Analysis of a Chord overlay

Peer to Peer Systems and Blockchains

Master’s Degree in Computer Science and Networking
Academic Year 2017/2018

ORLANDO LEOMBRUNI

23-04-2018

Contents

1. Introduction

The project concerns the design and implementation of a simulation of a Chord overlay network. A central Coordinator builds the overlay and performs the simulation of a certain number of queries, logging all relevant data to perform statistical analysis.

1.1 Contents of the submission

The submission `zip` file contains, besides this report (in both PDF and source Markdown format):

- The `doc` folder, which contains a complete Javadoc documentation
- The `simulations` folder, which contains a full set of graphs and the data used to generate them
- The `src` folder, which contains the full source code for the project
- The `lib` folder, which contains the libraries used (*Apache Commons Codec* and *Google GSON*)
- The jar file `p2p-midterm.jar` for executing the simulation. More info can be found in the next section.

1.2 Executing the simulation

The simulation can be ran using the enclosed jar file. It takes two integer arguments:

- the first one is the *number of bits for the identifiers* n, which must satisfy $n\%4 == 0$ in order to correctly trim the hashed strings representing the identifiers;
- the second one is the *number of nodes* m, which must satisfy $m \leq 2^n$.

By default, the simulation performs exactly m queries of random keys. No option has been added to modify this value, in order to be compliant with the assignment requirements; although, the program is built in a way such that any number of queries can be generated (by modifiying line 55 of `Main.java`).

2. Implementation Choices

2.1 Identifiers

In order to support any identifier size (up to 512 bits), identifiers are stored as byte arrays inside objects of class `BigInteger`. This allows for easy mathematical operations and comparisons, losing some speed in every operation as a tradeoff.

2.2 The Node ring

The ring of Nodes is stored in a central entity (the `Coordinator`) in a map data structure, to allow for efficient operations; in fact, it is implemented as a `TreeMap`, which allows for $O(\log m)$ operations.

2.3 Creation of the overlay

The `Coordinator` is also the entity responsible for building the overlay. It creates m distinct nodes, as strings of the pattern "`IPAddress:Port`" where `IPAddress` is built from 4 random integers in the range $[0, 256)$ and `Port` is an integer in the range $[0, 65536)$. Each string is then hashed with SHA-512 and its representation truncated to n bits.

After checking that there's no collision, the `Coordinator` builds a new `Node` object that stores the generated data (both the `IPAddress:Port` string, that functions as the name of the node, and the hashed string that serves as its ID) and puts it on the ring.

After populating the ring, the `Coordinator` starts building the finger table for each node.

2.4 Finger tables

The finger table creation is rather straightforward: for each node with ID x and for each $i \in [0, n)$, the procedure computes the ID $d_i = (x + 2^i)\%2^n$, finds the nearest `Node` j such that `j.getID()` $\geq d_i$, and then sets `nodes(x).fingertable[i] = j`.

The selection of j is done in $O(\log m)$ time thanks to the `TreeMap.tailNode(id)` method, so the total time complexity of the finger table construction for all m nodes is is $O(nm \log m)$.

As described in the code comments, a little trick has been deployed in order to consume less space when storing the finger tables, especially for large n and small m. In this scenario, the finger tables are long, contain identifiers expressed with an high number of bits, and most of them refer simply to the node's successor.

In order to save space, the finger table creation is slightly changed: if at any time the generated i-th finger for a certain node is equal to the node's successor (i.e., the 0-th finger), the corresponding entry in the finger table is simply put to `null`. Of course, the lookup algorithm must perform the dual operation of substituting any `null` value with the node's successor when utilizing the finger table.

2.5 Simulation

The simulation consists of generating m random sequences of n bytes and hashing them (with the usual SHA-512 plus truncation). Then, nodes are shuffled and each one of them starts a different query searching one of the generated hashes via the `Node.lookup` method.

A `RouteLogger` object is passed to each invocation of `lookup`; as the name says, it logs and holds useful information about the query, like the number of hops and the ID of the endnode of the query. This information, along other collected directly from the `Coordinator`, is then used to compute statistics about the simulation.

2.6 Routing and Topology information

Each successful invocation of the program will generate two Comma-Separated Values (CSV) files, one inside the *topology* folder and another inside the *routing* folder. In both folders, the file will be placed in the path identified, in order, by:

- the number of nodes in the overlay
- the number of bits to represent the identifiers
- the time (hours, minutes and seconds) of execution of the main class.

More info on the collected data can be found in the next chapter.

2.7 Lookup

The lookup algorithm is implemented as a method inside the `Node` class. Apart from the abovementioned trick of considering any `null` reference in the finger table as a reference to the node's successor, the algorithm is identical to the one discussed in class (and on the Chord paper specification) so it won't be further commented on.

3. Results and Graphs

This chapter will be a short summary on quantitative results obtained by running the simulation many times on different numbers of nodes. In order to keep this report concise, only the results plotted from a simulation with 4096 nodes will be shown in order to allow comparisons with the results in the Chord specification paper; the same plots for 1024, 2048, 8192, 16384, 32768 and 65536 nodes can be found in the `simulations` folder.

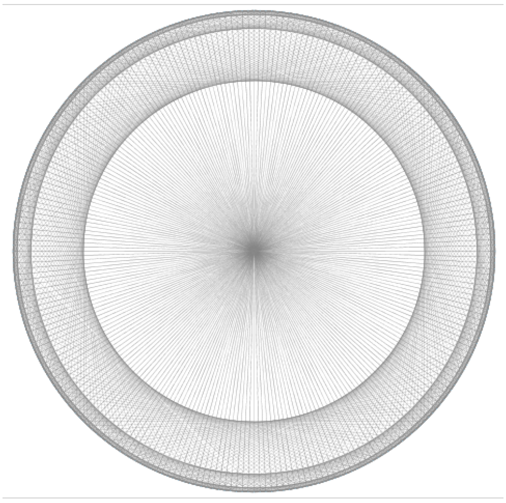
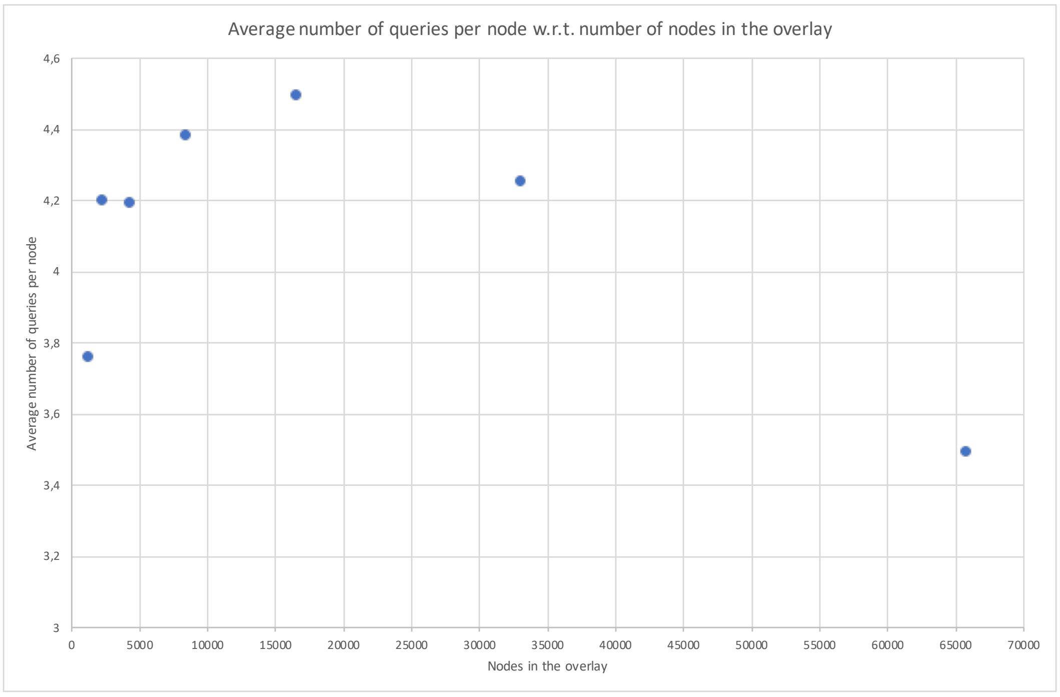


Figure 1: *A Chord overlay of 256 nodes with identifiers of 2^8 bit*

3.1 Aggregate results

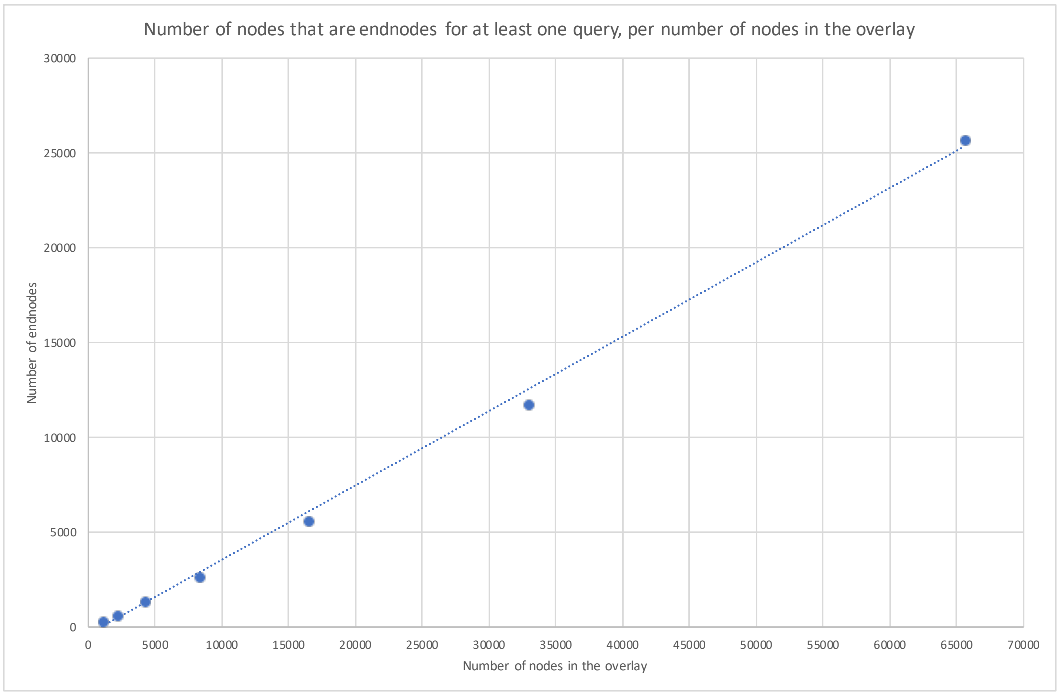
3.1.1 Average number of queries per node w.r.t number of nodes



This plot shows the average number of queries each node receives, when the number of nodes in the overlay changes. Note that also the number of queries changes as the number of nodes grows (a simulation with m nodes will perform m queries).

The behavior is the one of a power law; the maximum is reached when the simulation is on 16384 nodes and queries.

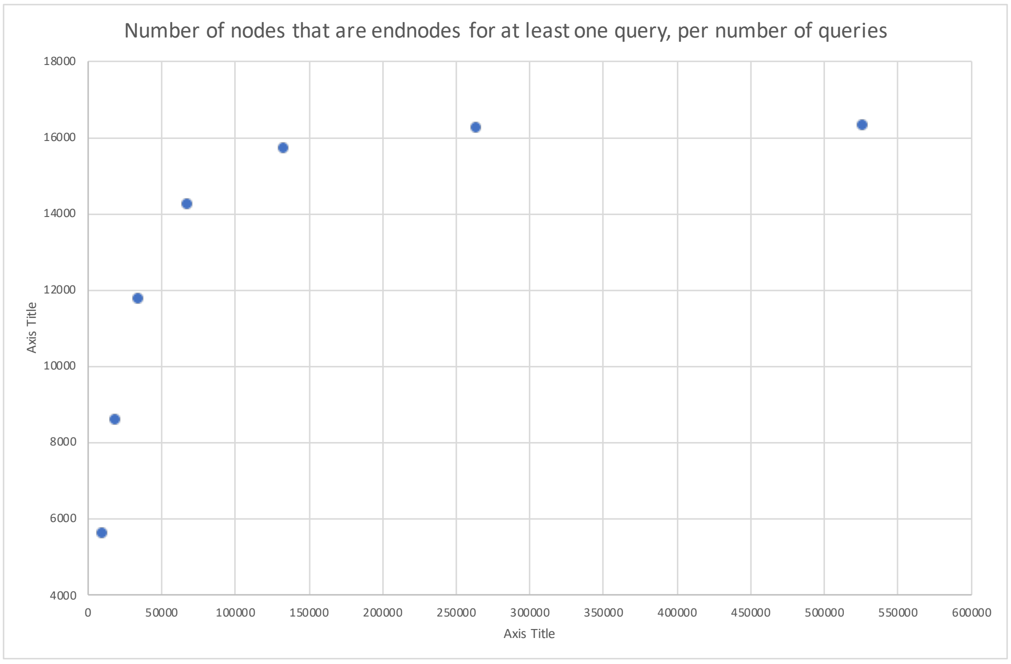
3.1.2 Number of endnodes w.r.t. number of nodes



This plot shows how many nodes in the overlay are endnodes for at least one query, as the number of nodes grows.

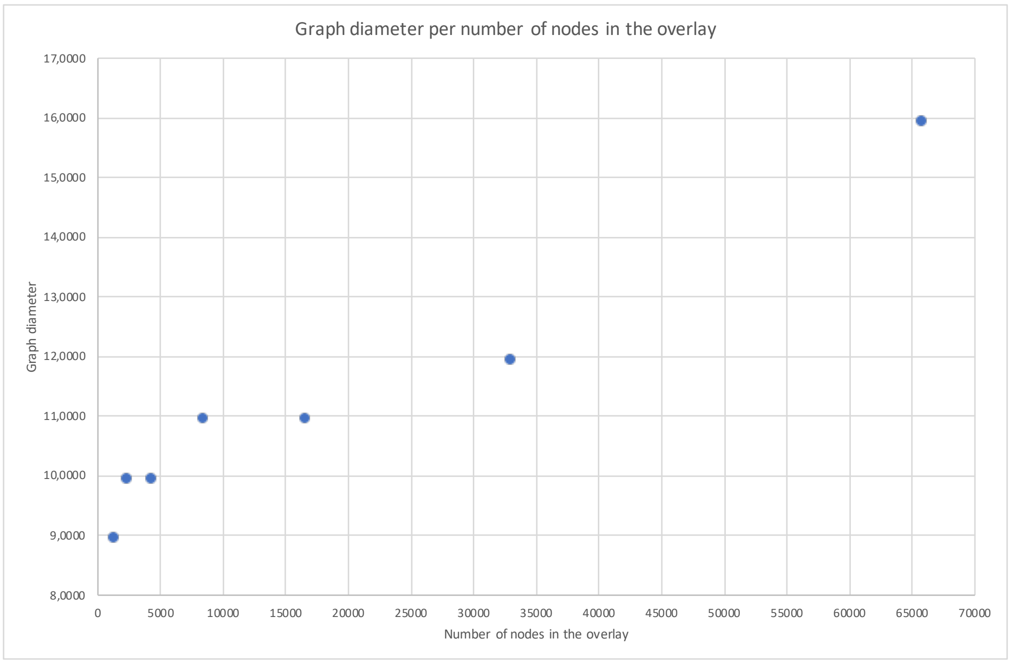
As seen on the graph, there's a linear correlation between the number of nodes and the number of endnodes, with the latter being around the 35% of the former. This is a rather low value that may be caused by an uneven distribution of the nodes on the ring, as seen over the next plots.

3.1.3 Number of endnodes w.r.t. number of queries



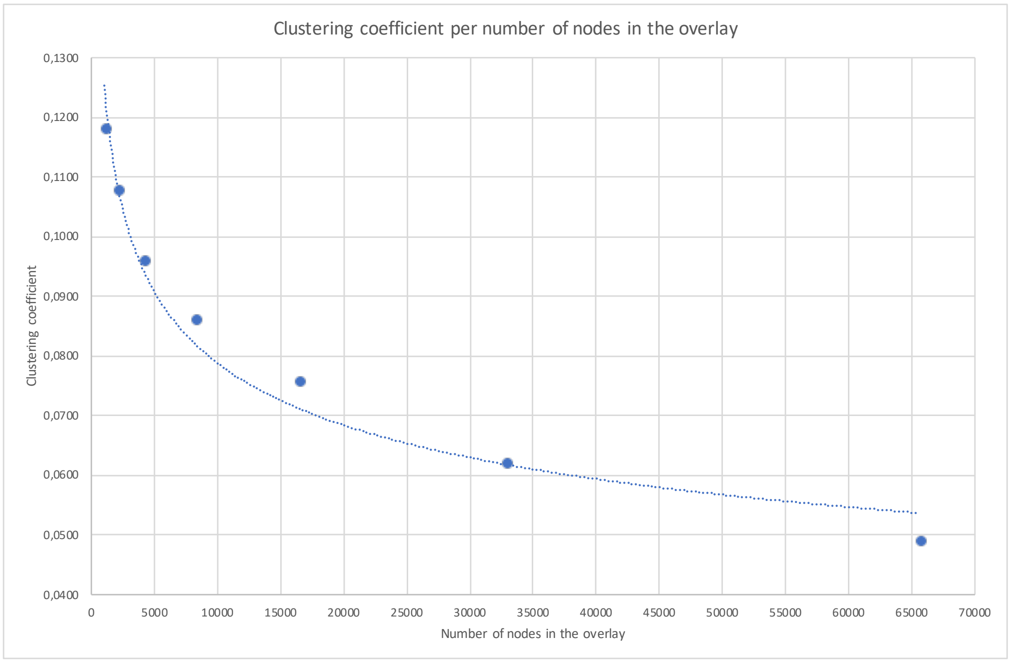
Unlike the previous plot, this one shows how the number of endnodes changes when the number of nodes is fixed (in this case $m = 16384$) and the number of simulated queries change: from $\frac{m}{2}$ (8192) to $32m$ (524288). The plot underlines the bad distribution of endnodes in the simulation; it needs at least $8m$ (131072) queries in order to reach at least 90% of the nodes in the ring.

3.1.4 Graph diameter w.r.t. number of nodes in the overlay



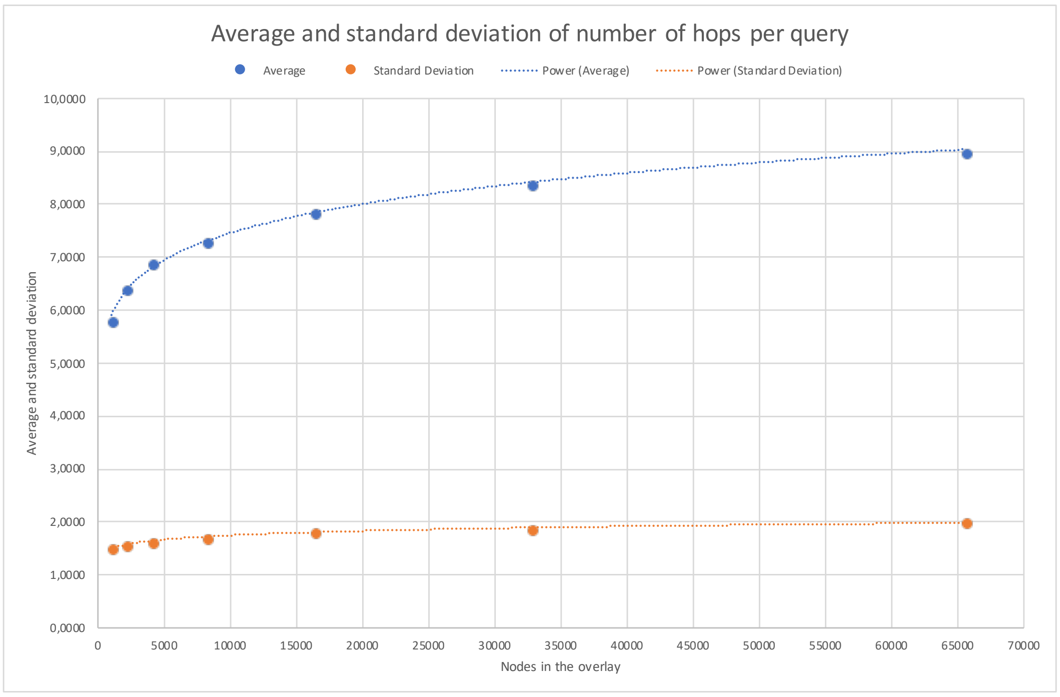
The plot for the graph diameter (length of the longest shortest path between pairs of nodes) shows that a Chord overlay network is fast to traverse: even with 2^{16} nodes in the overlay, a query wouldn't have more than 16 hops.

3.1.5 Clustering coefficient w.r.t number of nodes in the overlay



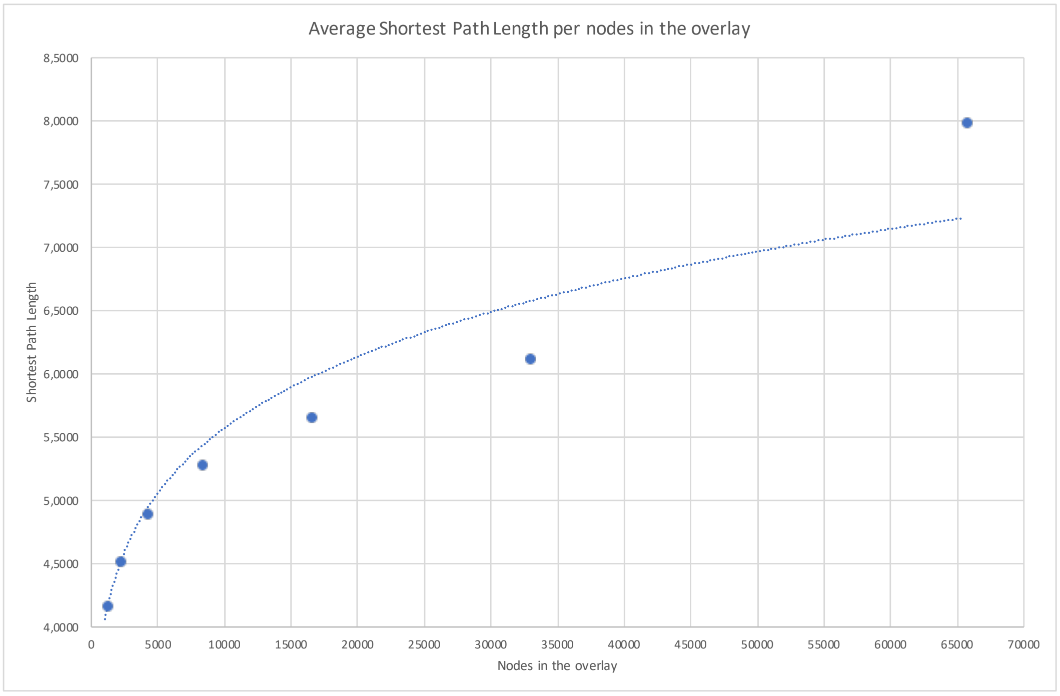
The clustering coefficient is decreasing with the increase of the number of nodes as expected, but for any m it's actually lower than the predicted value $\frac{2}{\log_2 m}$. For example, for $m = 1024$ the predicted value was $\frac{2}{\log_2 1024} = 0.2$, but the actual obtained value was less than 0.12. This can also be attributed to a poor uniformity of nodes in the ring.

3.1.6 Average and standard deviation of number of hops per query w.r.t number of nodes in the overlay



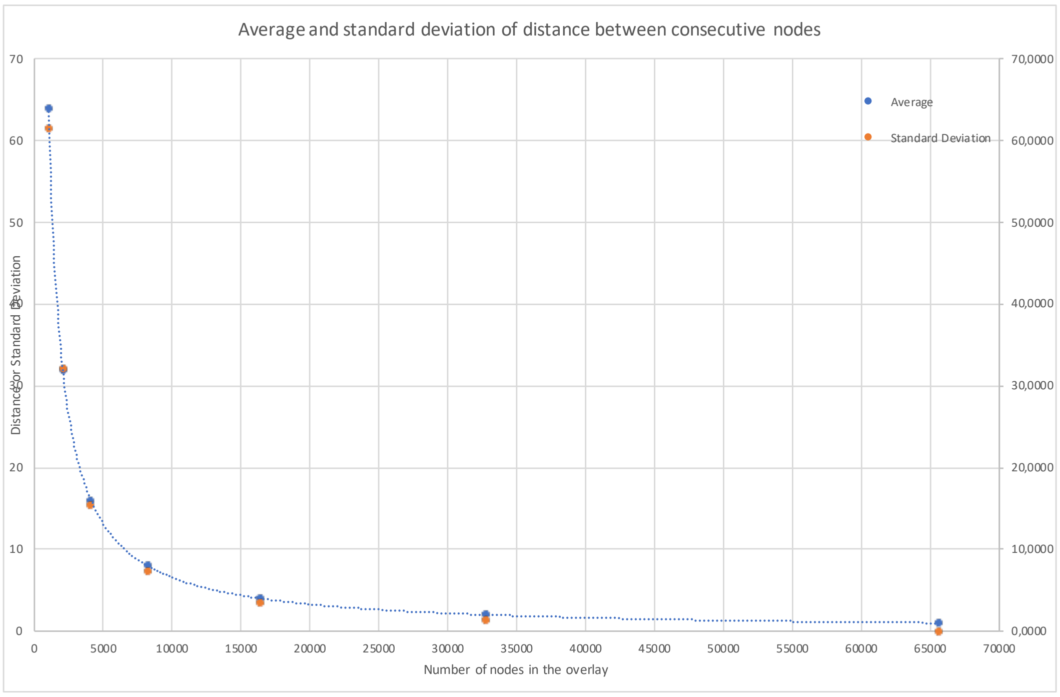
For what concerns the number of hops per query, the average increases of a small amount with the increase of number of nodes, while the standard deviation remains substantially unchanged. This shows the robustness of a Chord overlay when nodes join.

3.1.7 Average shortest path length w.r.t. number of nodes in the overlay



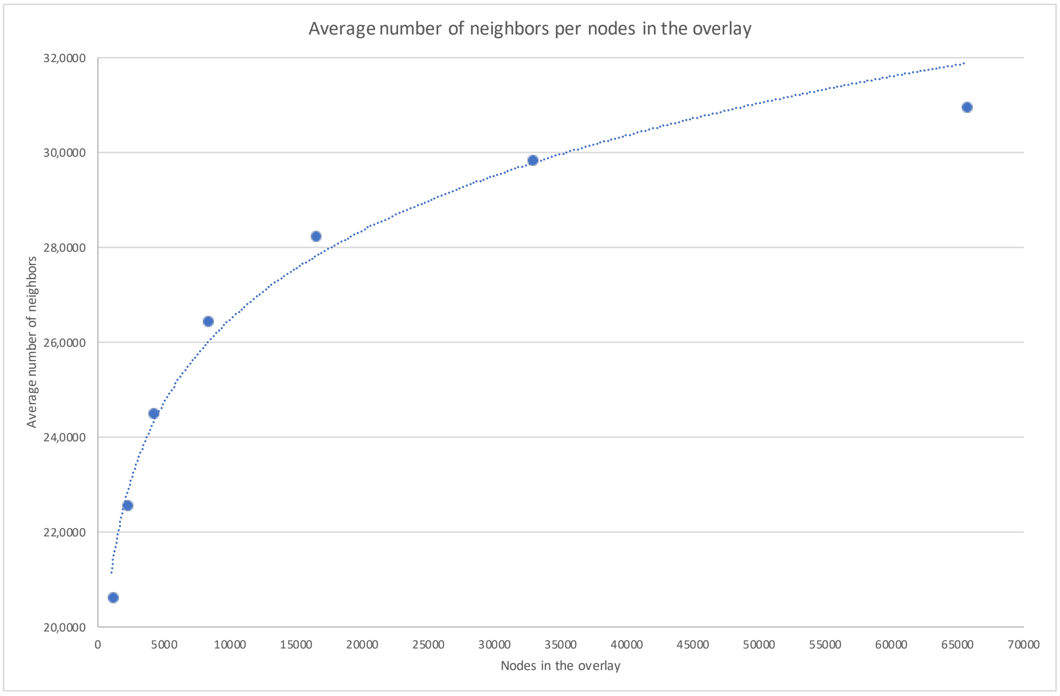
As was the case with the previous plot, also this one shows that the average shortest path grows in a minimal way when the number of nodes increase.

3.1.8 Average and standard deviation of distance between consecutive nodes, w.r.t number of nodes in the overlay



The average distance between consecutive nodes is also the average number of unique keys managed by each node; as such, is always $\frac{2^n}{m}$. The standard deviation is more interesting, since for any m is almost equal to the average distance.

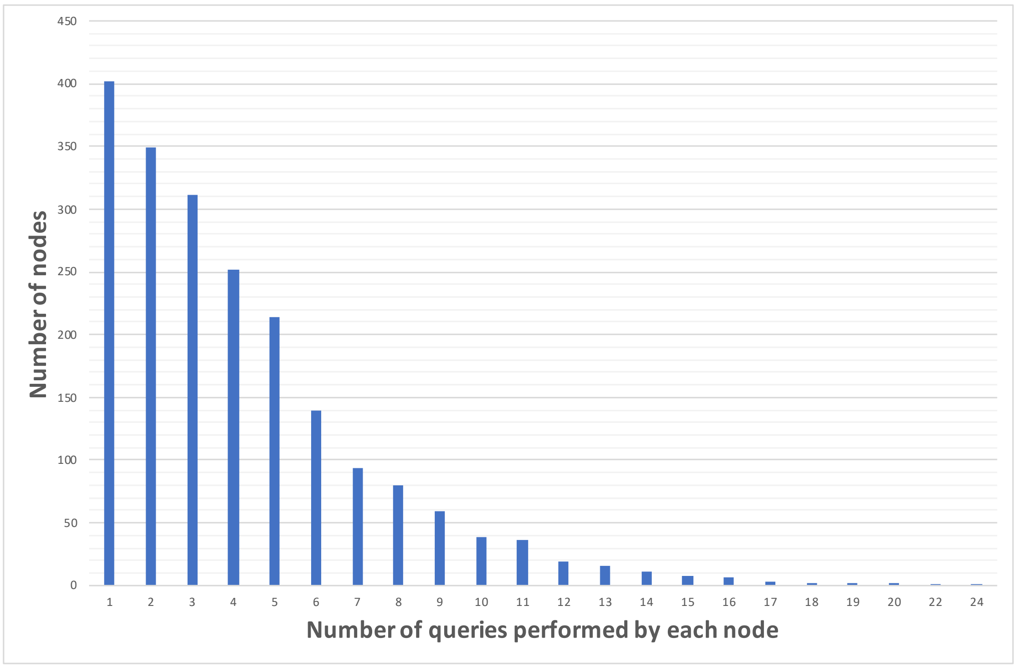
3.1.9 Average number of neighbors w.r.t number of nodes in the overlay



This plot shows the average number of neighbors (for both incoming and outgoing edges) in relation with the number of nodes in the overlay.

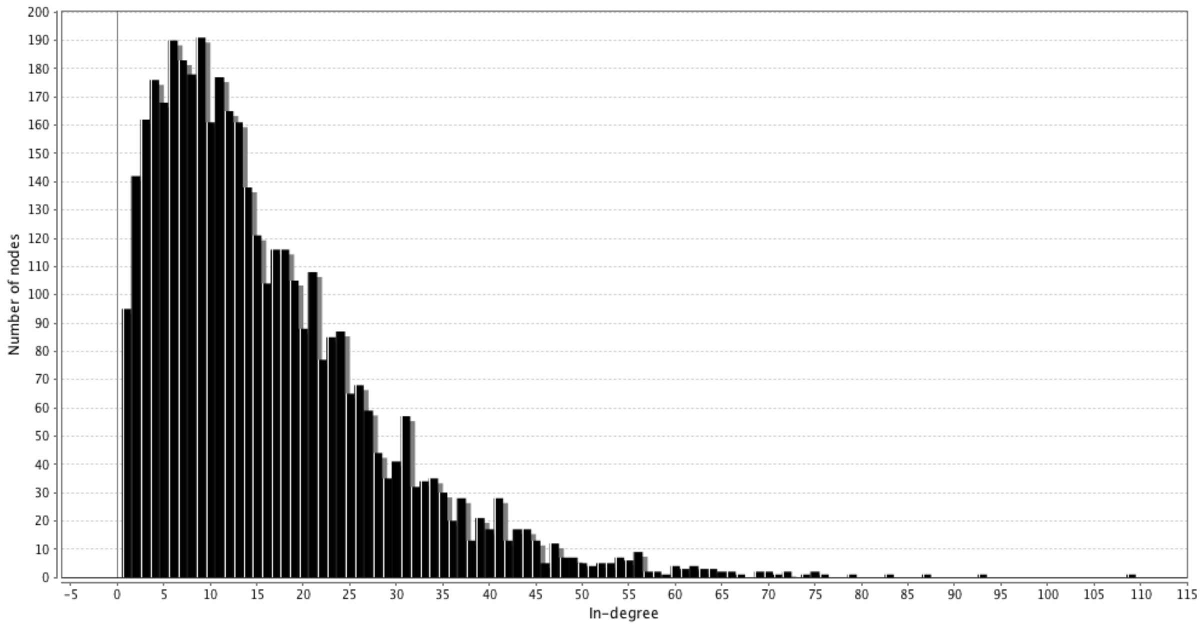
3.2 Results for $m = 4096$

3.2.1 Number of nodes that perform a certain number of queries



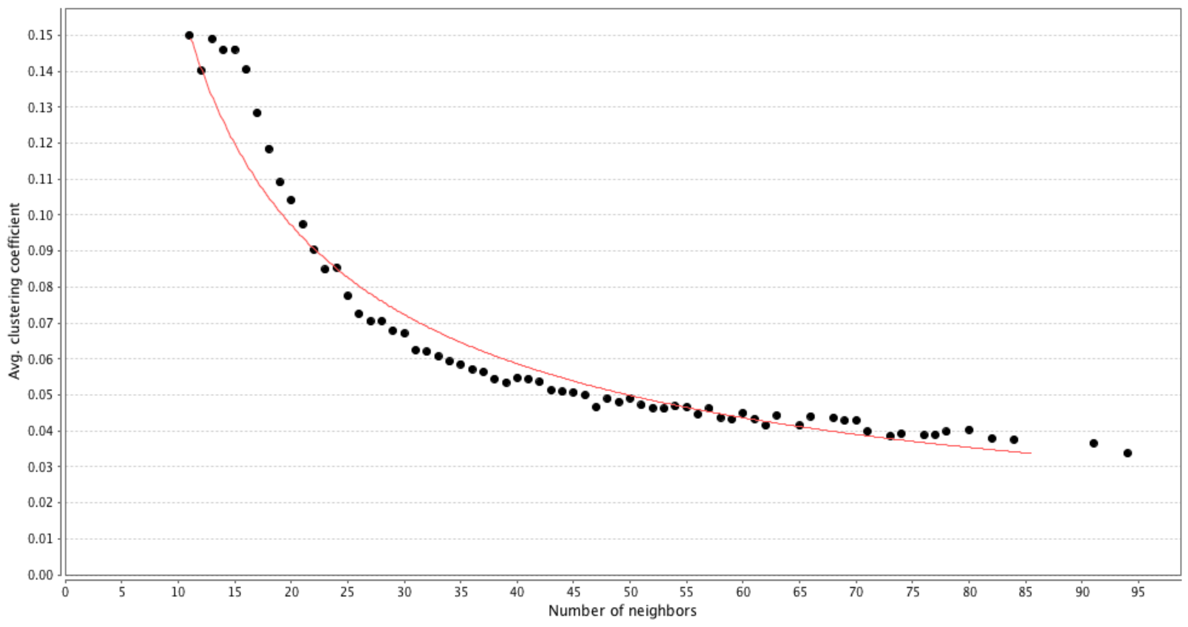
For the purpose of this plot, a query q is *performed* by node n if the method `n.lookup()` is called during q. In other words, this plot shows that ~400 nodes perform only 1 lookup, ~350 nodes perform 2 lookup, and so on.

3.2.2 Distribution of the nodes' indegree



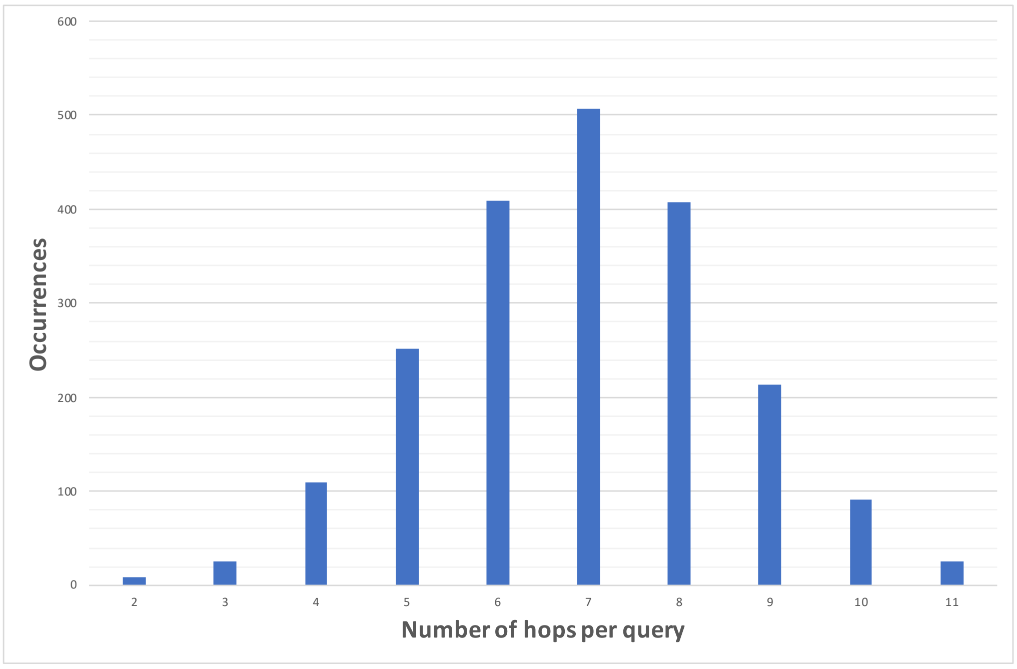
This plot shows how many nodes have a certain number of incoming fingers.

3.2.3 Distribution of the local clustering coefficient



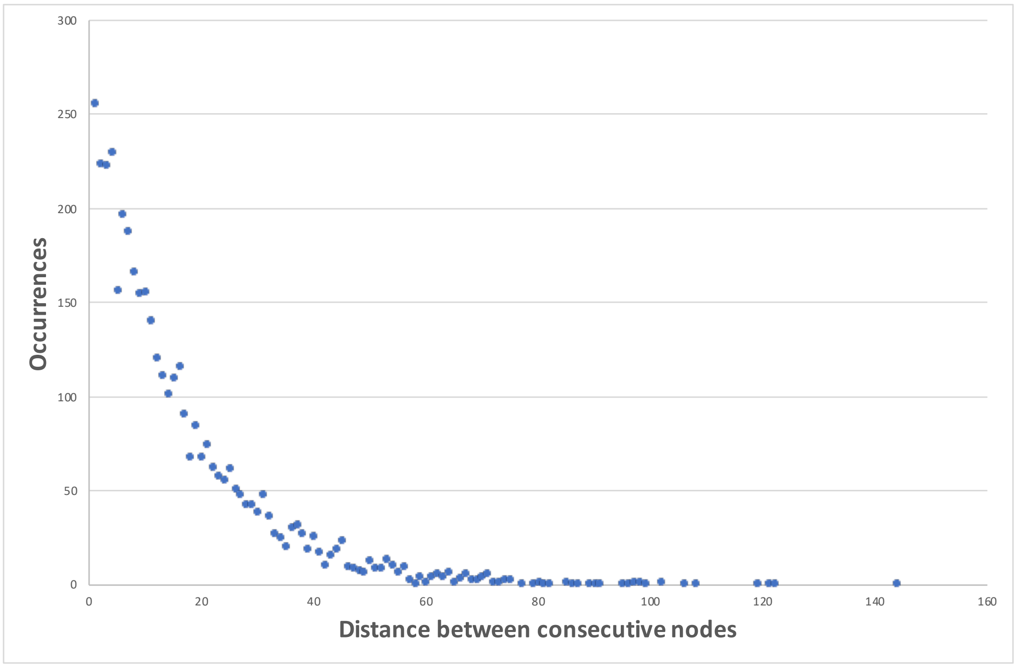
This plot shows how the clustering coefficient for a node varies depending on the number of neighbors of that node.

3.2.4 Distribution of query lengths



This plot shows the number of occurrences of a certain query length (in number of hops). Note that the measured average query length is 6,9106, which is consistent with the plot.

3.2.5 Distribution of distance between consecutive nodes



This plot shows the distribution of distance between consecutive nodes. In order for the nodes to be uniformly distributed over the ring, the distribution ought to be a Gaussian with the peak around the average distance (in this case 16); the plot shows that this isn't the case, and that there's a strong unbalance towards shorter distances.