LINFO2345: Chord DHT Project

Group Members:
Corentin Detry
Wing Tao Chan

Course:
LINFO2345
Languages and algorithms for distributed applications

Professor:
Peter Van Roy

December 05, 2024

Contents

Introduction

In this project for the LINFO2345 course, we used Erlang to implement a *Chord Distributed Hash Table* (*DHT*). In distributed systems, Distributed Hash Tables provide a fast lookup feature similar to that of a hash table but distributed across multiple nodes. This allows for a scalable and fault-tolerant system.

The first phase of the project involves creating a ***Distributed Hash Table*** structure with a 16-bit key space, following the implementation of the *Chord* algorithm. Each node stores a list of (hashed) keys, as well as its successor and predecessor.

In the second part, we were tasked with adding query functionality to our *Chord DHT* implementation. A finger table was created for each node, storing 16 entries (the same size as our key space), which allows for faster queries than the naive approach of querying each node in the network sequentially.

In the third part of the project, our goal is to add new nodes to the Chord DHT. This part of the project has not been done due to time constraints.

Part 1 Explanation

Hashing Method

Per the project statement, we use the SHA-1 hashing algorithm to hash the keys and node identifiers, all in a 16-bit key space. We use the `crypto` Erlang module to hash the keys and node identifiers.

Our hashing function is the following:

```erl
% Calculate the hash of a key/identifier
calculate_hash(Key) ->
    % Convert the integer key to a string
    StringKey = integer_to_list(Key),
    % Hash the string key using the SHA-1 algorithm
    Id = crypto:bytes_to_integer(crypto:hash(sha, StringKey)),
    % Return the hash modulo 2^16
    Id rem ?Max_Key.

% Convert a hash to a hexadecimal string
hash_to_string(Hash) ->
    string:to_lower(integer_to_list(Hash, 16)).
```

Listing 1: Hashing Functions

Implementation

Our Chord DHT implementation in part 1 is composed of two main modules: `dht` and `control`. The `dht` module contains the implementation of the Chord DHT, while the `control` module is used to control the DHT, read the keys from the input file, and write the results to files.

Control Module

The `control` module contains the following main functions:

- `start/0`: This function initializes the control node, which is responsible for reading the input file and starting the DHT.
- `init/1`: This function initializes the DHT by creating the nodes and adding keys to the DHT, as well as querying each node for their current state to write to the output file.
- There are other less important functions.

DHT Module

The `dht` module contains the following main functions:

- `start/2`: This function initializes the DHT, it takes in 2 arguments: `N` the number of nodes to create and `Keys` a list of un-hashed keys to add to the DHT.
  - It starts by creating a list of `N` nodes, each with a unique identifier (sequential integers from 1 to `N`).
  - It then hashes the node identifiers (it creates a tuple with the ID and Hashed ID) and sorts them based on the hashed ID.
  - We then also hash and sort the keys.

  - Based on those sorted lists, we can then assign the list of keys that each node will be responsible for.
    - To do so, we call the `set_keys_for_nodes/2` function, which returns a list of nodes with their respective keys.
  - After we have the keys that each node will be responsible for, we spawn each node and assign them their keys, hashed ID, and normal ID.
    - Nodes start with the predecessor and successor set to themselves.
  - Once the nodes are started, we collect all of their Pids in a list, and we then call `finish_initilization/2` to set the successor and predecessor of each node.
- `set_keys_for_nodes/2`: This function takes in a list of Hashed Node IDs (sorted) and a list of Hashed Keys (also sorted) and returns a list of nodes with their respective keys.
  - To do so, it iterates over the list of node IDs, and for each node, it finds the keys that are between the current node and the next node in the list.
  - It then creates a tuple with the node ID, node Index, and the list of keys that the node is responsible for.
- `finish_initilization/2`: This function takes in a list nodes with Hashed IDs and their Pids, as well as a list of Hashed Node IDs. It sets the successor and predecessor of each node.
  - It iterates over the list of nodes, and for each node, it finds the next and previous node in the list of nodes and sets them as the successor and predecessor of the current node.
  - It does so by sending a message to the node's PID to set the successor and predecessor.
  - The successor is the next node in the sorted hashed node IDs list, and the predecessor is the previous node in the list.
- `loop/5`: This function is the main loop of the node process. It takes in the Node Index, Identifier, Successor, Predecessor, and Keys.
  - It listens for messages to add keys, get information, set the successor, set the predecessor, or stop.
  - When receiving a message to add a key, it appends the key to the list of keys.
  - When receiving a message to get information, it sends the node's information to the caller.
  - When receiving a message to set the successor or predecessor, it updates the successor or predecessor accordingly.
  - When receiving a stop message, it stops the loop.
- `calculate_hash/1`: This function calculates the hash of a key or identifier.
- `hash_to_string/1`: This function converts a hash to a hexadecimal string.
  - These two functions are shown in Listing 1

Output

The output of part 1 is a directory with the name `dht_NumberOfNodes`, where `NumberOfNodes` is the number of nodes in the DHT. This directory will contain one csv file per node (with the name `node_NodeIndex.csv`), each containing one line following this syntax:

```
1 node_identifier,successor_identifier,predecessor_identifier|key1_identifier|
  key2_identifier|...
```

The commands used to run the code are explained in the README.md file.

Part 2 Explanation

Hashing Functions

The hashing methods for part 2 were the same as in part 1.

Implementation

Just like part 1, we also have two modules in part 2: `dht` and `control`.

Control Module

The `control` module contains the following main functions:

- `start/0`: This function initializes the control node, which is responsible for reading the input file containing the keys, and the input file containing the key queries, as well as starting the DHT and eventually querying a node for the keys.
  - To do so, it calls the `dht:start/2` function with the number of nodes and the list of keys to add to the DHT.
  - It then asks a starting node (defined by its ID which is in the `control.erl` file) for each Key in the `key_queries.csv` file. When the query of a key is done, we get a list of nodes that were queried to get to the node that has the key.
  - Finally, it writes the results of the queried nodes to a csv file whose contents will be explained in the output section.

DHT Module

The `dht` module contains the following main functions:

- `start/2`: This function initializes the DHT with the given number of nodes and a list of keys.
  - It creates a list of nodes with the given number of nodes. It then Hashes the node IDs and sorts them (in a tuple to keep the node Index and its Hashed ID).
  - It then does the same for the keys, it hashes them and sorts them.
  - Same as in part 1, it then assigns the keys to the nodes using the `set_keys_for_nodes/3` function.
  - Once that's done, it spawns the nodes, and collects their PID, which is stored in a tuple along with the node Index, and Hashed ID.
  - After the nodes were spawned, it calls `finish_initialization/2` to assign the successor and predecessor of each node, as well as creating its finger table.

`finish_initialization/2`

Let's look more at the `finish_initialization/2` function. This function takes in a list of nodes with their PID and a list of sorted hashed node IDs. It then goes through each node and assigns the successor and predecessor by looking at their index in the sorted list of hashed node IDs. It then creates the finger table for each node by calling the `create_finger_table/3` function.

`create_finger_table/3`

The `create_finger_table/3` function creates a finger table for a node by iterating through each bit position (since $m = 16$ is the key space size, and the finger table has m entries) and calculating the target position for each finger. It then assigns the responsible node for that finger to the finger table.

To figure out the node responsible for each finger position, we calculate the target position for each finger. Each finger jumps $2^{\{(i-1)\}}$ positions ahead. We then calculate the target position by adding the jump distance to the current node index and taking the modulo of the number of nodes. We then get the responsible node for that finger position by looking at the target position in the list of nodes with their PID.

Key query

When the controller goes through the list of keys to query, it calls `dht:query_key/3` with the list of nodes with their PID, the key to query, and the index of the starting node.

The function then hashes the key and sends a message to the starting node to query the key. The starting node then forwards the query to the appropriate node, and the queried nodes are stored in a list. When the key is found, the queried nodes are returned to the controller.

Each query message piggybacks the list of queried node identifiers, this then makes it easy to know which nodes the query went through to find the key since each one appends its identifier to the list.

When a node receives the message `{query_key, Key, Caller, QueriedIdentifiers}`, it then checks if it responsible for the key, if it is it just sends a `{key_found, NodeIdentifier, self(), NewQueriedIdentifiers}` message to the caller (with its own ID appended to the list).

If it's not responsible, it looks for the closest preceding node in its finger table and forwards the query to that node. To achieve this, we used the same technique explained in the oringinal Chord paper, where we go backwards in the finger table to find the closest preceding node.

Output

The control module will, once the queries are done, write the list of queried identifiers for each key in a file called `node_StartingNodeIndex_queries.csv` in the src/part_2 directory (with `StartingNodeIndex` being the value of that variable).

The csv file will contain one line for each key queried, with the following format:

```
1 key_identifier,contacted_node_identifier1|
  contacted_node_identifier2|contacted_node_identifier3...
```

Part 3

As we mentioned previously, we did not have time to implement the third part of the project. This was mainly due to encountering several roadblocks while implementing part 2, which required us to rework the entire code. These changes were also back-ported to part 1 to correct it. This process took considerable time, and in the hopes of ensuring that the first two parts functioned well, we chose to focus on them and planned to do part 3 if time allowed, which it did not.

Sources

Here are the sources that we used to implement our Chord DHT.

- Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Retrieved from https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
- Chord (peer-to-peer). Retrieved from https://en.wikipedia.org/wiki/Chord_(peer-to-peer)
- Distributed Hash Tables: In a nutshell - Recessive. Retrieved from https://www.youtube.com/watch?v=1wTucsUm64s&t=4s
- ChatGPT ErlangSparq to get going with Erlang.
- Chord: Building a DHT (Distributed Hash Table) in Golang - Farhan Ali Khan. Retrieved from https://medium.com/@jingyang_56841/key-lookup-in-chord-with-finger-table-c0179bafae13
- Key lookup in Chord with finger table - Jing Yang. Retrieved from https://medium.com/techlog/chord-building-a-dht-distributed-hash-table-in-golang-67c3ce17417b