
Large-Scale Distributed Systems

Project 2: Distributed Hash Tables

Laurent HAYEZ

November 23, 2015

Table of contents

1	Introduction	1
2	The base Chord protocol	1
2.1	Implementation without finger table	1
2.2	Implementation of the finger table	2
2.3	Search performances	2
3	The Fault-Tolerant Chord Protocol	3
3.1	Implementation of the fault-tolerant Chord protocol	3
3.2	Analysis of the fault-tolerant Chord protocol	3

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, $\text{lower} < \text{upper}$ and $\text{upper} < \text{lower}$. In the first case, $\text{nb} \in (\text{lower}, \text{upper})$ if $\text{nb} > \text{lower}$ and $\text{nb} < \text{upper}$. 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 $\text{nb} \in (\text{lower}, \text{upper})$ if $\text{nb} > \text{lower}$ (that is between lower and 2^m) or $\text{nb} < \text{upper}$ (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. As `successor = finger[1].node`, we deleted the variable `successor` and instead called the function `get_successor`, which returns `finger[1].node`.

The finger table is initialized as follow:

Listing 1: Initialization of the finger table for each node

```
1 finger = {}  
  for i = 1, m do  
3     finger[i] = {node = nil, start = (n.id + 2^(i-1)) % 2^m}  
  end  
5 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

```
1     for j = 1, n do  
      rand_number = compute_hash(math.random(0, 2 ^ m))  
3     local _, i = find_predecessor(rand_number)  
      print("Number of hops:", i)  
5     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

```
1 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 with a finger table, we used the same parameters and the same code to generate the keys. We obtained the results shown in Figure 2. As we expected, we can see that we obtain the keys in $O(\log(n))$ hops. To compare both plots, we put them together in Figure 3. On this figure we clearly see how much the finger table helps to improve the search performance.

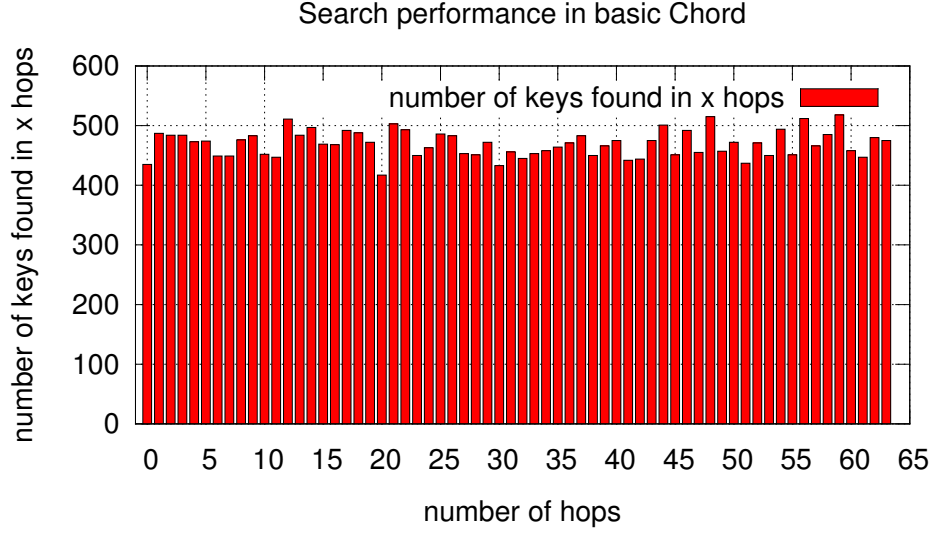


Figure 1: Search performance in basic Chord

3 The Fault-Tolerant Chord Protocol

In the previous section, we assumed that once a node joined the ring, it would not leave it. In this section we present an implementation of Chord that can recover from nodes suddenly leaving the ring. We will analyze this implementation by looking at the stale references in the finger table. Finally we will evaluate the search performance under churn.

3.1 Implementation of the fault-tolerant Chord protocol

We changed a little bit the join function from the given algorithm. We implemented it as follows:

Listing 4: join(n') for the fault-tolerant Chord protocol

```

1 function join(n1)
2   if n1 then
3     predecessor = nil
4     set_successor(rpc.call(n1, {"find_successor", n.id}))
5   else
6     predecessor = n
7     set_successor(n)
8   end
9 end

```

The implementation of the fault-tolerant Chord protocol is provided in the file `Chord-fingers-v2.lua`. There is nothing tricky in this implementation, apart from the previously discussed parts.

3.2 Analysis of the fault-tolerant Chord protocol

We start analyzing the average number of stale references under churn. We consider a reference stale if it points to a node not available anymore, ie a non nil node not responding. To count the stale references, we used the following code:

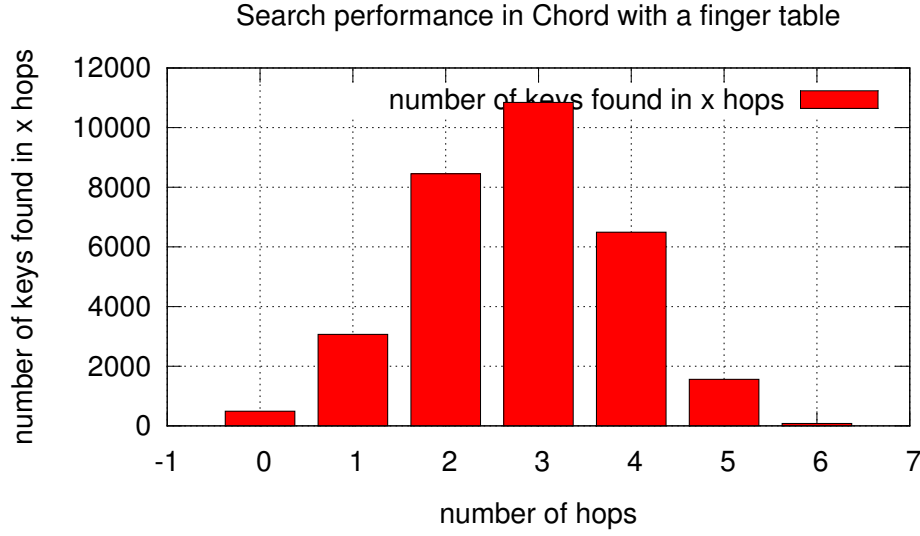


Figure 2: Search performance in Chord with a finger table

Listing 5: Check for the stale references

```

1 function check_fingers()
2   for i = 1, m do
3     if finger[i].node and not rpc.ping(finger[i].node) then
4       print("Stale reference to finger["..i.."]")
5     end
6   end
7 end

```

We made various plots, each with one different parameter varying. We used a set up of 64 nodes for the ring, and the provided churn trace to simulate the nodes leaving the ring. On Figure 5, we see that when we set the stabilization period to 10 seconds, the fix_fingers period to 20 seconds and the check_fingers period to 20 seconds, we have less stale references than for the other periods. That result was surprising, so we launched the same experiment one more time, but as you can see the result was similar.

We expected that the more we update the fingers, the less stale references we would obtain, but it is not what happens on Figure 5. When the fix_finger period is 5 or 10 seconds, the results are quite similar, but surprisingly, when it is set to 20 seconds there are less stale references. For Figures 4 and 6, we have approximately the same number of stale references, which is what we expected since we updated the fingers at the same frequency for all experiments

We now analyze how the nodes recovers from churn. It seems like the

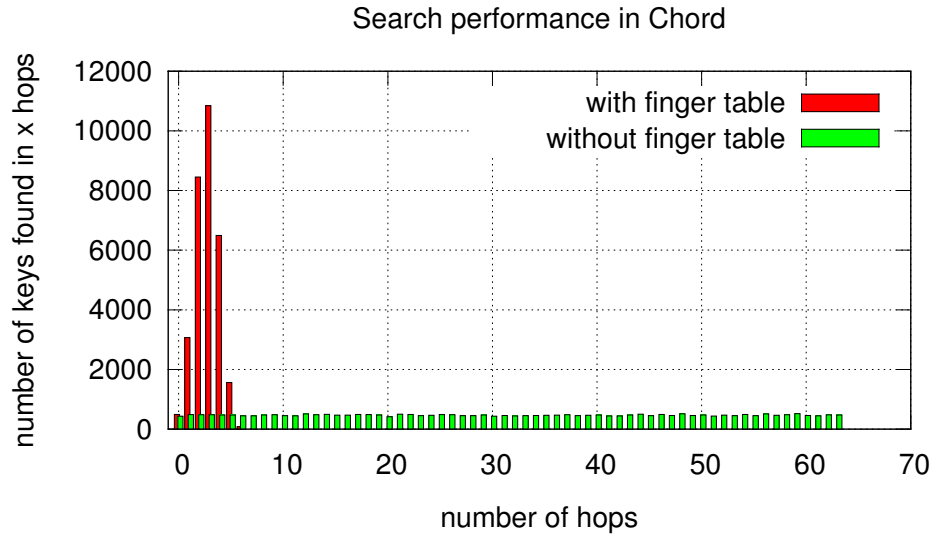


Figure 3: Comparison of the search performances in Chord

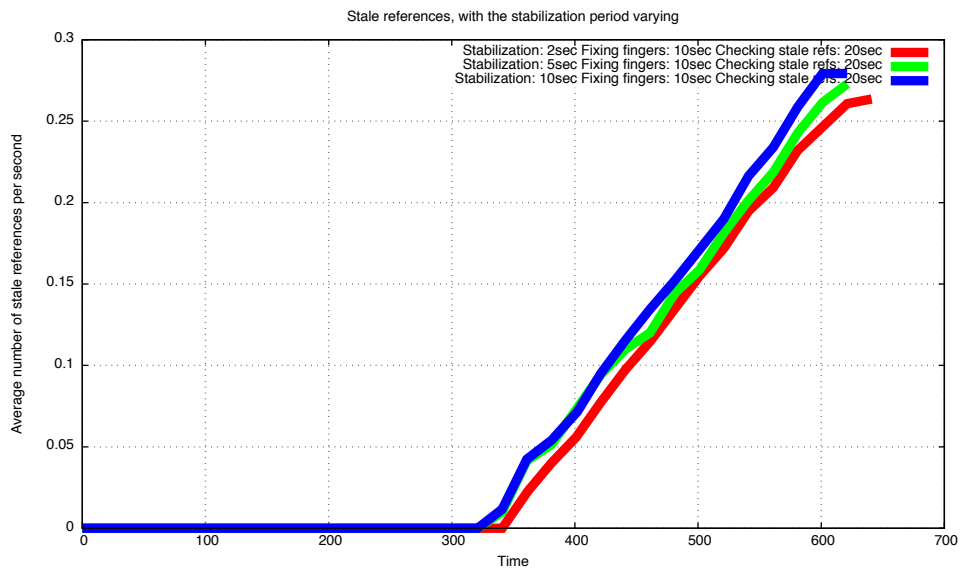


Figure 4: Average number of stale references, with the stabilization period varying

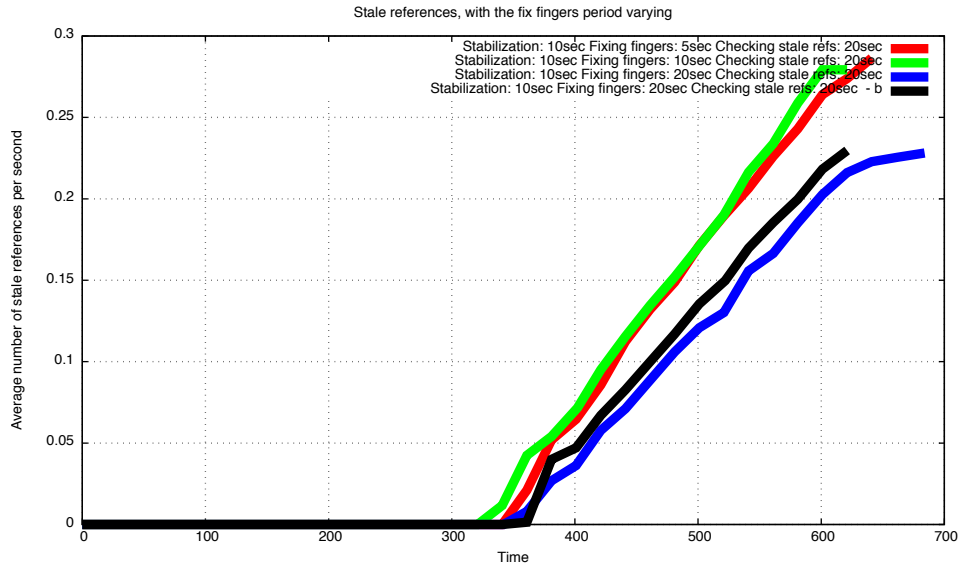


Figure 5: Average number of stale references, with the fix_fingers period varying

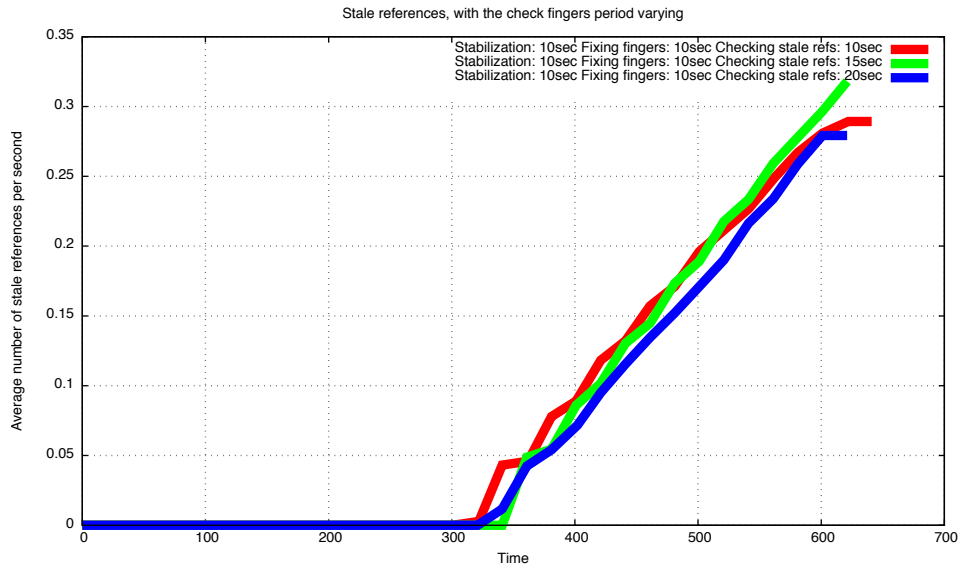


Figure 6: Average number of stale references, with the check_finger period varying

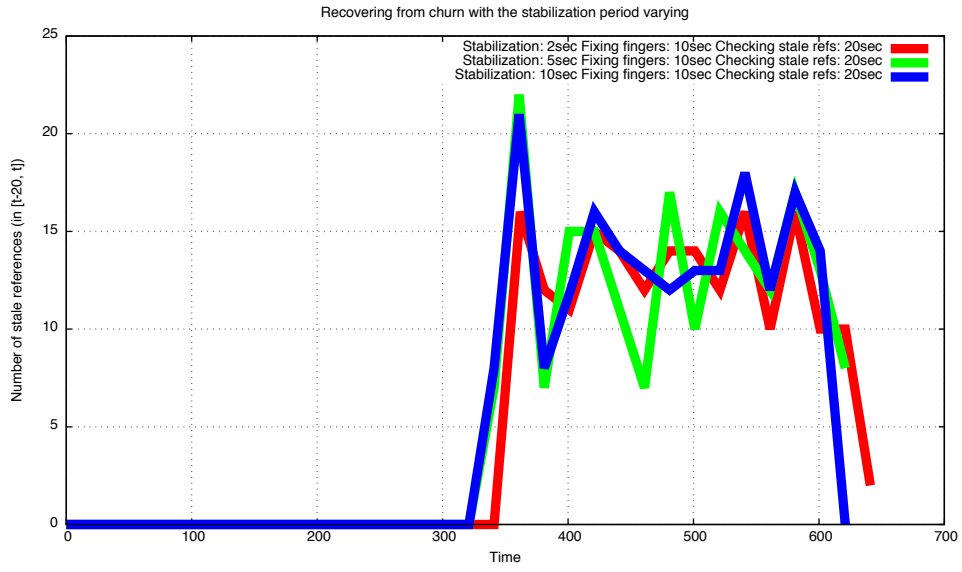


Figure 7: Recovery of the finger table after churn, with the stabilization period varying

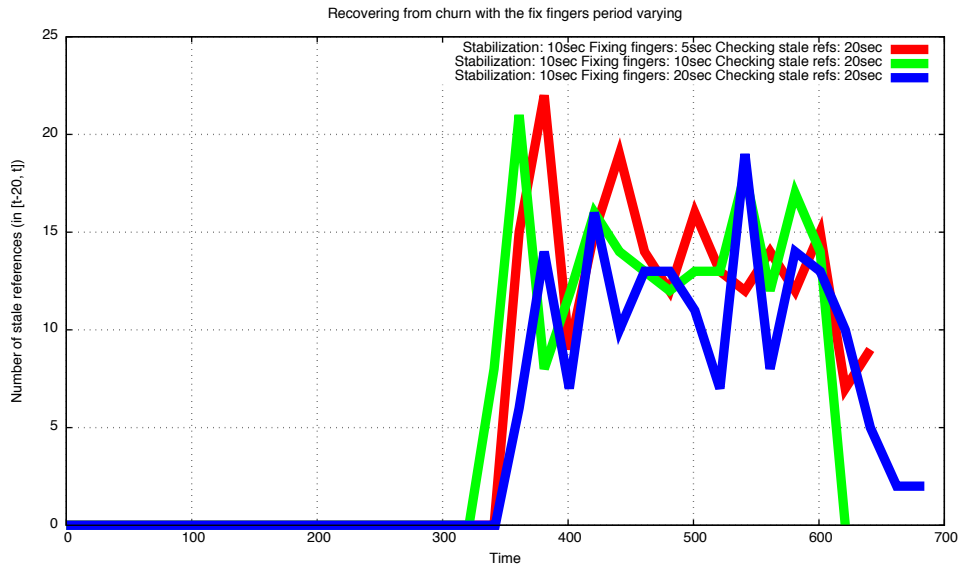


Figure 8: Recovery of the finger table after churn, with the fix_fingers period varying

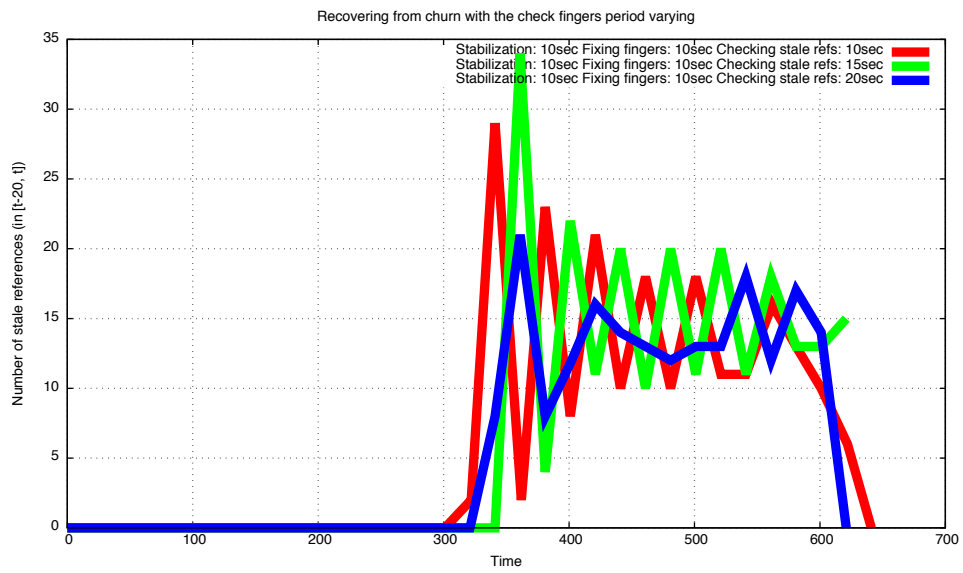


Figure 9: Recovery of the finger table after churn, with the check_finger period varying