Assignment1

Distributed Key-Value Store

Nicolai Bakkeli, nba014@post.uit.no

Introduction

This paper will briefly explain the design and implementation of a distributed key-value store. The key requirement for this work was to alter an existing key-value store architecture in such a way that storage becomes distributed among multiple machines.

Technical background

Distributed data store

A distributed data store is a computer network where information is stored on more than one node, often in a replicated fashion. It is usually specifically used to refer to either a distributed database where users store information on a number of nodes, or a computer network in which users store information on a number of peer network nodes.[1]

Design

The architecture is separated into three parts. These are the client application, the frontend and the backend. The client application serves no other purpose than to issue GET or PUT requests to the value store. Its exchangeable and its other functions are irrelevant but it's there to symbolize the outer connection to the architecture. These applications when trying to connect to the value store, will only be communicating with the frontend.

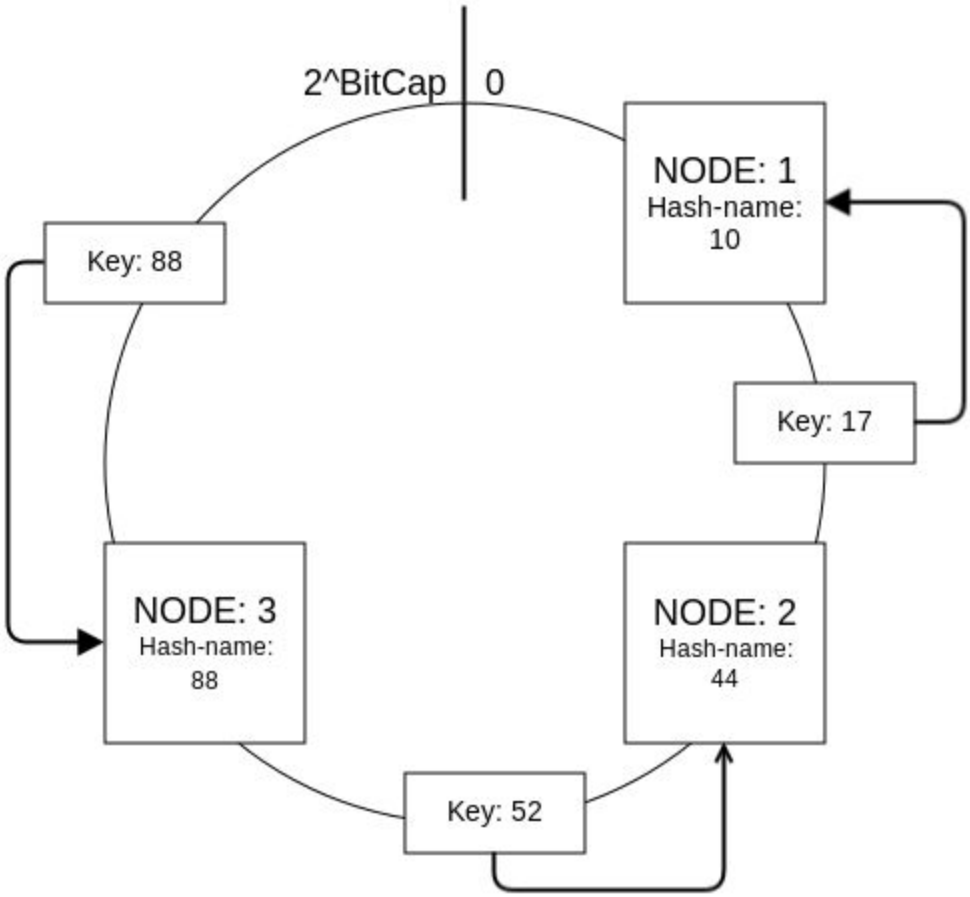
Frontend

The frontend serves as the middleman between the client applications and the backend nodes. As mentioned is the frontend the only node in the architecture that communicates with the client applications. Whenever a client issues a request to the frontend, will the frontend keep the connection with the client until the response is ready. The frontend does not have any data stored locally and must rely on the backend nodes for storing the actual data. So the frontend serves as a middleman between the client and the backend nodes. But the frontend have no real knowledge of the structure between the backend nodes. But it is however aware of the location of every backend node. So when it receives a request will the frontend assign this task to backend node chosen at random. The response received from the backend will be forwarded to the client application when the backend is done with its routine.

Backend

Most of the choices related with the distributed key-value store architecture are related to the structure of the storage and communication between the backend nodes. As a foundation is every backend node running the same identical service, where they have the responsibility of storing the actual data. The data should by design be distributed among the nodes. In order to distribute the data have the nodes schemes to figure out which values to store or not to store. And since the frontend sends requests to random nodes at the backend must each node have a way to retrieve or place the data at other nodes. This makes it necessary for the nodes to be able to issue requests from other nodes and in turn have knowledge of at least one other node each. The implementation pared with this paper gets knowledge of other nodes in the following way. At initiation will every backend node receive every address of every other node in the backend from the frontend. These node-addresses are hashed down, resized and placed in a list of other nodes. Hashing down the addresses of the nodes yields a value that will be used to identify the node as well as assign its responsibility. This list will then be sorted based on the hashed addresses and every node will find its own index in this list. Each node will get one successor node each, and these successor nodes are chosen by choosing the next node from the list that relative to its own index. In this way will let's say the first node in the list have the

second node as its successor and so on. Each node know the hashed address of itself and its successor. These two values are the floor and ceiling of the responsibility span of the nodes. This means that have responsibility of storing every value with a key image that is within this span. Lets look at the image below. By going clockwise around the ring can we see that node 1 comes first. It is followed by node 2 and there after node 3. In this instance will node 1 have node 2 as successor who have node 3 as its successor. The responsibility span of node 1 is based on its hashed address = 10 to its successors hashed address = 44. A value with a key hashed down to 17 is within the span of node 1 and is stored within that node. Node 3 has responsibility for every value with key image larger than itself and smaller than its successor.



Whenever a node gets a request from the frontend will it check if it is responsible for the key. If it is responsible can it do the job itself and store/retrieve the data needed. But if the key is not in the rage of its responsibility must the node forward the request to its successor, who will start the same procedure. This continues until the node with the responsibility for the key gets the request. When it is done will it send a response who will be returned counterclockwise through the chain back to the node that was requested to do the task in the first place. When the data is at the first node will it send it back to the frontend who will forward it back to the client application and the task is considered finished.

Implementation

The code was written in the programming language Python and was a alteration of some existing code. During development was python logging and grep used to identify flaws in the codes. They both output messages to log files while the programs are running. These were useful as the programs were running on other computers, which made error messages harder to get to at the client end.
The frontend code (storage_frontend.py) had a built in test feature that was used to check if the implementation could handle multiple PUT and GET requests.

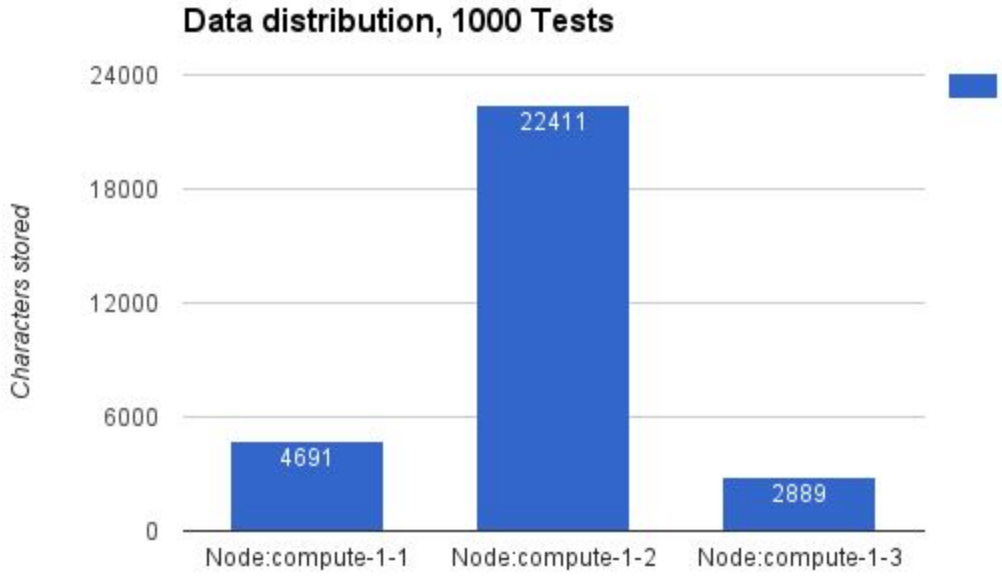
Here are the distribution when running the server test:

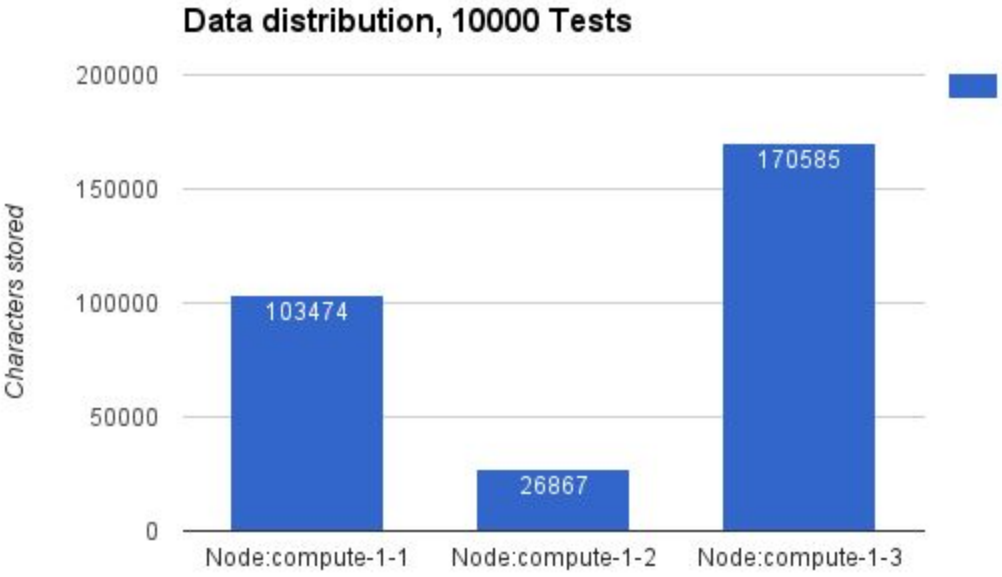
Number of tests:	1000	10000
Node:compute-1-1	4691	103474
Node:compute-1-2	22411	26867
Node:compute-1-3	2889	170585
TotalSize:	29991	300926

Graphs added for this data added on the next page.

The program crashed when running 10000 tests, will be discussed in the discussion.

Issue a GET request with “/size” as key to output the amount of data stored on the backend nodes.


Data distribution, 1000 Tests
24000
18000
12000
6000
0
Characters stored
22411
4691
2889
Node:compute-1-1
Node:compute-1-2
Node:compute-1-3


Data distribution, 10000 Tests
200000
150000
100000
50000
0
Characters stored
170585
103474
26867
Node:compute-1-1
Node:compute-1-2
Node:compute-1-3

Discussion

A problem with this implementation is that new nodes can not be added and old nodes can not be removed. If any node is removed from the backend for any reason will the entire value store stop functioning. But it might work for some time or until the node would have gotten a task at least. In any case is this a problem. This could be changed by adding a procedure for the nodes that checked whether or not the successor nodes are present. And if they are not present then the successor is updated to be the next node in the list.

A far from optimal design choice here is to give every node the address of all the other nodes without using the information properly. Currently do each node only pass its task to its successor but by having a better scheme that utilises the other addresses can the lookup be reduced. Optimally in this scenario(where every node know of every node) could the request be sent directly from the first backend node to the responsible node. It's not even hard to implement since the list is there sorted and all. The current lookups cost is O(N). Having the current implementation makes it hard to scale as well.

The implementation distributes data poorly. It seems to be that way at least. One reason for this might be that the key span assigned to each node is based on their addresses. The problem with this is that the output values from the hashing might end up being close, thus preventing the distribution from being even. On both tests ended the last node in the ring with the highest amount of data stored.

The test crashed on 10000 tests, this might be because the bitsize of the keys are too small.

I was originally planning on using the chord protocol. It can with proper use have a lookup of O(log N) where N is the number of nodes in the network. It scales better and can be dynamic in regards to adding or removing nodes. But i didn't have enough time to implement it so i had to choose a easier solution.

Conclusion

Data is being distributed among backend nodes that can retrieve data from the correct locations. The frontend keeps no key-value data in memory and relies entirely on the backend. The design are far from optimal but it works to some degree.

References

[1].”Distributed data store”, wikipedia; Date retrieved: 4 October 2015 15:25 UTC: https://en.wikipedia.org/w/index.php?title=Distributed_data_store&oldid=679086517