The Arctic University of Norway

Report Inf-3200
Distributed Systems Fundamentals

Distributed Hash Tables

05.10.2015

Tim A. Teige

Fredrik Høiseter Rasch

E-mail: tte008@post.uit.no, fredrik.h.rasch@uit.no

Introduction

This report describes the implementation and design of a distributed hash table system.

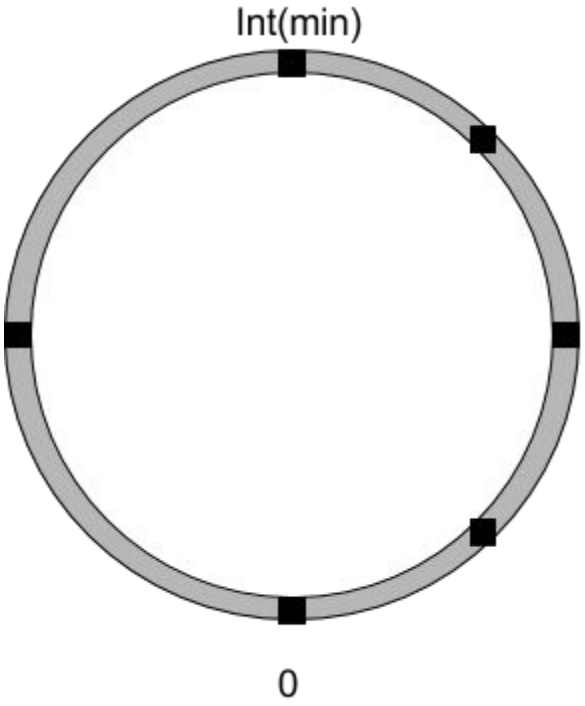
Technical Background

Distributed Hash Table

A distributed hash table is a decentralized storage system. This system provides a lookup service on keys that maps to values. The key-value pairs are stored on nodes within the system. The distributed hash table is transparent to the user, since it should not matter which node is queried.

Design

The system is designed with a frontend server handling user input, and backend servers storing the key-value pairs. The backend servers are ordered in a ring. The ring index is based on the number of nodes in the ring. The node indices are separated evenly, spanning from minimum to maximum the value of a 32-bit integer, in order to reduce the load on each node.

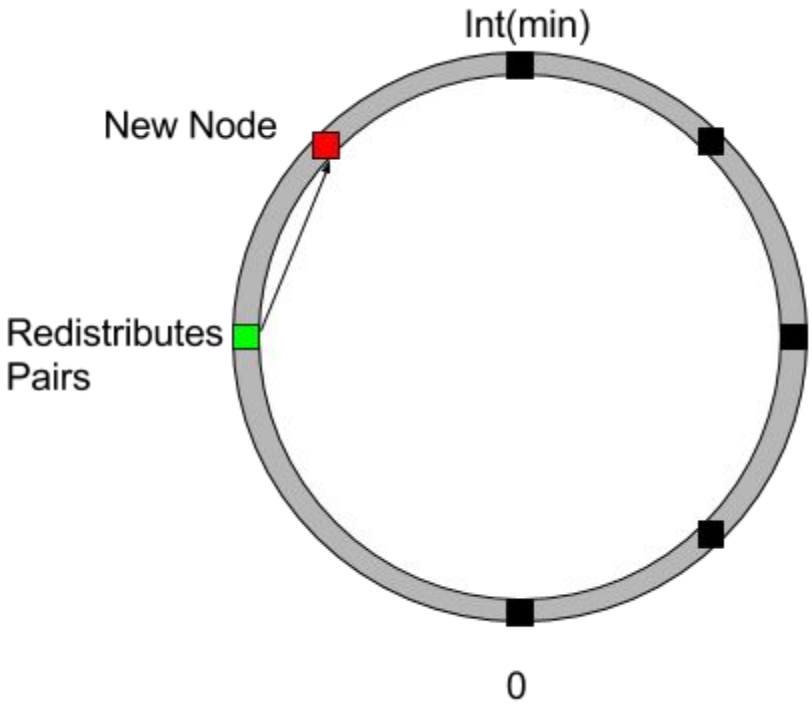


Each node is assigned the id when it logs on to the frontend server. This is id will be half the greatest distance between nodes that are next to each other in the ring.

Storing and retrieving

Storing and retrieving key-value pairs is done over an http connection using the http get and put methods. When the frontend server is prompted to retrieve or store a value it resolves the target backend server based on the hash value and the backend ring locations. The node that has/will receive the pair is the first node with a higher value than the hash value. The http methods are handled asynchronously and stored in a thread safe dictionary.

Logon, logoff and Redistribution



When a new node connects to the frontend server, a redistribute command is sent out to the node who has a greater ring id than the newly connected node. This ensures that both nodes have the correct values. These key-pair values are sent to the new node using put for each pair.

When a backendserver logs of it starts to flush the hash table by sending the values to the frontend server for reassignment. When a node logs off, it is removed from the ring in order to avoid any further incoming requests.

Implementation

The distributed hash table system has been implemented using c# and compiled with mono. A development technique used during the entire implementation was pair programming. Visualization has been done with WinForms.

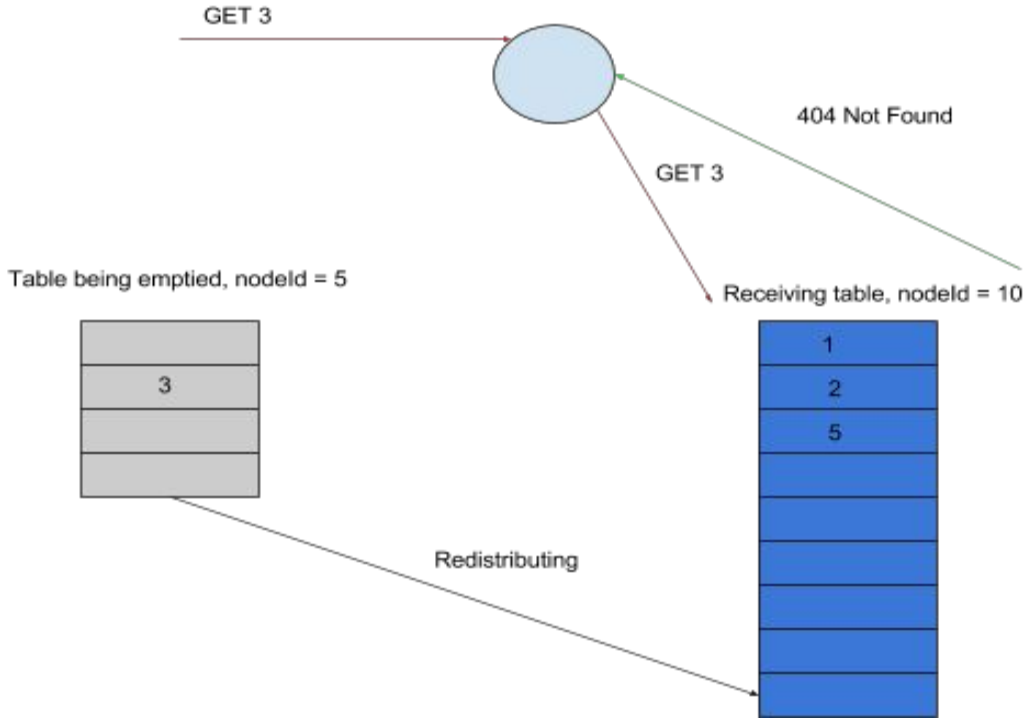
Discussion

The current design uses a single frontend server. This is not the most fault tolerant solution. If the frontend node crashes, the distributed hash table will be inaccessible. This could be solved by using the backend nodes as possible frontends in case the frontend crashes. The backend node which assumes the role of the frontend would have to redistribute its key-value pairs and

launch the frontend server. Another solution would be to have multiple frontend servers which would take over when one crashes. This in turn would increase the costs of running the system.

The entire system has no fault tolerance. If a node crashes, the data is lost. This could be solved by using replication of data or by introducing redundancy in the system. Replicating data would work on a small scale. When the amount of data increases, the number of nodes would also have to increase, thus increasing costs. A scheme for replicating data could be that each node also stores the data the next node stores. If node n crashes, node n-1 would still have this data. Only when node n and n-1 or n and n+1 crashes would there be data loss.

There is a possibility for false output in the current implementation. If a node is logging off and has the key-value pair that is searched for, the search would be redirected to a node that currently might not yet have received the pair from the node logging off.



A solution to this problem would be to wait for the node to finish exiting before allowing queries for keys which that node might contain, ensuring that if the key exist it will be resolved correctly.

The ring only supports up to 256 nodes, this is because only a single byte is used to store the ids. Using an integer would support up to 2^32 nodes, but there is no need to support that many nodes.

Conclusion

The design and implementation works. There is room for improvement, especially fault tolerance. There are optimizations possible, such as more efficient logoff handling.