Large-Scale Distributed Systems
Project 2: Distributed Hash Tables

Laurent Hayez

November 21, 2015

Table of contents

1 Introduction

2 The base Chord protocol

In this section, we will implement the base Chord protocol. At first, we will consider a simplified version of Chord with no finger table. Each peer will only have links to its successor and its predecessor, thus we expect the search time to be $O(n)$ hops, where n is the number of peers in the ring. Secondly, we will add a finger table to each node so that the queries can be performed in $O(\log(n))$ hops. Finally we will compare both protocols' search efficiency for 500 queries per node.

2.1 Implementation without finger table

As you can see in the file `Chord-v1-2.lua`, the only tricky function (at first sight at least) is `is_between`. It works as follow (example for the bracket pair "()"): as we have a ring topology, we must consider two cases, lower $<$ upper and upper $<$ lower. In the first case, nb $\in$ (lower, upper) if nb $>$ lower and nb $<$ upper. In the second case, we have that lower is the last element of the ring, and upper the first one (this can happen because we work modulo 2^m). Thus nb $\in$ (lower, upper) if nb $>$ lower (that is between lower and 2^m) or nb $<$ upper (that is between 0 and upper). For the other bracket pairs, it is the same principle. This function will be used through the whole project.

`Chord-v1-2.lua` implements both algorithms 1 and 2 provided in the instructions.

2.2 Implementation of the finger table

`Chord-fingers-v1.lua` provides an implementation of the Chord protocol with a finger table. From the previous implementation, we changed the function `init_neighbors` to

three functions, namely `init_finger_table`, `update_finger_table` and `update_others`. The function `join` was also changed to initialize the finger table when a node joins and to tell the other node in the ring to update their finger table. Finally, the function `find_predecessor` was split into the function `closest_preceding_finger` and `find_predecessor`. In view of task 3.4, we already implemented the hops' counter for the queries.

The finger table is initalized as follow:

Listing 1: Initialization of the finger table for each node

```
finger = {}
for i = 1, m do
    finger[i] = {node = nil, start = (n.id + 2^(i-1)) % 2^m}
end
finger[1].node = n
```

Before joining the ring, the nodes know no other node, so all the fingers are initialized to nil. The start is where to start searching in the ring, this does not change so we initialize them to their correct value. Finally we simply set `finger[1].node` to `n`, that is the node itself is its own successor.

2.3 Search performances

We start analyzing the search performance in the basic Chord implementation. We used a ring with 64 nodes and 500 queries per node. We randomly generated the 500 keys as follows:

Listing 2: Generating random keys

```
    for j = 1, n do
rand_number = compute_hash(math.random(0, 2 ^ m))
local _, i = find_predecessor(rand_number)
print("Number of hops:", i)
print("Key to find:", rand_number)
    end
```

In order to parse the produced logs, we used the following code in bash:

Listing 3: Parser for the logs

```
grep "Number of hops" Logs/log-task2-5-cluster.txt | sed -r
    's/.*\t([0-9]{1,2})$/\1/g' | sort -n | uniq -c | sed -r 's/
    +([0-9]+)/\2 \1/g' >> ParsedLogs/log-task2-5-cluster.txt
```

We produced the plot shown in Figure 1. We see that the we found approximately the same number of keys for each number of hops. This tells us, as we expected that the search time in the basic Chord is $O(n)$ where n is the number of nodes in the ring.

For the Chord implementation, we used the same parameters and the same code to generate the keys. We obtained the results shown in Figure

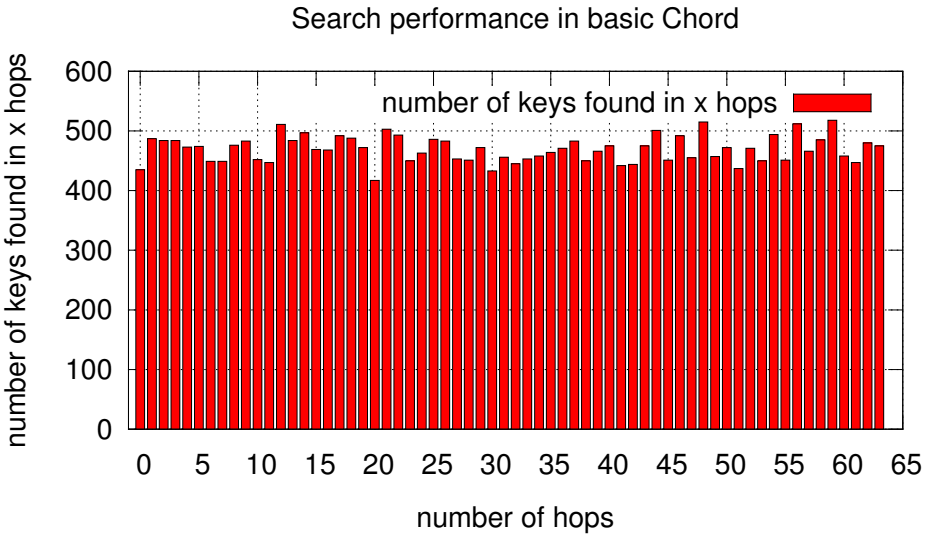

Figure 1: Search performance in basic Chord

Figure 2: Search performance in Chord with a finger table