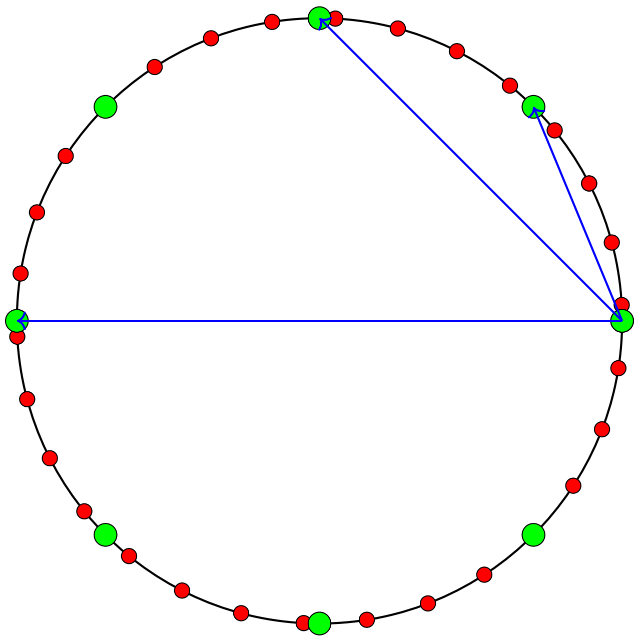
LINFO2345 - Projet
Chord DHT

2024

Author:
Matthieu Pigaglio



Chord DHT Ring

1 Introduction

The size and complexity of modern large-scale distributed systems, such as IPFS, create the need for reliable data storage and retrieval mechanisms more complex than just replicated databases. Distributed Hash Tables (DHTs) emerged as a good solution to solve this need.
In a DHT, each participating node is responsible for a subset of the data, and a consistent hashing mechanism ensures that each key-value pair is assigned to a specific node. This decentralized approach enables DHTs to handle node additions and failures dynamically, making them robust against single points of failure. The structured nature of DHTs allows them to efficiently locate data with low latency.

This project will let you explore the Chord DHTs. You will learn to create a DHT structure and how to retrieve data stored in the DHT.

2 Project Summary

Your contribution to this project consists of 3 parts:

- Implement in Erlang a distributed hash table structure.
- Implement protocol to query data in the DHT.
- Allow new nodes to join.

3 Part I: Distributed Hash Table structure

As explained previously, you will implement a distributed hash table in Erlang. You will implement an instance of the Chord DHT [3].

3.1 Chord

Chord [3] is one of the first distributed hash table protocols, along with Pastry [2]. It was introduced by I.Stoica, R.Morris, D.Karger, F.Kaashoek, in 2001.

In Chord, you have two types of items:

- Nodes, which are the users of the DHT and which store data in a distributed way.
- Keys, which are the identifiers of the data in the DHT.

Nodes and keys are assigned an m-bit identifier using a hashing function. In Chord, the hashing function used is SHA-1 [1]. Using the identifier space for nodes and keys helps to robustness and performance of Chord. The identifier space is a ring of 2^m identifiers from 0 to $2^m - 1$. m should be large enough to avoid collision. Each node stores the identifier of the successor node in the circle in a clockwise direction and the predecessor in a counter-clockwise direction.
The concept of a successor can be used for keys as well. The successor node of a key k is the first node whose ID equals k or follows k in the identifier circle. Every key is assigned to (stored at) its successor node, so looking up a key k is to query the successor of k.

3.2 Your task

Your first task will be to set up a Chord dht in Erlang.

You will use m equal to 16. Which will give you a 16-bit identifier space, it allows you to store up to 65536 keys and allows you to run the DHT with 10 to a hundred nodes. A list of keys (before being hashed), stored in a CSV, is given on Moodle to test your DHT. Each node will be identified by a number from 0 to the number of nodes in the system. A node stores the identifier of its successor and predecessor and the identifier of all the keys stored in it.
To log the system you will create a folder named *dht_N* where N is the number of nodes in the DHT. In this folder, each node will create a CSV file named *node_number.csv*. This CSV file will contain one line as follows:

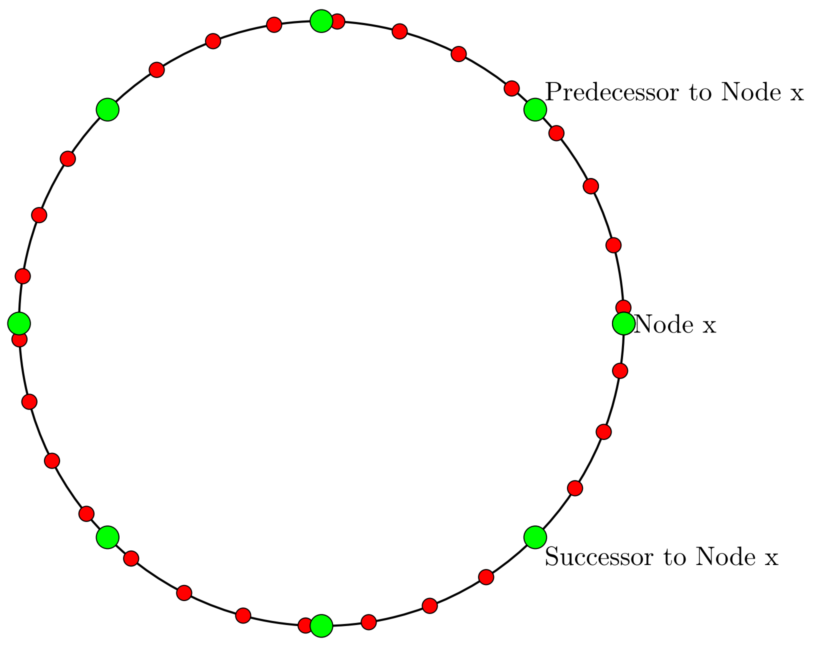


Figure 1: Chord DHT Ring (Nodes in green and keys in red)

node_identifier,successor_identifier,predecessor_identifier|key1_identifier|key2_identifier|key3_identifier..

All the identifier need to be a string representing the hex identifier. For exemple: "a3e324f01ab359d2"
You need to strictly follow this as the project correction is partially automated.

Hints

- *No need to implement SHA-1, Erlang already can do SHA-1.*
- *Use a control node to start the experiment and don't start each node manually.*
- *Try with systems of only 10 nodes before trying with a hundred nodes.*

4 Part II: Query a key in the hash space

You now have a Chord DHT structure. You now need a way to efficiently locate the node having specific data. A naive way is to pass the query to the successor until you find the right node. This will lead to a $O(N)$ query time where N is the number of nodes in the ring.

4.1 Finger Table

To avoid linear search, Chord implements a faster search method. Each node keeps a $fingertable$ containing up to m entries (m the number of bits in the hash space). The first entry of this $fingertable$ is the node's immediate successor. The i^{th} entry of the $fingertable$ will contain $successor(n + 2^{i-1} \mod 2^m)$. When a node wants to query a key. It will query the closest preceding node in its $fingertable$ to the key. The queried node will check if it is the node containing the key. If this is the case, it will send its identifier back. If not, it will query its $fingertable$ and send to the queue the next closest preceding node to contact until the node containing the key is located.

Example of Updating the Finger Table

Suppose we have a ring with 16 nodes (IDs 0 to 15) and a node n with ID 3 wants to find the node responsible for a key with ID 12.

- Step 1: Node 3 checks its finger table to find the entry closest to 12 without exceeding it. Let's say the finger table points to nodes at IDs 4, 5, 7, and 11.

- Step 2: Node 3 sees that the closest preceding entry to 12 is node 11. So it forwards the request to node 11.
- Step 3: Node 11 checks its finger table and finds that its immediate successor is node 12, which is responsible for the key.

Result: The lookup is completed in two hops (from 3→11→12) instead of traversing every node around the ring.

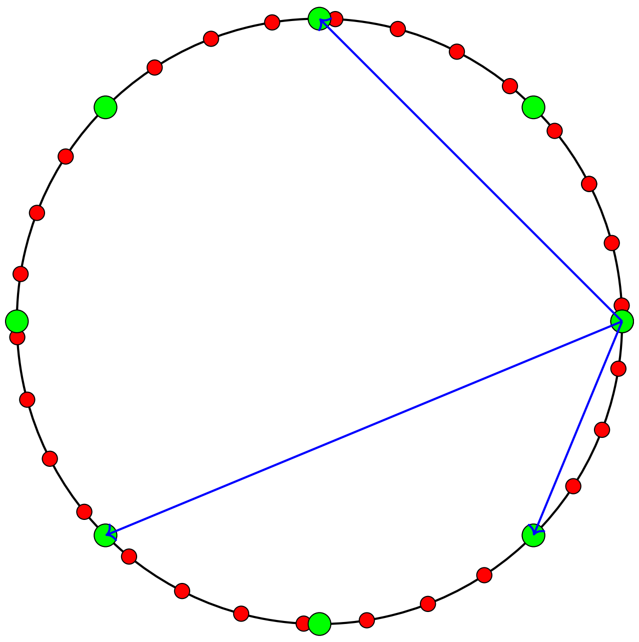


Figure 2: Chord DHT Ring with finger table

4.2 Your task

Your task is to implement a finger table and the process to retrieve the closest preceding node of a key.
You have on Moodle a CSV with a list of keys to retrieve to help you test your implementation. The node that runs queries will create a file named *node_number_querries.csv*. This CSV file will be constructed as follow:

key_identifier,contacted_node_identifier1|contacted_node_identifier2|contacted_node_identifier3...

All the identifier need to be a string representing the hex identifier. For exemple: "a3e324f01ab359d2"
You need to strictly follow this as the project correction is partially automated.

5 Part III: Adding a new node

Now, you have a working CHord DHT, but it is static. No nodes can enter the DHT after the start. In this last part, you will allow new nodes to join the DHT.
You need to update three invariants:

- successor of predecessor and successor of the new node.
- finger table of all nodes.
- node n needs to retrieve keys stored in his successor.

You will follow this process:

- Initialize new node n (the predecessor and the finger table).
- Notify other nodes to update their predecessors and finger tables.

- The new node takes over its responsible keys from its successor.

You will implement a function *add_node(n)* to easily add a new node to the system and update files of nodes of the system.

6 Grading

The project can be done by a group of two or alone.
You will find the weight of each part below:

- Part I: Distributed Hash Table structure: 40%
- Part II: Query a key in the hash space: 40%
- Part III: Adding a new node: 20%

The project will count for 5 points on your final grade for the course.
You will provide a zip file containing 3 directories, one for each part of the project, and a report in PDF format. In each part directory, you will provide your Erlang code and a readme on how to read and launch your code. Code with no readme explaining how to launch your code will not be reviewed. The report will contain an explanation of your implementation. You need to complete all the previous parts of the project before beginning a new one. You have until Friday 29/11 at 6 pm to submit your project on Moodle.

Good luck and feel free to send me an email, **matthieu.pigaglio@uclouvain.be**, if you have any questions.

References

[1] E. Biham and R. Chen, “Near-collisions of SHA-0,” Cryptology ePrint Archive, Paper 2004/146, 2004. [Online]. Available: https://eprint.iacr.org/2004/146

[2] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” *ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.