Large-Scale Distributed Systems
Project 2: Distributed Hash Tables

Laurent Hayez

November 24, 2015

Table of contents

1 Introduction **1**

2 The base Chord protocol **1**
2.1 Implementation without finger table 2
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 4

4 Conclusion **6**

1 Introduction

The main goal of this project is to implement Chord protocol in a basic way. Chord is a distributed lookup protocol that has a key-routing layer. The key-routing mechanism then permits to implement a distributed hash table on top of it.

We will start by implementing Chord with only successors and predecessors. This implementation is not optimal, as the search time is $O(n)$. But we can do much better by adding a finger table to each node, which will reduce the search time $O(\log n)$. This will be our next step.

Finally, we will modifiy some functions in order to have a fault-tolerant implementation. That is the nodes can recover from other node's failure by updating periodically their successor and their finger table. We will analyze how nodes failures affect the system by looking at stale references in the finger tables, and we will also take a look at the search performance under churn.

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. 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 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.

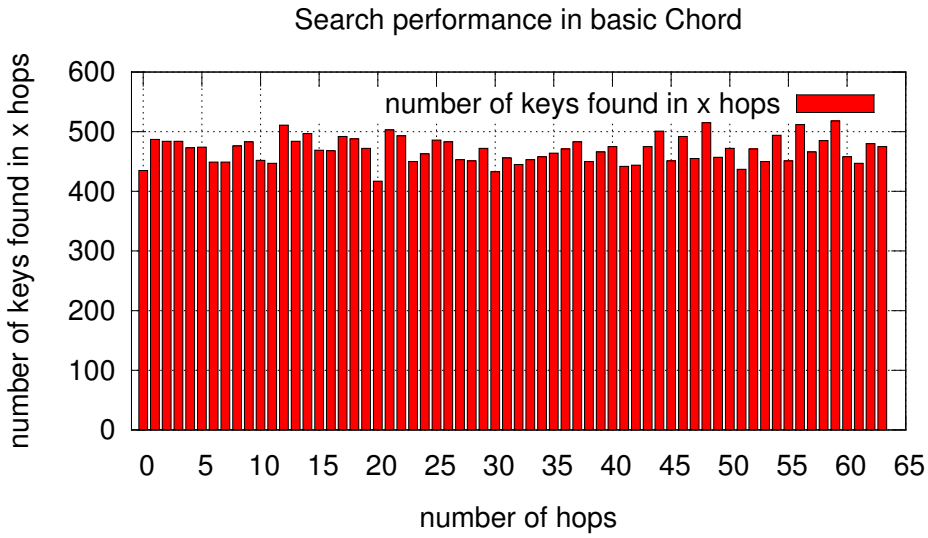

Figure 1: Search performance in basic Chord

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.

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:

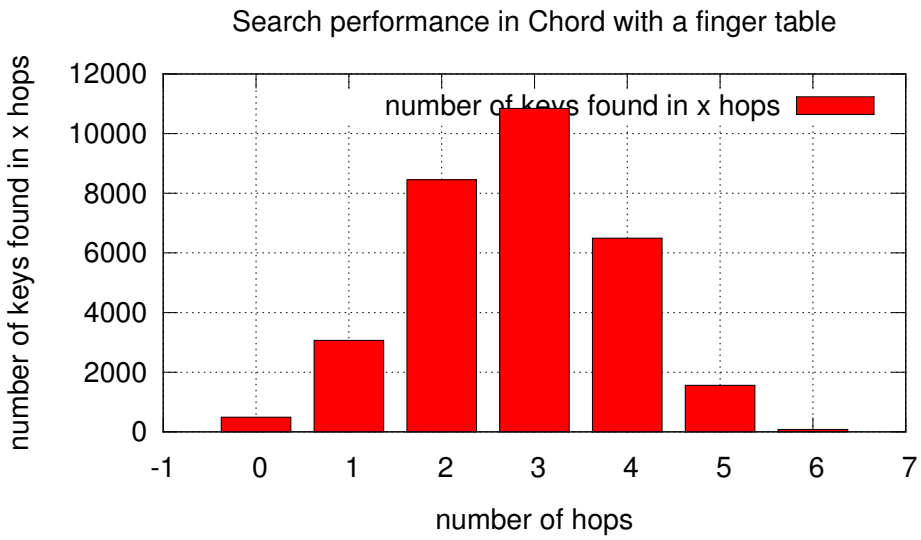

Figure 2: Search performance in Chord with a finger table

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

```
function join(n1)
    if n1 then
        predecessor = nil
        set_successor(rpc.call(n1, {"find_successor", n.id}))
    else
        predecessor = n
        set_successor(n)
    end
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, appart 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:

Listing 5: Check for the stale references

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

The average number of stale references was calculated as

$$\frac{\text{current number of stale references at current time}}{\text{total elapsed time}}.$$

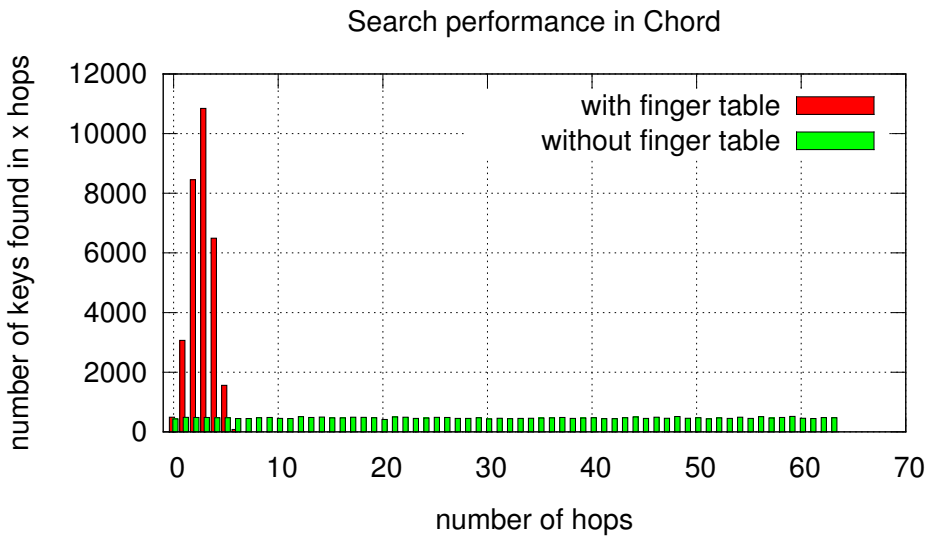

Figure 3: Comparison of the search performances in Chord

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. We can see on Figures 7, 8 and 9 that the finger table does not recover well from churn. This is due to the function `fix_fingers`, shown in Listing 6.

Listing 6: Function fix_fingers

```
function fix_fingers()
    i = math.random(2,m)
    finger[i].node = find_successor(finger[i].start)
end
```

This finger simply select a random finger and updates its reference, but the chosen finger might be a valid one and the stale entries will thus not be updated. And as this is done every 5, 10 or 20 seconds it would take a long time to remove all the stale entries of the finger table. A possible amelioration of this function could thus be to get the best of the functions `fix_fingers` and `check_fingers` to as shown in Listing 7.

The results with this function are shown in Figures 10 and 11. The amelioration is not transcendent, but we see that we have approximately two times less stale references than before.

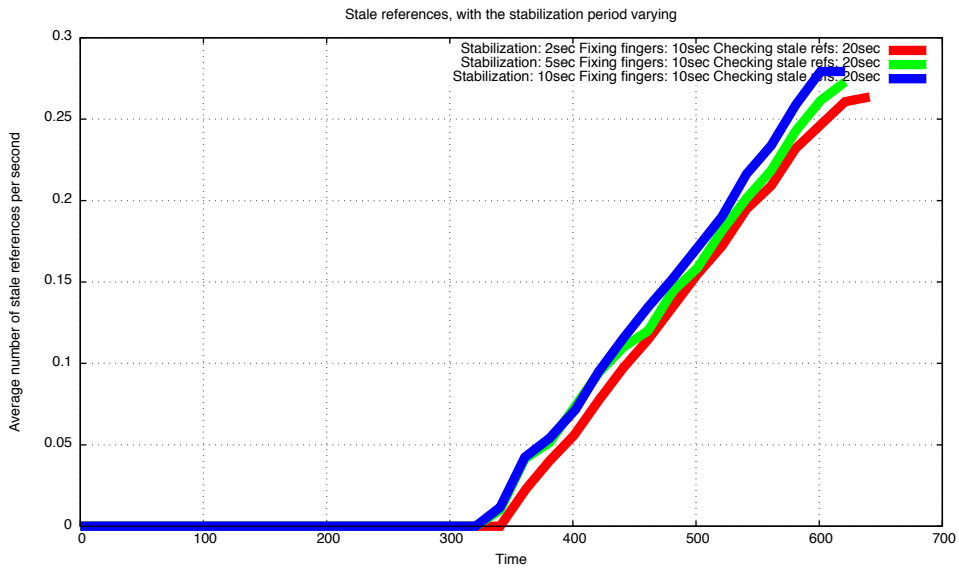

Figure 4: Average number of stale references, with the stabilization period varying

Listing 7: Function fix_fingers ameliorated

```
function fix_fingers()
    for i = 1, m do
        if finger[i].node and not rpc.ping(finger[i].node) then
    finger[i].node = find_successor(finger[i].start)
        end
    end
end
```

We also see that, unlike before, this time the best result is obtained when we set the stabilization period to 10 seconds, the fix_fingers period to 10 seconds and the check_fingers period to 20 seconds.

Let's now take a look at Figures 12 and 13. These graphs respectively represent the percentage of successfully found keys and the number of keys found in x hops, that is the search performance in Chord with churn. On the first one, we observe that while there is no churn, the success ration is 100% (except at the beggining, where there was a little bug). Although there is churn, the success ratio is still over 99% after the churn. This is due to the fact that the successors of each node are periodically updated in order to have a reliable path, even if we have stale references in the finger tables. We can see that on Figure 13. If we compare this histogram with Figure 2, we see that it takes a little bit longer to find a key when there is churn because of the stale references, but this is still more efficient than having only a reference to the successor and to the predecessor.

4 Conclusion

The goal of this project was to implement a distributed hash table on top of a key-routing layer. To do so, we implemented Chord, with multiple steps. We started modestly by implementing a basic Chord protocol that could search key values in $O(n)$ hops. Then we added a finger table to each node that boosted the search performance to $O(\log n)$

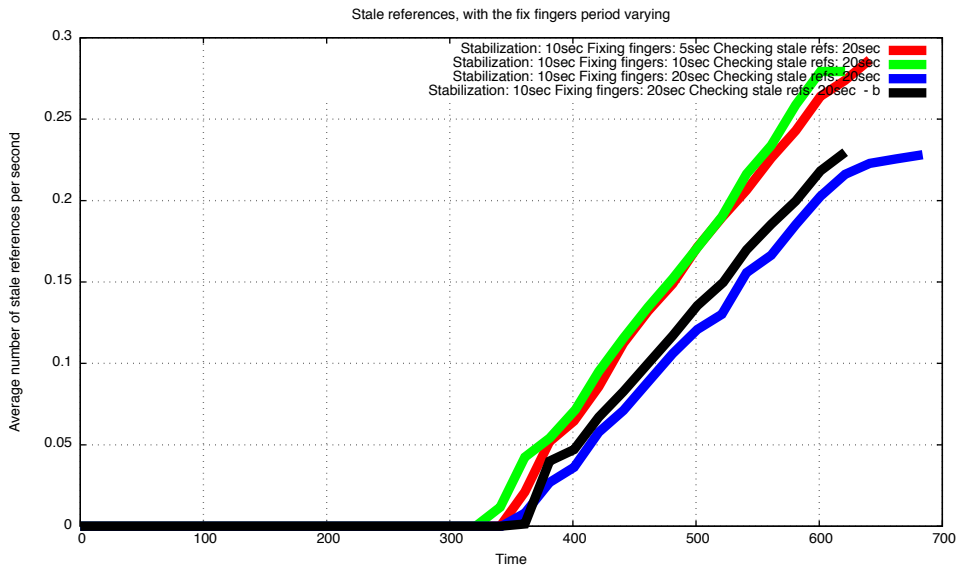

Figure 5: Average number of stale references, with the fix_fingers period varying

hops. After that, we consolidated the implementation by updating periodically the successor and the finger table for each node. We observed that updating one random finger at a time was not the best way to recover from churn, and that periodically updating the whole table was a little bit better. The search performance with churn was not too deprecated, as more than 99% of the queries were satisfied. The price to pay was a bit more hops to find the desired key, but it was still more efficient than the first Chord implementation we presented.

There are some ameliorations we could bring to what we have done. For example instead of having only one successor, we could maintain a list of n successors and predecessors, in case the direct successor or predecessor fails. We could also, as presented, update lazily the whole finger table periodically.

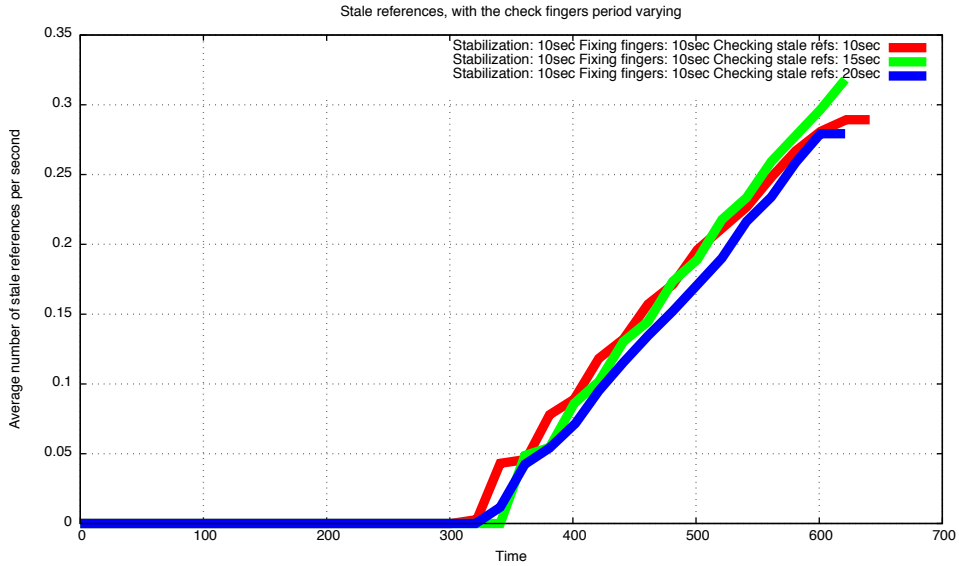

Figure 6: Average number of stale references, with the check_finger period varying

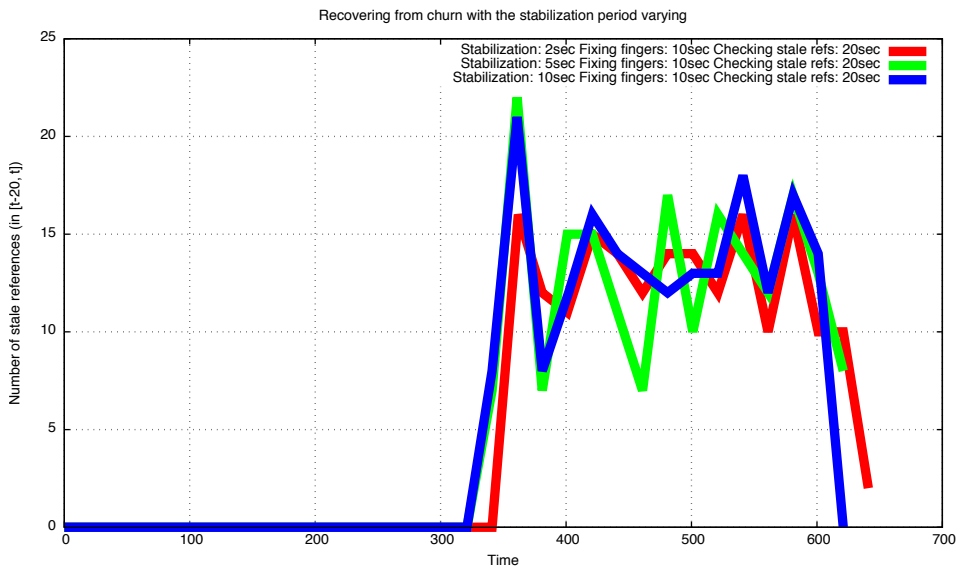

Figure 7: Recovery of the finger table after churn, with the stabilization period varying

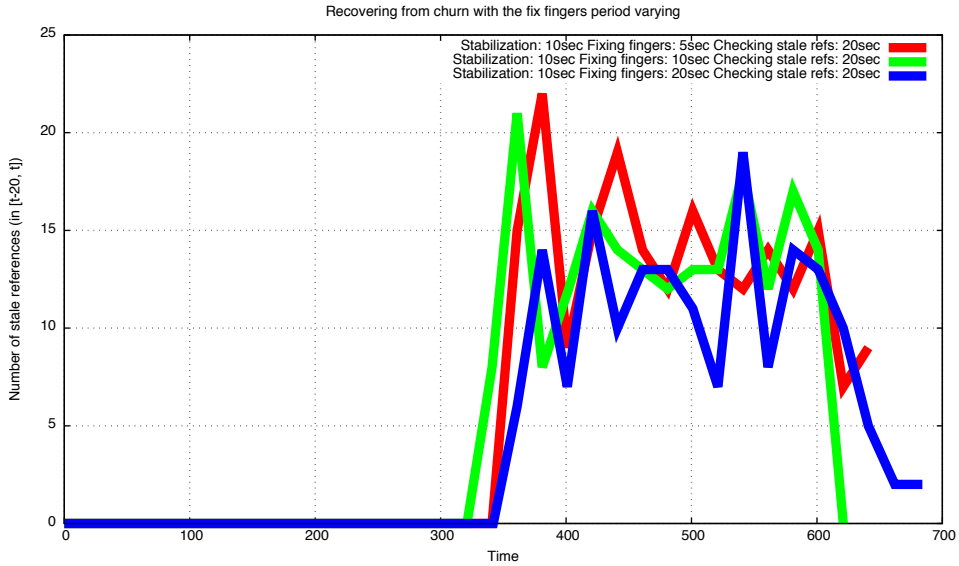

Figure 8: Recovery of the finger table after churn, with the fix_fingers period varying

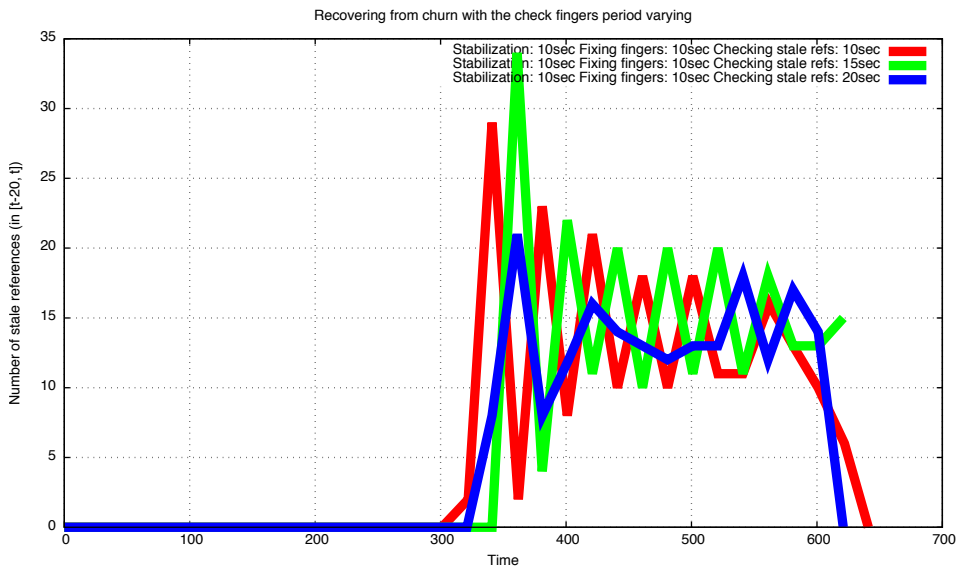

Figure 9: Recovery of the finger table after churn, with the check_finger period varying

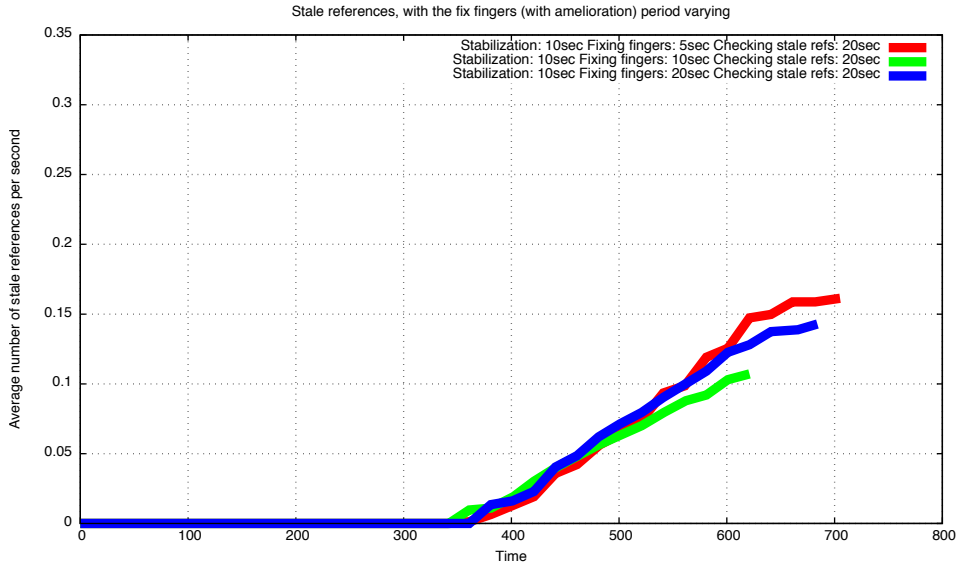

Figure 10: Average number of stale references, with the ameliorated fix_finger function

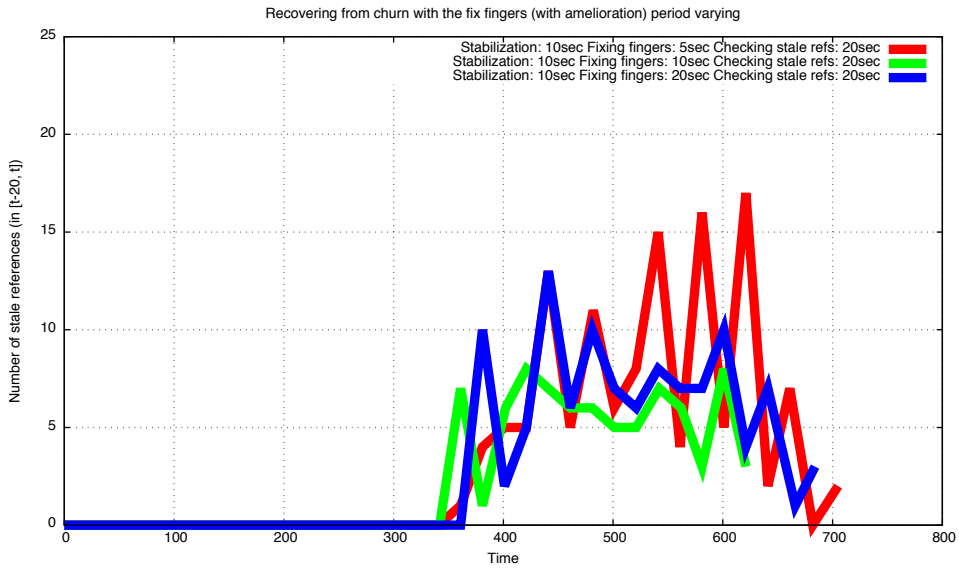

Figure 11: Recovery of the finger table after churn, with the fix_finger (ameliorated) period varying

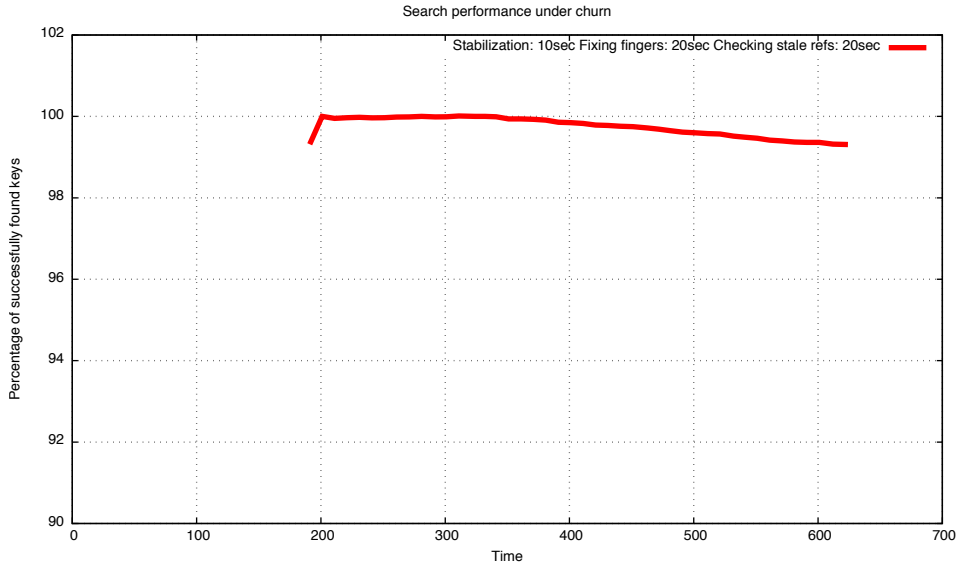

Figure 12: Percentage of successfully found keys

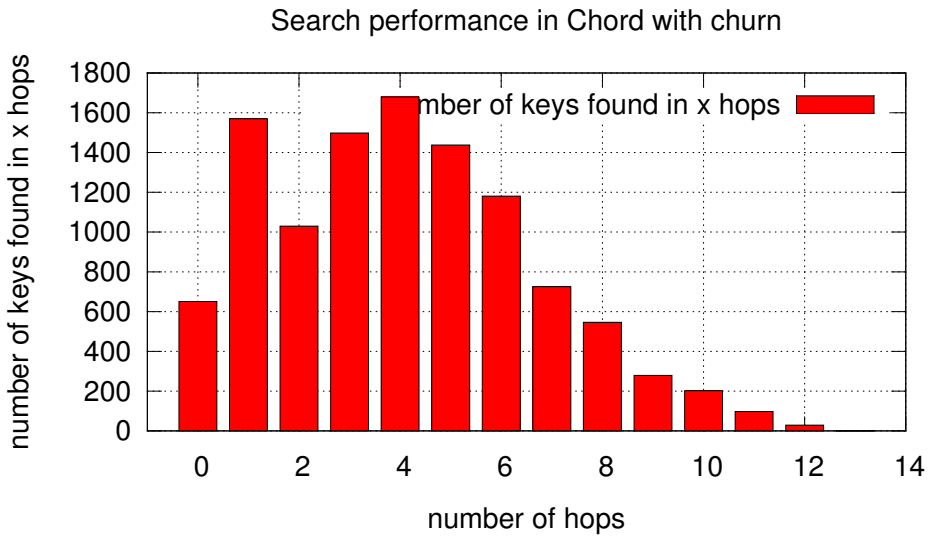

Figure 13: Search performance in Chord with churn