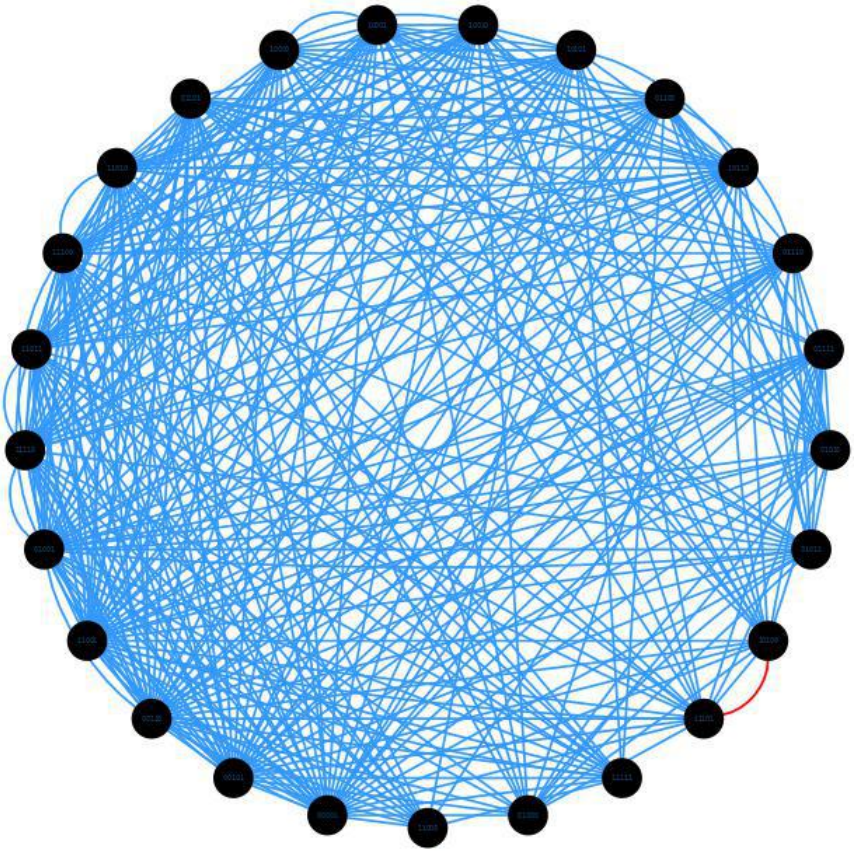
Analysing the Kademlia DHT

Andrea Bongiorno

April 30, 2019



Contents

1 Introduction

The assignment requires to implement a small portion of Kademlia protocol and analyse the networks obtained varying some parameters. In this report are described the main design choices in Section 2, then in Section 3 is described how different parameters affect Kademlia networks and finally in Section4 there is a reflection on the results obtained.

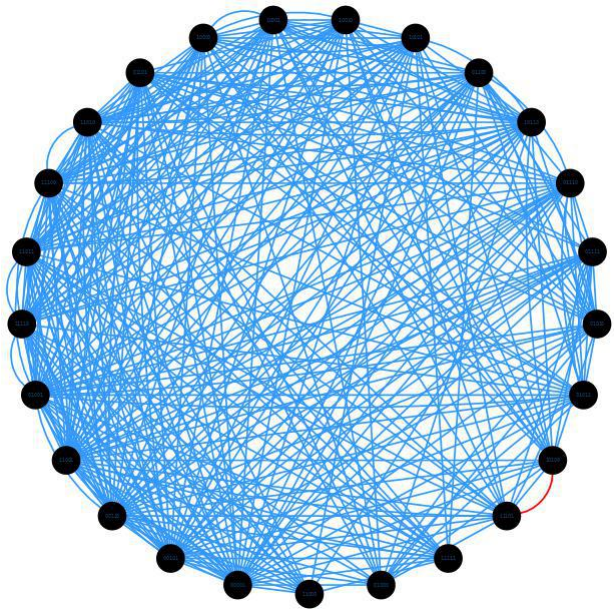


Figure 1: A little network obtained with the simulation discribed in this report.

2 Code description

2.1 Coordinator

The Coordinator class contains the main function and its duty is to generate nodes paired random identifiers (as explained in 2.3) and simulate a join to the network for each one, calling the *recursiveNodeSearch* (described in 2.5) of each node. This class contains the *main* method, that takes in input 3 parameters:

- m: number of bits of identifiers
- k: size of each bucket of nodes' routing table
- n: number of node to be generated

2.2 Node

The Node class contains all the information about a node of the network. Every node has its own routing table, represented by an hash map with $keys \in [0, m-1]$. Each key is paired with a bucket of size k which contains identifiers (precisely contains object of the class Message, described in 2.4) whose distance from the node owner of the routing table is between 2^{key} and $2^{key+1} - 1$.

2.3 Node Identifier

Node identifiers are implemented by the class NodeID, that contains also utility methods for identifiers management. As representation of identifiers I choose the BitSet class and in order to avoid negative values when calculating xor distance between two identifiers, the content of every BitSet is interpreted as an unsigned representation instead of a 2's complement one. There are two methods for the identifiers generation:

- NodeID.randomID(): generates a completely random identifiers using the SHA-256 of some random generated bytes and then truncating them to the first m bits, where m is the size of the identifiers. This method is used for creating IDs for nodes joining the network.
- NodeID.random(id1, distance): generates a random ID that has a xor distance from id1 in the interval $[distance, 2*distance)$ (assuming $distance$ is a power of 2) in the following way:
  1. Given $i = log_2(distance)$ and id2 the identifier to be generated, the $i-th$ bit of id2 is the opposite of $i-th$ bit of id1
  2. The leftmost $m \dots i+1$ bits are the same both for id1 and id2
  3. The rightmost $0 \dots i-1$ bits of id2 are randomly generated

  This method is used when generating random sequences of identifiers belonging to the identifiers' range paired with the different buckets of the routing tables of a joining node (it realizes point 2b of assignment).

2.4 Message

Messages are exchanged by nodes as result of the $find\_node$ procedure and are stored in the hash tables of each node. A message contains two information: the identifier to which it refer (the response of a request) and a list of identifiers, the ones that have received and forwarded the message.

2.5 Algorithm

The algorithm for the simulation of the network construction and for the routing tables update is realised mainly by the use of 2 methods of the Node class: $recursiveNodeSearch$ and $find\_node$.

2.5.1 find_node

This method is invoked by *recursiveNodeSearch* on a node *receiver*. It takes as input the identifier of the node who sent the request and the requested identifier, then if there is enough space in the right bucket the receiver inserts the sender into its routing table. The receiver returns to the sender a set of at most k identifiers and this is a requirement of Kademlia, when there are less than k identifiers in the bucket, other buckets are visited until k identifiers are found or all the routing table has been traversed.

Please note: since the routing tables' buckets contain objects of type Message, to the sender is returned not only the closer nodes to the requested one but also the list of identifiers traversed by the message that contains that information. Moreover, in some experiments I limited the number of identifiers returned, due to limited computational power.

2.5.2 recursiveNodeSearch

This method is called by the Coordinator on a node joining the network. As explained in 2.1, some identifiers paired with the range of the k-buckets are randomly generated and for each of this identifiers a call to *recursiveNodeSearch* is performed.

recursiveNodeSearch takes as input the requested identifier and a list of nodes to be contacted (initially obtained calling $find\_node$ on the bootstrap node). Given p the node executing this method and q a generic node in the list, the following happens:

1. p ask to q information about the requested identifier (it may belong to the network or not) calling $q.find\_node$
2. q returns to p a list of at most k messages, each of which contains one identifier *close* for q to the requested one and a list of nodes traversed by the message.
3. For each identifier contained in the messages, including the ones in the *traversed* lists, p call their $find\_node$ method, accumulating the results in a set
4. *recursiveNodeSearch* is recursively called on the union of the new set and the set of identifiers left from the previous call

This 4 steps are repeated for each node in the set obtained by the bootstrap node. The methods ends in two cases: when the requested node is found or when there aren't nodes to contact. Every node contacted is also inserted in the routing table (when there is enough space in the right bucket).

3 Graph analysis

Experiments for the analysis of the networks obtained from the algorithm described in this report have three parameters: size of the network n, bits of

identifiers m and size of routing tables' buckets k.

After every simulation, the Coordinator (2.1) saves the obtained network in a .csv file, used as input for the analysis tool Cytoscape.

In the rest of this section I will show how the three parameters n, k and m affect the network topology and its characteristics.

3.1 Size of identifiers

In this section is shown in which cases the size of identifiers influences the network. In particular is found one reason why Kademlia requires the $find\_node$ to return k identifiers when possibile.

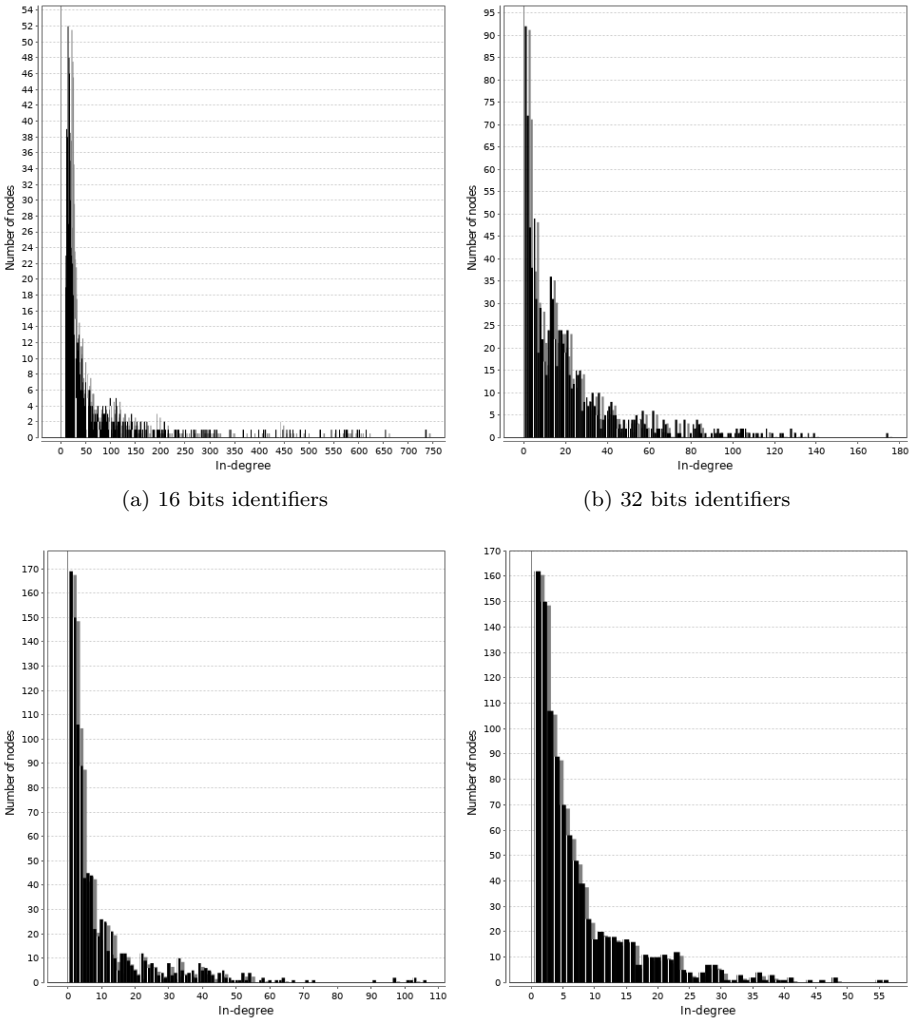
Experiments shown that once fixed the number of nodes n and the size of buckets k, the dimension of identifiers' space strongly influences the network only when is admitted for the $find\_node$ procedure to return list without a minimum of identifiers, as table 1 highlights.

This experiments were done initially because due to the limted computational power at my disposal I was not always able to run a version of the $find\_node$ returning k neighbours. In any case this experiments show the imprtance of returning always k identifiers from $find\_node$ (when possible).

m (bits)	Clustering coefficent	Diameter	Avg. path length	Edges
16	0.359	4	2.438	$\approx$ 62k
32	0.521	18	4.870	$\approx$ 22k
64	0.477	18	6.240	$\approx$ 11k
128	0.502	20	7.419	$\approx$ 8k

Table 1: Topologies parameters when varying size of identifiers. n and k fixed respectively to 1000 and 8.

It is also possible to see that while the in-degree distribution remains stable across the different model (Fig. 2), out-degree and clustering coefficient distributions degenerate (Fig. 3 and Fig. 4).



(a) 16 bits identifiers

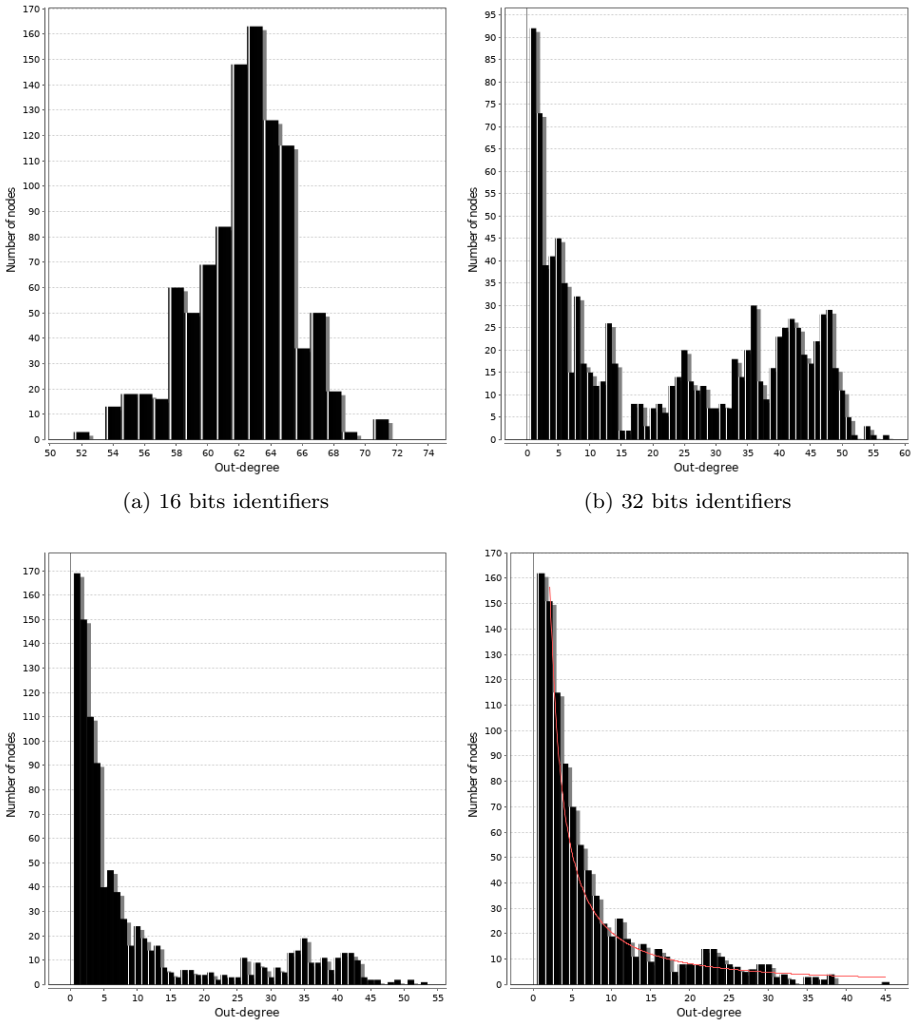
(b) 32 bits identifiers

(c) 64 bits identifiers

(d) 128 bits identifiers

Figure 2: Identifiers' space does not affect heavy tail distribution of in-degree, although the maximum in degree of a node increases as the size of identifiers increases

The out-degree distribution becomes from gaussian to exponential, this means that the buckets of the routing tables becomes more empty (less nodes with high out-degree) and even increasing their size k won't affect the sparseness of the graphs. This result could bring to more expensive queries, as shown by diameter and avg. path length obtained.



(a) 16 bits identifiers

(b) 32 bits identifiers

(c) 64 bits identifiers

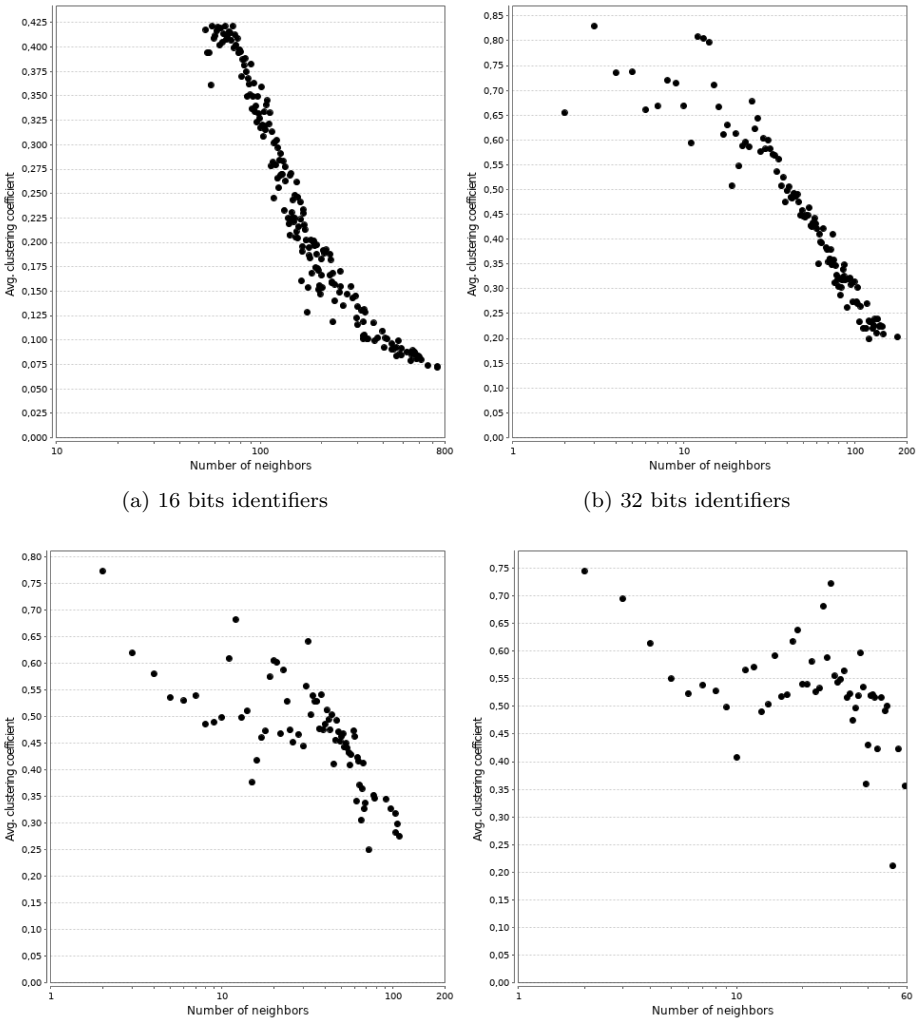
(d) 128 bits identifiers

Figure 3: Identifiers' space affect normal distribution of out-degree

A local analysis of clustering coefficient distribution can tell if the studied network is organised according to some rules or if its connection are randomly generated.

Clustering coefficient of a node tell us in which measure the neighbours of that node tend to form a clique, and here is what we can understand from Fig. 4:

- In the good case (Fig. 4a) the clustering coefficient of each node tends to regularly decrease as the number of neighbours increases. This lead to a structure with one giant (connected) component.
- In the other three cases the clustering coefficient distribution tends to degenerate: the values are greater on average and doesn't go under a certain threshold. This distribution is symptom of a graph structure where we can identify different clusters, each of which with a low diameter but almost isolated between each other.



(a) 16 bits identifiers

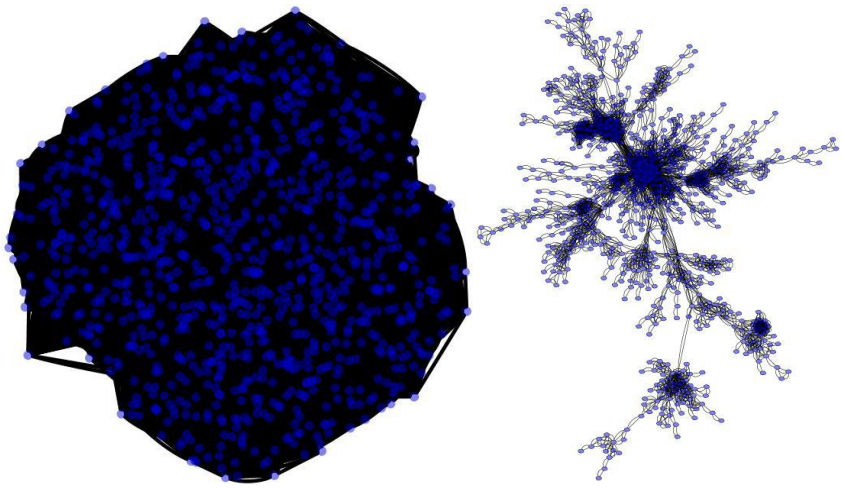
(b) 32 bits identifiers

(c) 64 bits identifiers

(d) 128 bits identifiers

Figure 4: ....

Figure 5 shows the two opposite cases: one single giant component in the denser network (16 bits identifiers) and several "micro" clusters in the sparser one (128 bits identifiers).



(a) 16 bits identifiers (b) 128 bits identifiers

Figure 5: Different clustering influenced by identifiers' space

The reason behind the impact of identifiers' space is to be found in the fact that the Coordinator is more likely (statistically speaking, and when n is sufficiently small) to generate an identifier at a distance such that the bootstrap node doesn't know anyone close to it and then the calls to "$bootstrap.find\_node$" returns almost empty lists.

The influence of the size of identifiers' space is completely avoided by real implementation of Kademlia, because the protocol as already said requires $find\_node$ to return k identifiers or the entire content of the table if there are less.

In particular, experiments shown that even returning $k/2$ neighbours, the size of identifiers is almost irrelevant as shown in table 2.

m (bits)	Clustering coefficient	Diameter	Avg. path length	Edges
16	0.359	4	2.438	$\approx$ 62k
32	0.378	4	2.341	$\approx$ 63k
64	0.392	4	2.351	$\approx$ 65k
128	0.375	4	2.367	$\approx$ 61k

Table 2: Topologies parameters when varying size of identifiers. n and k fixed respectively to 1000 and 8. Every $find\_node$ returns k values when possible.

From now on the $find\_node$ procedure will return k neighbours if not differently specified.

3.2 Size of network and buckets

As one can think, the number of nodes in the network and the required size of the routing tables are parameters closely related: if a network is large, a larger routing table is required in order to obtain a graph with low diameter and high (0.3-0.5) clustering degree. In Kademlia the maximum size of nodes' routing tables is determined by $m \cdot k$, recalling that m is the number of bits of an identifier and k is the size of each bucket of the routing table.

The experiments on this two parameters show one important characteristic of Kademlia DHT: even in a large network a small routing table for each node is sufficient to achieve low diameter and high clustering.

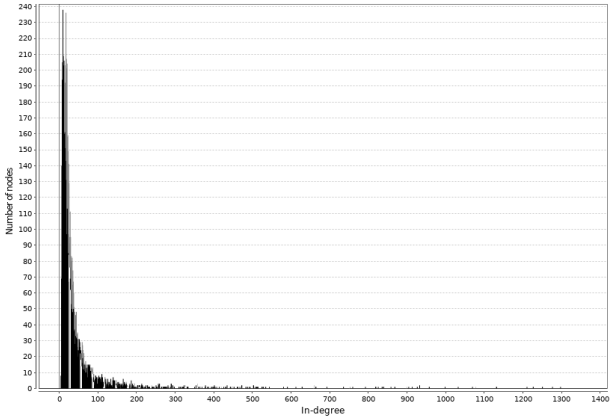
Table 3 summarise the values obtained with $k = 5$ (fixing k is realistic as it is a common practice, in real Kademlia implementation $k \approx 20$) and $m = 15$, so the maximum size of the routing tables is 75 elements.

n	Clustering coefficient	Diameter	Avg. path length
1000	0.309	5	2.7
5000	0.252	6	3.167
10.000	0.253	6	3.374
20.000	0.246	6	3.569

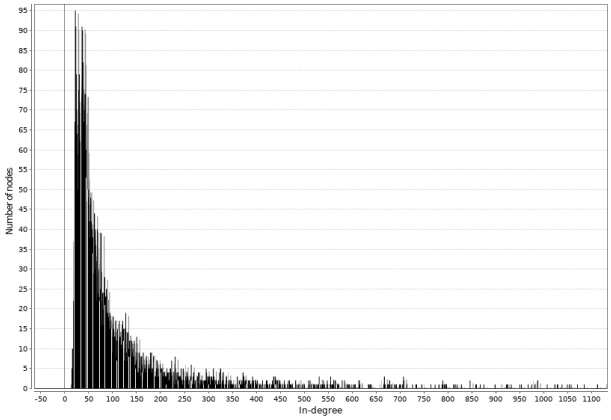
Table 3: Kademlia protocol is tolerant to network' s dimension scaling with low space complexity

The result obtained are slightly better than the theoretical one, in particular the diameter is always less than $log(n)$. This results are justifiable by the fact that, since messages keep information about nodes traversed, every node has more chance to update its routing table.

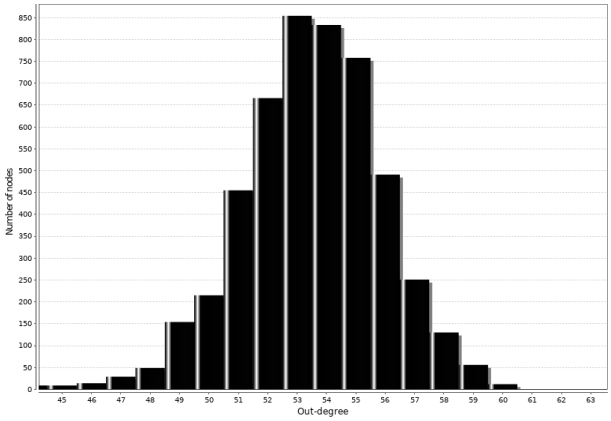
From figure 6 is possibile to see how the in degree and out degree change when k increases, n is fixed to 5000.



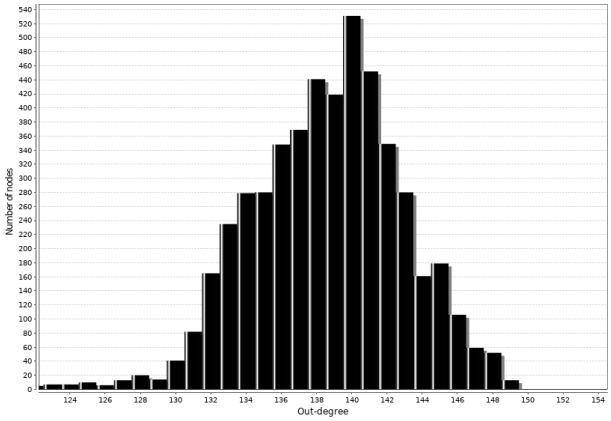
(a) In degree with $k = 5$



(b) In degree with $k = 15$



(c) Out degree with $k = 5$



(d) Out degree with $k = 15$

Figure 6: In and out degree varying k

As expected the out degree increases on average when k increases, this is obviously explained by the fact that each bucket can contain more neighbours. For the in degree is possibile to notice a slight improvement of the distribution when k increases, and the maximum has been more than halved (90 versus 240). This result in a more balanced network and a less overheaf for nodes with higher in degree.

4 Conclusions

Experiments described in this report are dimensionally small, due to the limited computational power at my disposal but they are sufficient to have a rough but correct idea about the Kademlia network behaviour.

Analysing the results obtained it is possible to state that:

- Size of identifiers m is irrelevant for the network topology in real implementation of kademlia, where the $find\_node$ method return always a minimum number of identifiers. The number of returned identifiers can be a linearly smaller than k as shown by Table 2. The bad distribution of identifiers in the identifier space can result in unbalanced routing tables and expensive routing process, this is a known weakness of Kademlia protocol.

- Keeping track of nodes traversed by every message could improve performance of real implementation of Kademlia, but may be necessary to limit the size of the list in order to limit space complexity.

- The simulation, in the cases of "good" networks, shows another weakness of Kademlia: since the distribution of in degree can be described by an exponential curve with heavy tail (as expected), this can introduce load balancing problems for the nodes with higher degree that have to manage an high number of messages (this issue is shared with chord DHT). This overhead can be mitigated by increasing the size of buckets, as shown by the last experiment.

- Thanks to its low memory overhead, Kademlia may be a suitable solution for IoT networks.