Distributed Hash Tables

Large-Scale Distributed Systems

Sébastien Vaucher
sebastien.vaucher@unine.ch

26 November 2015

Contents

MASTER IN COMPUTER SCIENCE

unine
UNIVERSITÉ DE NEUCHÂTEL

1 Introduction

This report presents the results obtained in the second assignment of the Large-Scale Distributed Systems course taught at the University of Neuchâtel. The goal of the assignment is the implement a distributed hash table using the Chord algorithm. The implementation is done in the Lua programming language, using the Splay framework.

Apart from this report, a number of files are supplied:

dht.lua
: Contains the implementation of the fault-tolerant DHT.

dht-noft.lua
: Contains the implementation of the DHT without fault-tolerance (task 2 of the assignment).

dht.sh
: Bash script to launch the program on a local machine.

***.txt**
: Raw logs generated by the program running on the cluster of the university.

parse_stale.pl
: Perl script to generate the number of stale nodes over time.

parse_task-34.pl
: Perl script to generate the number of failed queries over time.

***.gp**
: Gnuplot scripts generating the graphs found in this report.

generate_graphs.sh
: Script to generate the plots found in this report from the raw logs. Some plot data is directly generated by this script, while other are delegated to Perl scripts.

dht.churn.txt
: Churn trace to upload to the SplayWeb interface to simulate churn. Is as was given by the instructors.

All the data presented in this report is the result of executions on the Splay cluster of the university.

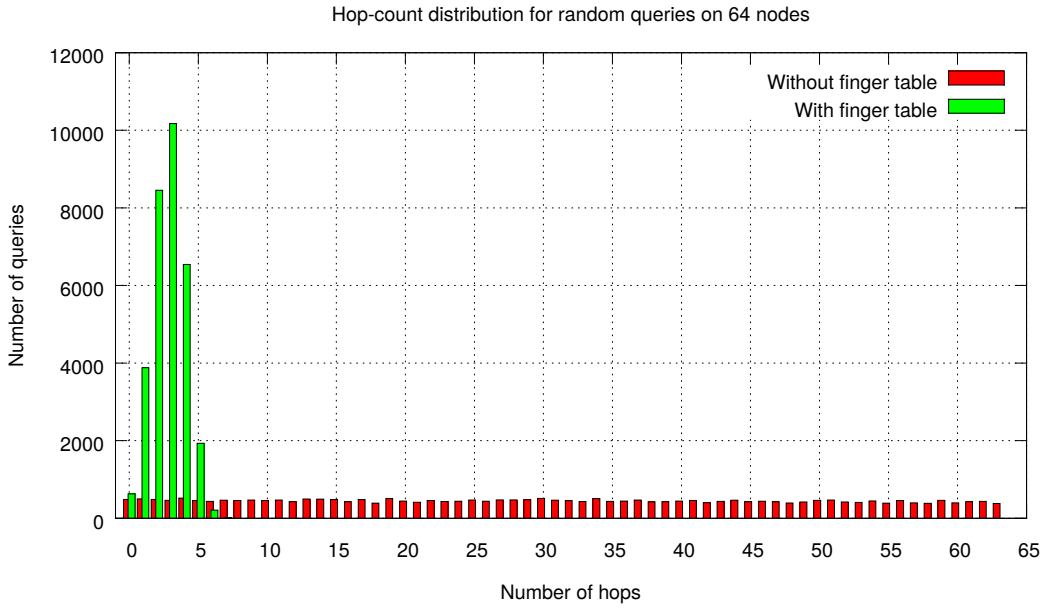

Figure 1: Distribution of the number of hops for random queries, with and without a finger table

2 Search performance of the base Chord protocol

The goal of a Distributed Hash Table is to store various objects under certain keys. The responsibility for each key is defined by the identifier of a node and its position in the ring. To assess the search performance of our DHT, we queried the ring 500 times per node. We then recorded the number of nodes that had been traversed to reach the node responsible for each key.

Figure 1 shows the number of hops that have been traversed on the abscissa. The ordinate is the number of occurrences for a particular hop-count. The lowest the hop-count, the better. Therefore, the more "left-aligned" the distribution, the better.

2.1 Without a finger table

The first implementation of our DHT only used a single pointer towards the successor on the ring. Its performance is represented with red bars on Figure 1. In this configuration, a given key can be found by sending a query around the ring until the responsible node receives the query. The traversal of the ring is unidirectional, therefore the hop-count for a query is bounded by $[0, RingSize - 1]$, where $RingSize$ is the number of nodes in the DHT. 0 indicates that the node sending the query is already the responsible node (the

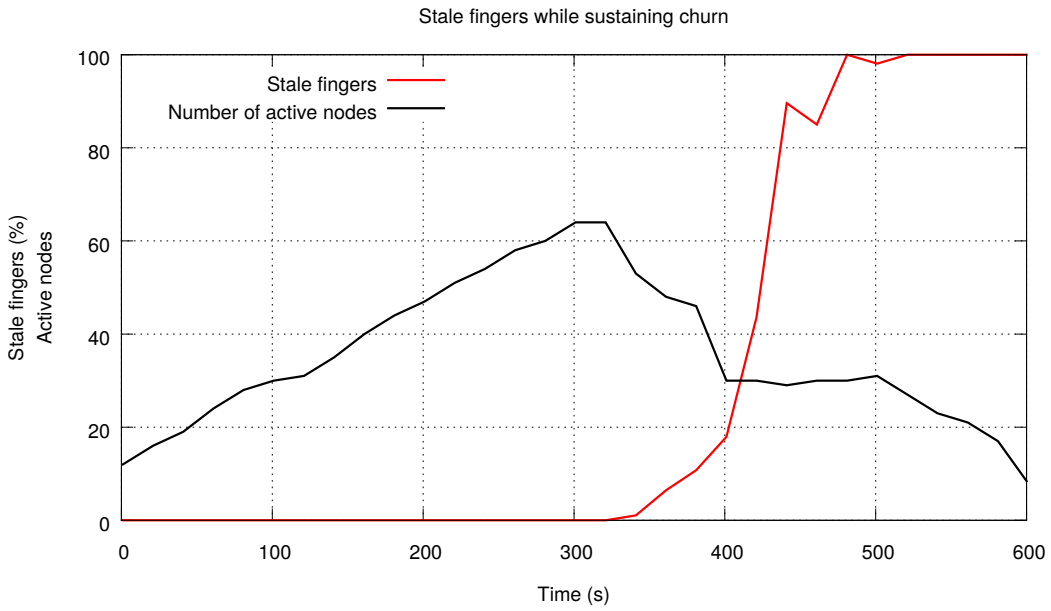

Figure 2: Percentage of stale entries in the finger table under churn

best-case scenario), $RingSize - 1$ is the worst case: all the nodes have been traversed until the responsible node gets found.

From the plot, we can conclude that the distribution of hop-counts is uniform in this configuration. The performance is therefore abysmal. In a DHT with more nodes, this implementation would be unusable.

2.2 With a finger table

The second implementation adds a so-called finger table to each node. A finger table maps a set of keys to specific nodes. Each node stored in it serves as a shortcut to a certain range of keys. We can therefore reach responsible nodes in less hops.

In Figure 1, the green bars show the improved performance brought with the introduction of the finger table. The worst case is now as low as 6 hops, versus 63 without this optimization. The finger table undoubtedly tremendously improve the performance of searching keys within the DHT.

3 Fault-tolerant Chord protocol

In a realistic setup, nodes are able to join or leave the DHT at any time. The Splay framework has a feature to simulate this behavior: churn traces. A churn trace is a file

listing start times and end times for each node. At each start time, a node tries to join the DHT, while at an end time a node suddenly dies.

3.1 Stale entries

In the perspective of a well-functioning node, a pointer towards a dead node is still valid (i.e not nil). Finger table entries that are not nil but point to a dead node are called stale entries. Figure 2 shows the number of stale entries over time when churn happens. We can see that the rise in stale pointers is concurrent with the death of some nodes. At approximately $t = 500\,\text{s}$, the ring reaches a state of no return where all tested fingers point to dead nodes.

The rise in stale entries has an obvious cause: the sudden death of multiple nodes, making all pointers towards them stale. In the beginning of the churn, from 300 s to 400 s, we can observe that the algorithm manages to stabilize the ring structure and keep the number of stale entries low. The percentage of stale entries stays below 30 % until the 460^{th} second. After that mark-point, almost all fingers are reported are stale. Please note, however, that the absolute total number of non-nil fingers is more than an order of magnitude lower than in the period of "normal" operation[1].

My explanation for this result is that the majority of nodes get stuck while their requests to dead nodes are timing out. The churn trace kills all nodes where the values stop being reported. It would be interesting to see whether the ring can eventually recover when the system stays stable again.

3.2 Performance under churn

The relevant plots for this section are in Figure 3 and Figure 4. Figure 3 shows the distribution of hop-counts for random queries, along with the amount of queries that failed. Figure 4 shows the evolution of the average hop-count and the percentage of failed queries over time.

Figure 3 tells us that the performance of the fault-tolerant algorithm is similar to its basic counterpart (in Figure 1). The finger table does its job considerably well, and the distribution clearly resembles that of a Gaussian.

In absolute, the number of failed queries is really low, as shown in Figure 3. However, when looking at the relative percentage of failed queries in Figure 4, we see that the protocol more-or-less stops operating correctly after some nodes die. The cause of this effect is discussed in subsection 3.3. The average hop-count also plummets to zero, indicating that the rare queries to succeed are the ones where the responsible node is the one initiating the query.

[1] The values can be obtained by looking at the stale-churn.txt.plotdata file generated by generate_graphs.sh.

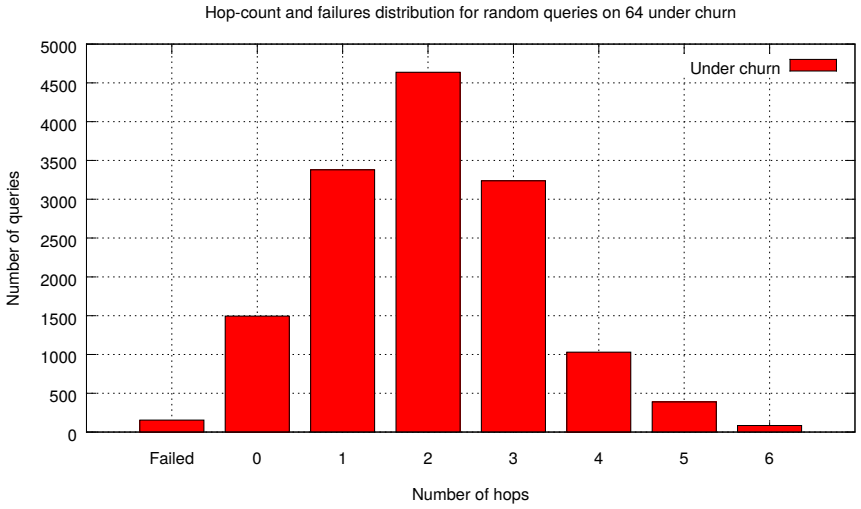

Figure 3: Distribution of the number of hops for random queries under churn

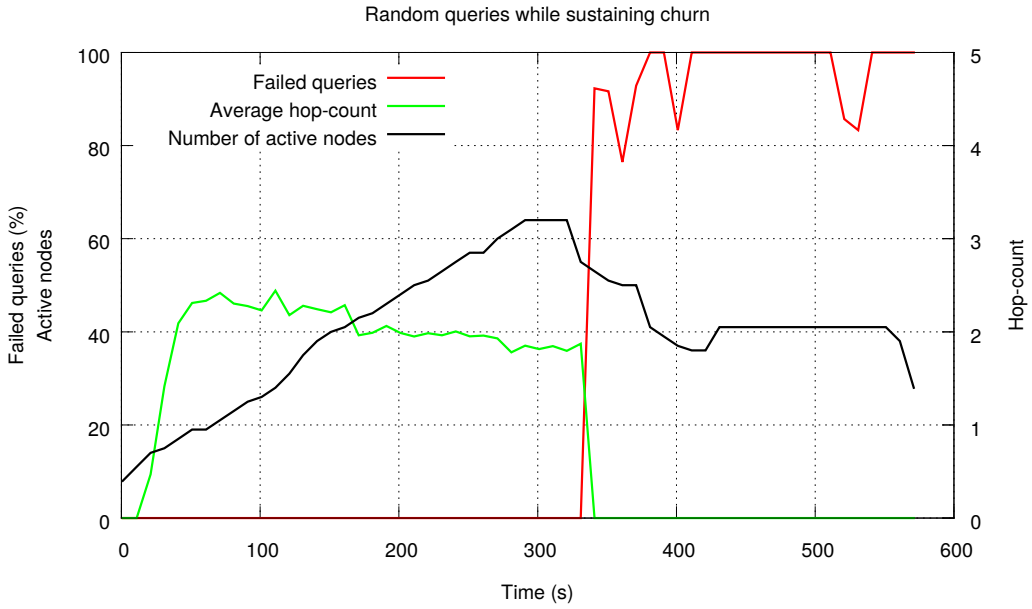

Figure 4: Evolution of the hop-count and percentage of failed queries under churn

As far as performance is concerned, we can observe that while the number of nodes in the system grows, the average hop-count goes down. Without finger tables, the average hop-count would tend to be half the number of active nodes in the DHT. However, in this case, fingers are created fast enough to still ameliorate the performance of the system. Therefore, nodes gradually joining the DHT have no effect on searching performance.

Removing only a small fraction of the nodes does not have an immediate effect. The queries still succeed in the handful of seconds following the first deaths. Removing more nodes at a time has catastrophic consequences, as is outlined in next subsection.

3.3 Major issue of the implementation

As we saw in subsection 3.2, the Chord ring has trouble reconstructing itself after some nodes die. The explanation can be derived from the way the stabilize function operates. The first thing it does is get the successor of the current node. It will then issue a Remote Procedure Call on it. The problem appears when the successor dies. The stabilize function cannot operate anymore, therefore the ring will never recover from churn.

A second problem arises when nodes die: timeouts. We use the implementation of Splay to perform RPCs, which uses UDP as the transport protocol. When a node die, requests addressed to it will timeout. In our implementation, we set that timeout to 5 s. That means that every RPC to a stale node will take 5 s to basically do nothing useful. All the nodes – in the addition of now being part of a "broken ring" – cannot operate in due time. This is highlighted in the raw logs[2] by the profusion of messages saying that periodic tasks have missed a tick (a.k.a the task took longer than the period) towards the end of the execution.

In Figure 4, we noted that the queries can still complete when only a small number of nodes die. This situation can be explained by the fact that the majority of queries can still resolve successfully, even when some fingers are stale. The combination of all finger tables in the system provides enough redundancy for that.

[2] In task-34.txt

4 Conclusion

We have shown that the Chord protocol clearly works. The addition of the finger table is capital for any useful implementation. It drastically improves the performance of the search operation.

Chord's ring membership management is dynamic, so nodes can join at any time with no measurable impact on the existing structure. Each node in a Chord ring will self-optimize over time, as it fills its finger table.

The fault-tolerant version of the Chord protocol that we were asked to implement is not really fault-tolerant. It can deal with temporary outages of a couple of nodes, but cannot recover when a node definitely leaves the ring. This situation could be improved by storing a list of the next k successors in place of only the direct successor.