UCLouvain
École polytechnique de Louvain

LINFO2345 – Languages and algorithms for distributed applications

Project 2024-2025 : Chord DHT

Authors :
GUERRERO Anthony - 1905-2000
KHEIRALLAH Cédric - 0462-1900

Teacher :
VAN ROY Peter

Teaching Assistant :
PIGAGLIO Matthieu

2024 - 2025

1 Part 1 : Distributed Hash Table structure

In this project, we were tasked to create a Chord Distributed Hash Table (DHT) in Erlang. A DHT is made up of nodes that collectively store keys between them by comparing its value to their hashed values allowing for efficient storage (similar to a Hash Map structure).

To accomplish this we divided our code into multiple parts :

- Main module ‘`main.erl`‘
- Node module ‘`node.erl`‘ and ‘`node_utilities.erl`‘
- CSV creation module ‘`csv.erl`‘

1.1 Creating the structure

We start by creating N nodes whose values are hashed and mapped to get a consistent ring of nodes going from 0 to $2^m - 1$ possible slots available to store keys (we had to use m = 16 bits which means we had $2^{16} = 65535$ unique slots available to store keys on the ring). Nodes are spawned and interconnected very similarly to a Circular Linked List where each node remembers its immediate predecessor and successor neighbours on the ring.

Once the (hashed) nodes are set in a ring we can assign them the keys given to us in ‘`keys.csv`‘. Keys are assigned to the first node with an ID greater than or equal to the key’s hash using message passing (wrapping around if necessary).
Each node is instantiated through the ‘`spawn`‘ process and a receive loop listens for incoming messages such as for setting the predecessor or successor or adding a key to its key list.
Finally before inserting them into their internal storage (which are simply CSV files assigned to each node named ‘`<n>.csv`‘ where n is the node number) we convert the hashed node values and keys into hexadecimal to encrypt them. All the files are then saved into a directory ‘`dht_<N>`‘ where N is the number of nodes in the DHT.

2 Part 2 : Query a key in the hash space

We now introduce finger tables to the original DHT. We start by setting up the ring and set up a finger table for each of them. We then read the key queries from ‘`key_queries.csv`‘ and send lookup requests to nodes.
For the lookup each node checks if it owns the key and if not reroutes the request to the nearest hop node using its fingertable.
Finally the path taken for each query is saved into a CSV file.

The finger tables of every node n are of size m (here 16) and each entry correspond to the nodes responsible for specific keys we can get from the given formula :

$$Entry \quad k = (n + 2^{k-1}) \ \% \ 2^m$$

where k is the entry in the finger table. The finger table stores informations about the first node whose ID is equals or follows $(n + 2^{k-1}) \ \% \ 65536$.
To generate those tables we first calculate "finger values" using the formula we’ve just seen and then iterate over all nodes in the ring to find the node responsible for each finger value. When

a node with an ID $\geq$ (taking a possible overflow around the ring into account) the finger value becomes the entry in the finger table (otherwise we loop around to the first node in the ring).

In the lookup process we try to determine which node is responsible for the given Key in the ID space by checking for each key where does it fall in the circle (with the help of the previously generated finger tables). We advance in the ring and compare the value of the Key with the Current node ID and the Predecessor node ID. With this method of simply comparing values we also have to take into account that values would in a way overflow when having looped around the entire ring.
Once the node that should store the key is found it is sent to the node that made the initial request with the path taken using message passing to find it and is saved into a CSV.

3 Part 3 : Adding a New Node

We now introduce the concept of being able to add new nodes to the DHT table. In order to add a node there are multiple steps that needs to be followed : instantiating the node, adding it to the node ring, updating predecessors and successors, transferring the keys and finally updating their finger tables.

Firstly we instantiate the node using the same hash method as before to create its identifier then we identify where in the hash ring it needs to be inserted, using that information we can insert it in the ring and update the predecessor and successors of adjacent nodes.
The successor of the new node will go through its keys and identify which ones needs to be transferred to the new node, those keys are sent to the new nodes using message passing and then the CSV's are updated. While this happens, new finger tables are being generated for each nodes and sent to them to update them and take the new node into consideration.

4 Validation and Verification

To ensure the correctness and robustness of our implementation, we conducted validation by comparing our results with those of other groups working on the project.

We ran our DHT with the same parameters and input files as the other groups, including node identifiers, keys and key queries. After running the DHT with the same inputs, we shared the resulting CSV files, which contain the lookup paths and key ownership distributions.
Upon comparison, we found that our results were identical to those of other groups when using the same inputs. This consistency gives us greater confidence in the correctness of our implementation.