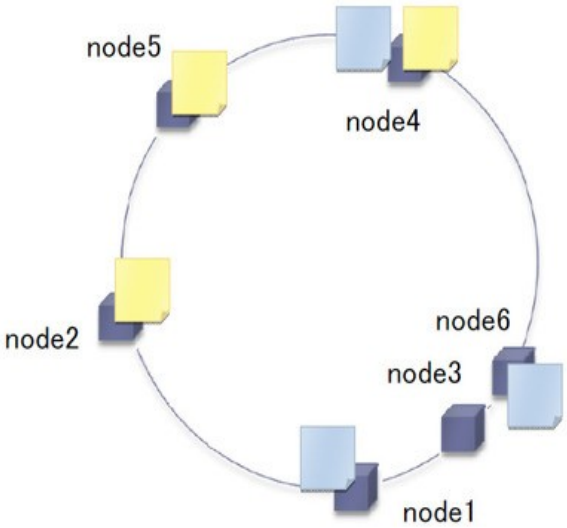
# INF-3200 Assignment 1

## Introduction

In this assignment the author were asked to design and implement a distributed key-value storage system. Were the are mainly two requests which stores(PUT) and retrieve(GET) values stored on nodes that are used. There will be used backend nodes and frontend nodes, where the frontend node will distribute the put og get to the backend nodes which further handles the request. It must handle at least three backend nodes and the data must be stored in-memory.

## Design

The author started by choosing to implement a Chord implementation[1] which uses consistent hashing and thereby distribute the values by the keys. Each backend node knows only of the next node, and the key as hashed and the node address are hashed also. Then the hashed key will be compared against the hashed node address to figure which place the value must be stored. Here the span of value between nodes determine where the value will be stored depending on choice of implementation if the node holds the lower or upper span. The figure to the right shows an abstract idea of how the design is.




The author chose to implement it where the node stores the values which are in the upper span. Here the fact that the lowest node has some values which will be excluded by this approach, so the last node will here handle the values which are lower than the first node. Also by hashing the node address the nodes must be rearranged, since 1-1, 1-2 etc. will not translate directly to the same if they are hashed. By looking on the figure above you may see that the nodes are not arranged by name, if the hashed value of a node are greater it must be placed later in the circle of nodes. So the nodes will only know of next node, which are of greater value except the last one which knows of the first one.

The first thing the implementation does is to send out to the backend nodes the list of nodes so it may figure out which node are the next node in line. Then it stores which node are next, before handling any requests, here it stores all of info needed. Both the node name, hashed value of node and next node. And when it knows which node are the next it knows where to send the put request if it does not belong at the node it stands.

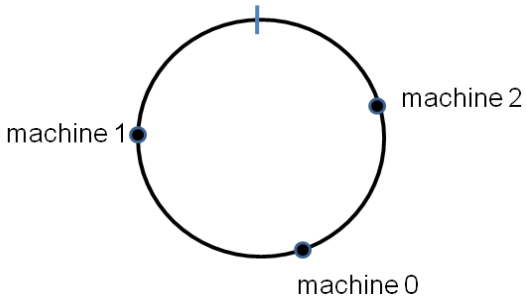
The get request will work in the same way, where it looks at the span above the node for the key, if it does not find it will go to the next node.

There will be hashing with the sha1 algorithm, since it seems that sha1 is the one that works best under the circumstances for the Chord solution.

1 http://dl.acm.org/citation.cfm?doid=964723.383071

## Discussion

The author had some issues regarding the assignment where time were most of the problem, had to redesign the solution after some bugs that the author did not had time to fix. Have checked the log which shows after running the tests that the distribution of key-value are not perfectly, where one node receives 50 of the key-values and one just received 11.




During the work the author had problems with discovering hidden problems because the code runs on the cluster and error and warnings does not appear in the terminal. This caused many hours of fault handling on small issues which could have been used on implementation instead. The author first used the logging library of python which were useful, but the best help where from other students which showed the author how to write the error messages to a file for further investigation. There were also a lot of issues when trying to sort and distribute which node were supposed to be in which order, where the author did not discover that after hashing the node address that the order were changed. So when the next node for machine-1 were supposed to be machine-2 it were machine-0, so that caused some issues which took some time to wrap the head around. So by adding in the key in the wrong order it did not find the key again when searching for it in get. And also since there are some if statements that must be furfilled the put requests ended up in a everlasting circle where it went from one node to another and back. Thought this were because of how the list where sorted at first, that the lowest value were last and highest were first.

After this were discovered the author had no problems with how the values were distributed among the nodes. During the implementation of get, the author chose to just let the frontend choose a random node and set a request. From here the node checks the map for a result, if not found go to the next node. If found the node returns the value to the frontend. This will result in a case where it could potential go through every node before finding the correct value. There are ways to work around this by potential adding a previous node where it could check the hash again and decide if the next or previous node are the best way to go. This could also result in a case if the last node is chosen among the random nodes and it has to traverse through everyone for the first node.

After some issues which occurred at the end, the author also had some issues with the monitoring system. So that solution did not work at the last run before hand in. But there are not enough time to fix this issue.

Also this solution does not handle adding of more nodes during the run. It could have more then 3 nodes when it starts though.

## Conclusion

The author has designed and implemented a solution that works, but not perfectly. There are many thing that could be improved, for example by adding a previous node so that get gets a lower look up time. There are also the issue where it does not distribute the values even among the nodes, but this is something that be because of the hashing algorithm and will be better with more data and more nodes. But the author is pleased with the solution.