edu/103/LinkedListBasics.pdf Linked lists