Evaluating Kademlia performances

Andrea Bruno
Department of Computer Science
University of Pisa

Abstract. The growing interest in overlays networks (mostly because of cryptocurrencies) lead us to study the underlying technology both to understand future trends and to discover possible issues. In this paper we describe an analysis of Kademlia protocol, with special attention to its lookup procedure. First, we created the infrastructure, and then we did an exhaustive study of the performance according to the directed graphs obtained after the experiments. An appropriate set of parameters have been tested based on both literature and real use cases.

1 Introduction

According to recent studies[1], file sharing is consuming about 7% of global bandwidth and this trend will be stable[2] for the next years, but if we consider also IoT devices[3] and blockchain-based applications[4] (who widely use P2P technologies), these numbers determine a very rough estimate of the real future bandwidth consumption. Due to the large usage of P2P networks, there is a need to understand the performance of this technology through a comprehensive study, in order to reduce the waste of resources globally. The success of these network is behind the scenes, in fact, the Distributed Hash Table has been the real disruptive innovation. These algorithms (e.g. CAN[5], Pastry[6], Chord[7]) offer an elegant solution for an efficient decentralized data mapping. Although Chord promised the best theoretical properties, Kademlia[8] has become the de facto standard, in fact both BitTorrent[9] and Ethereum [10][11] (which manage more than a million nodes) are based on Kademlia. As the original paper suggests, Kademlia assigns each node an id chosen uniformly at random from $\{0,1\}^d$, where d is usually 128[12] or 160[13]. Hence, we always refer to a node by its ID. Given two IDs:

$$x = (x_1, x_2, \dots, x_d) \quad \text{and} \qquad y = (y_1, y_2, \dots, y_d)$$

Kademlia defines their XOR distance by:

$$\delta(x,y) = \sum_{i=1}^{d} (x_i \oplus y_i) \times 2^{d-i}$$

Where the $\oplus$ denotes the XOR operation:

$$a \oplus b = \begin{cases} 1 & if \;\; a \neq b \\ 0 & otherwise \end{cases}$$

In this work, when we talk about distance and closeness, we always mean XOR distances between IDs. Roughly speaking, a Kademlia node keeps a table of a few other nodes (neighbours) whose distances are sufficiently diverse. So, when a node searches for an ID, it always has some neighbours close to its target. By inquiring these neighbours, and these neighbours' neighbours, and so on, the node that is closest to the target ID in the network will be found eventually. Other DHTs work in similar ways. The differences mainly come from how distance is defined and how neighbours are chosen. For a more detailed survey of DHTs, see [14].

2 System Model

In this work, the system has been split into three main logical parts, based on their functionalities. The first one, called *Network* (Figure 1)*,* is the core of the system indeed it contains all the classes that in the real world are either physical devices or logical instances. The design choice that stands out immediately, is the separation of the node logic from its own representation. In fact, the class *Peer* represents what in the original paper is called "contact", that basically is the tuple *<ID, IP Address, UDP Port>*, whereas the *Node* class, englobes both an object *Peer* and a *Routing Table*. To represent the physical network of the real world, a class called *Kademlia* has been added to the package *Network*. This way, every new *Node* can ask a bootstrap *Node* to a *Kademlia* object, in order to start communicating and thus join the network.

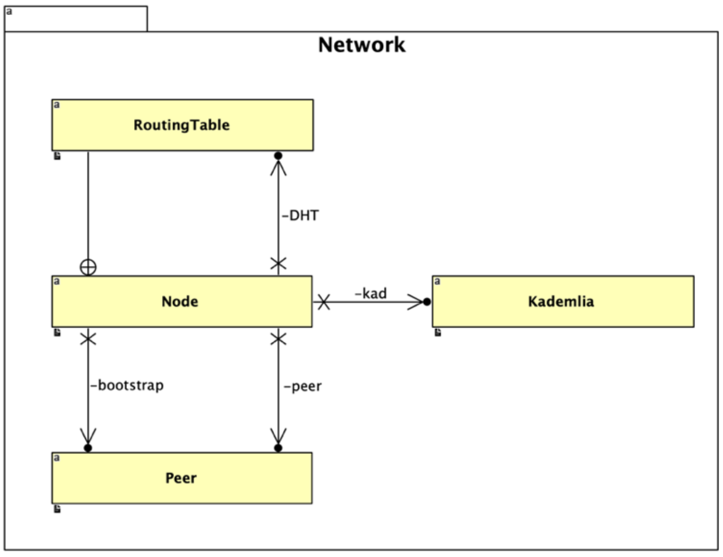


Figure 1: UML diagram of Network package

The second package called *Utility* contains all the procedures useful to the creation of a Peer, such as a random IP, a random Port and a random ID. Although Kademlia creates IDs based on a 160-bit SHA-1[15], in this work, since we were asked to test the reaction of the protocol by varying the number of the bits, these hash algorithms have been put aside.

Finally, the package *Start* executes the experiments by varying the number of the nodes connected, the number of the *k-buckets* and the number of bits. In other words, is what in the requirements is called "coordinator".

3 Protocol

In Kademlia each node is identified by a numerical ID with fixed bit-length b. These IDs are generated using hash functions with the goal of equal distribution of the identifiers inside the identifiers space. Each node maintains a routing table with the IDs of other nodes called *contacts*[1]. The routing table consists of b so-called *k-buckets* to store the contacts of the node. The buckets are indexed from 0 to $b-1$, and the contacts are distributed into these buckets depending on the XOR distance[2]. The buckets are populated with those contacts id_x fulfilling the condition

$$2^i \leq dist(id, id_x) < 2^{i+1}$$

with i being the bucket index. This means that the bucket with the highest index covers half of the ID space, the next lower bucket a quarter of the ID space, and so on. The maximum number of contacts stored in one bucket is k. Next to b and k, another defining property of a Kademlia setup is the request parallelism α, which determines how many contacts are queried in parallel when a node wants to locate another node. The Kademlia authors set the default values $b = 160$, $k = 20$, $\alpha = 3$. The nodes of a Kademlia network can locate other nodes by their identifiers. Given a target identifier, a node queries α nodes from its routing table closest to that identifier. Those, in turn, answer with their own list of closest nodes, which can then be used in new queries. This way, the requesting node iteratively gets closer to the target identifier. This process ends when no more progress is made in getting closer to the target identifier.

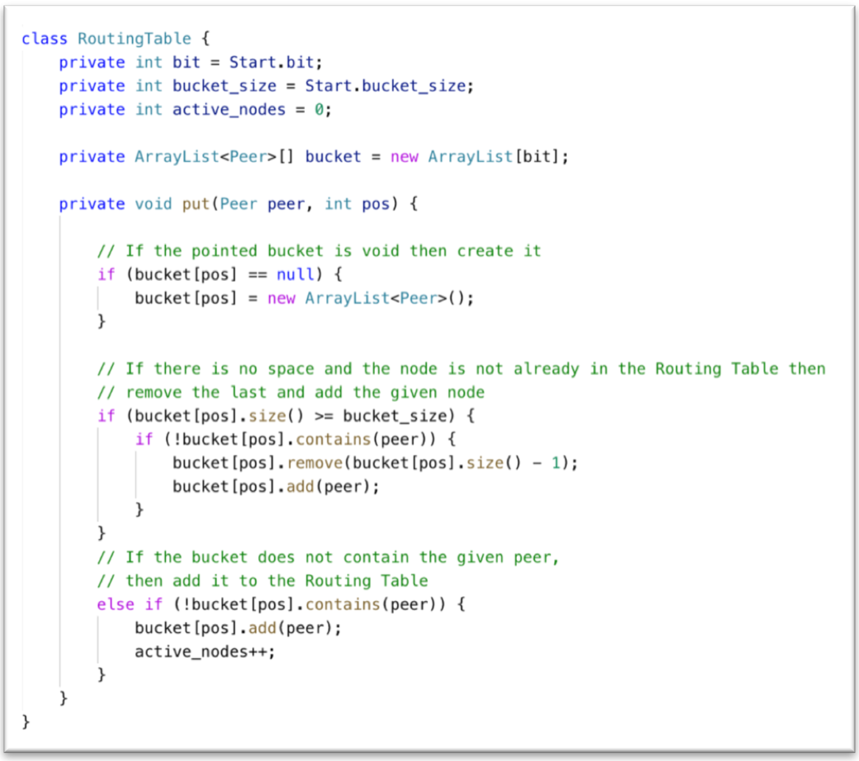
3.1 Node construction

Before talking about the design choice of the network, there is a need to explain how a new node is being created. As described in Figure 1, every *Node* is composed by a *Peer* object. Inside the constructor of the *Peer* class, there is the generation of a random tuple *<ID, IP Address, UDP Port>*. Since we did not use any hash function and since we don't want collisions between IDs, there is a static

[1] In this work, the term *contact* has been replaced by the term *peer.*

[2] As described in the introduction.

HashSet[16] that takes care of remembering all the IDs previously generated. This way, by using a simple do-while loop, we guarantee collisions avoidance. Another fundamental aspect is the *Routing Table* (Figure 2) inside each node. Following the guidelines of the original paper, we create *m-bit* buckets (by exploiting Java *ArrayLists[17]*), each capable of storing *k* elements. Note that in order to save up memory, instead of saving *Node* objects, we keep track only of the *Peer* component.



```java
class RoutingTable {
    private int bit = Start.bit;
    private int bucket_size = Start.bucket_size;
    private int active_nodes = 0;

    private ArrayList<Peer>[] bucket = new ArrayList[bit];

    private void put(Peer peer, int pos) {

        // If the pointed bucket is void then create it
        if (bucket[pos] == null) {
            bucket[pos] = new ArrayList<Peer>();
        }

        // If there is no space and the node is not already in the Routing Table then
        // remove the last and add the given node
        if (bucket[pos].size() >= bucket_size) {
            if (!bucket[pos].contains(peer)) {
                bucket[pos].remove(bucket[pos].size() - 1);
                bucket[pos].add(peer);
            }
        }
        // If the bucket does not contain the given peer,
        // then add it to the Routing Table
        else if (!bucket[pos].contains(peer)) {
            bucket[pos].add(peer);
            active_nodes++;
        }
    }
}
```

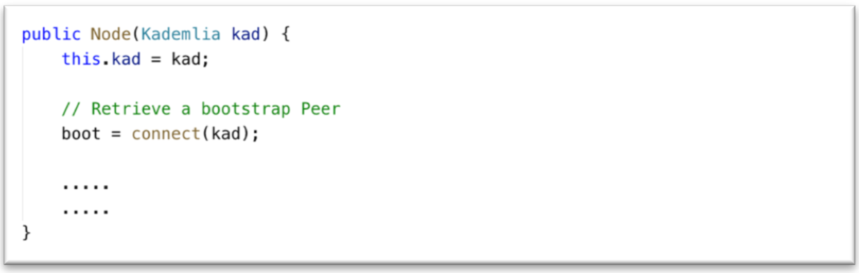
Figure 2: RoutingTable class code

3.2 Initialization phase

The first phase of Kademlia protocol is the bootstrap node creation. This phase is fundamental because it is done only once during the entire life of the network. The main idea is to connect to Kademlia a node containing a void routing table. In this work, to maintain a consistent view of what is happening, the network stores information about every node that connects to Kademlia. Hence, in order to complete this phase, the node described above is merely added to the internal data structure of Kademlia.

3.3 Routing table construction

The crucial points of Kademlia are both the connection and the lookup procedure. The former has been implemented in a very easy way. Basically, the constructor of the *Node* class requires as input (Figure 3), an object of type *Kademlia*, this way it can retrieve a bootstrap node useful to start communicating with the network.



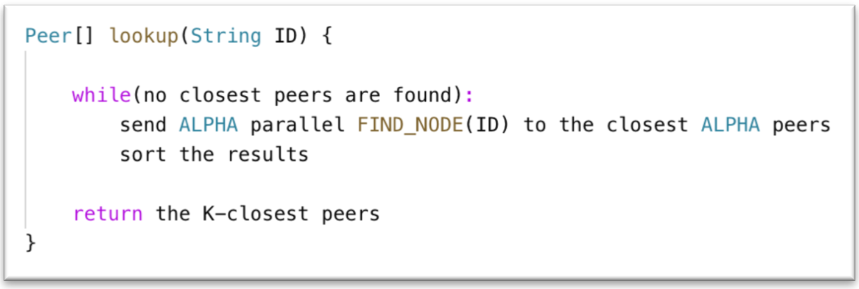
```
public Node(Kademlia kad) {
    this.kad = kad;

    // Retrieve a bootstrap Peer
    boot = connect(kad);

    .....
    .....
}
```

Figure 3: A partial view of the Node class

Afterwards, the protocol establishes to start a lookup procedure, having as input argument, the ID of the node just created. Such procedure is quite complex, but we can summarize it in few steps as follows (Figure 4):



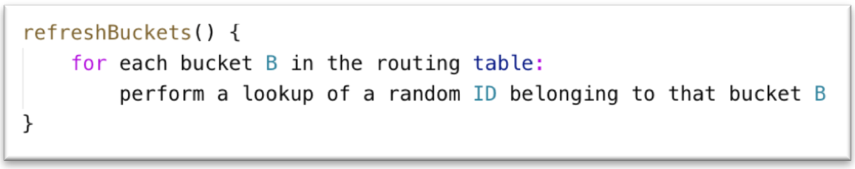
```
Peer[] lookup(String ID) {

    while(no closest peers are found):
        send ALPHA parallel FIND_NODE(ID) to the closest ALPHA peers
        sort the results

    return the K-closest peers
}
```

Figure 4: The pseudo-code of the lookup procedure

At the end of the lookup procedure, as the literature suggests, there are two possibilities. The first used in Ethereum[11], recommend to lookup a random peer having an ID closer to the ID of the bootstrap node. The second instead, suggests a "buckets refresh" that we can sum up as follows (Figure 5):



```
refreshBuckets() {
    for each bucket B in the routing table:
        perform a lookup of a random ID belonging to that bucket B
}
```

Figure 5: The "buckets refresh" pseudo-code

4 Simulation

In this section, we describe the simulation environment and the tools used to make the experiments. Afterward, we present the parameters used during the test.

4.1 Environment

For our simulations, we have been using an Amazon Web Services EC2 instance (more precisely one m5.xlarge[18]), that is a virtual machine running on four vCPU of an Intel Xeon Platinum 8000 series, capable of delivering up to 3.1 GHz in Turbo Boost[19].

As far as the code concerned, we have been using Java[20] both to simulate the networks creation and to generate the graphs, whereas, the graph analysis part has been done using MATLAB[21] through NetworkX[22] library.

4.2 Parameters

We were asked to test Kademlia by varying *m* (number of bits), *n* (number of nodes) and *k* (size of the routing tables). To get a concise view of the results, we performed 512 tests[3] by applying the following sets of parameters (Figure 6):

m	20	40	60	80	100	120	140	160
n	625	1250	1875	2500	3125	3750	4375	5000
k	2	5	7	10	12	15	17	20

Figure 6: List of parameters tested in this work

More precisely, for each possible tuple <*m, n, k*>, we created a new *Kademlia* object containing *n* nodes, each having a routing table composed by *m* buckets of size *k*.

[3] Each set of parameters was composed by 8 values. Hence, since we tested 3 parameters, $8^3 = 512$ tests.

5 Metrics

In this section we describe our evaluation metrics used along all the simulations.

5.1 Average degree

The degree *k* of a node is the number of edges connected to it. Hence, the average degree is equal to

$$\langle k \rangle = \frac{E}{N}$$

where *E* is the total number of edges and *N* is the total number of nodes.

5.2 Average shortest path

The average shortest path length is

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$$

where *V* is the set of nodes in *G* , *d(s, t)* is the shortest path from *s* to *t*, and *n* is the number of nodes in *G*.

5.3 Diameter

The *diameter d* of a graph is the maximum *eccentricity* [4] of any vertex in the graph. That is, *d* is the greatest distance between any pair of vertices or, alternatively is, $d = max_{v \in V}\, \epsilon(v)$. To find the diameter of a graph, first find the shortest path between each pair of vertices. The greatest length of any of these paths is the diameter of the graph.

[4] The eccentricity $\epsilon(v)$ of a vertex *v* is the greatest distance between *v* and any other vertex; in symbols that is $\epsilon(v) = max_{u \in V}\, d(v,u)$. It measures how far a node is from the node most distant from it in the graph.

5.4 Clustering coefficient

The local clustering coefficient of a node n_i in a graph *G* is:

$$c_i = \frac{y_i}{\binom{d_i}{2}}$$

Where y_i is the number of links between neighbours of n_i , and d_i is the degree of the node n_i.

The clustering coefficient *C* of the whole graph *G* is the average of the local clustering coefficients for all the nodes in *G*,

$$C = \frac{1}{N} \sum_{i \in N} c_i$$

This number is precisely the probability that two neighbours of a node are neighbours themselves. This is equal to 1 on a fully connected graph (everyone knows everyone else).

5.5 RPC Depth

This ad-hoc metric counts the number of recursive calls performed by the lookup procedure of Kademlia protocol.

6 Results

# Nodes	Average Time (ms)
625	171.47
1250	344.33
1875	555.08
2500	778.86
3125	1027.50
3750	1337.97
4375	1603.78
5000	1913.20

Figure 7: Average time in milliseconds to complete a simulation having a fixed number of nodes and a variable number of buckets and bits.

K Buckets	Average number of edges
2	15915.06
5	40797.39
7	54918.05
10	74598.48
12	86676.95
15	103283.55
17	113820.78
20	128294

Figure 8: Average number of edges having a fixed number of buckets and a variable number of nodes and bits.

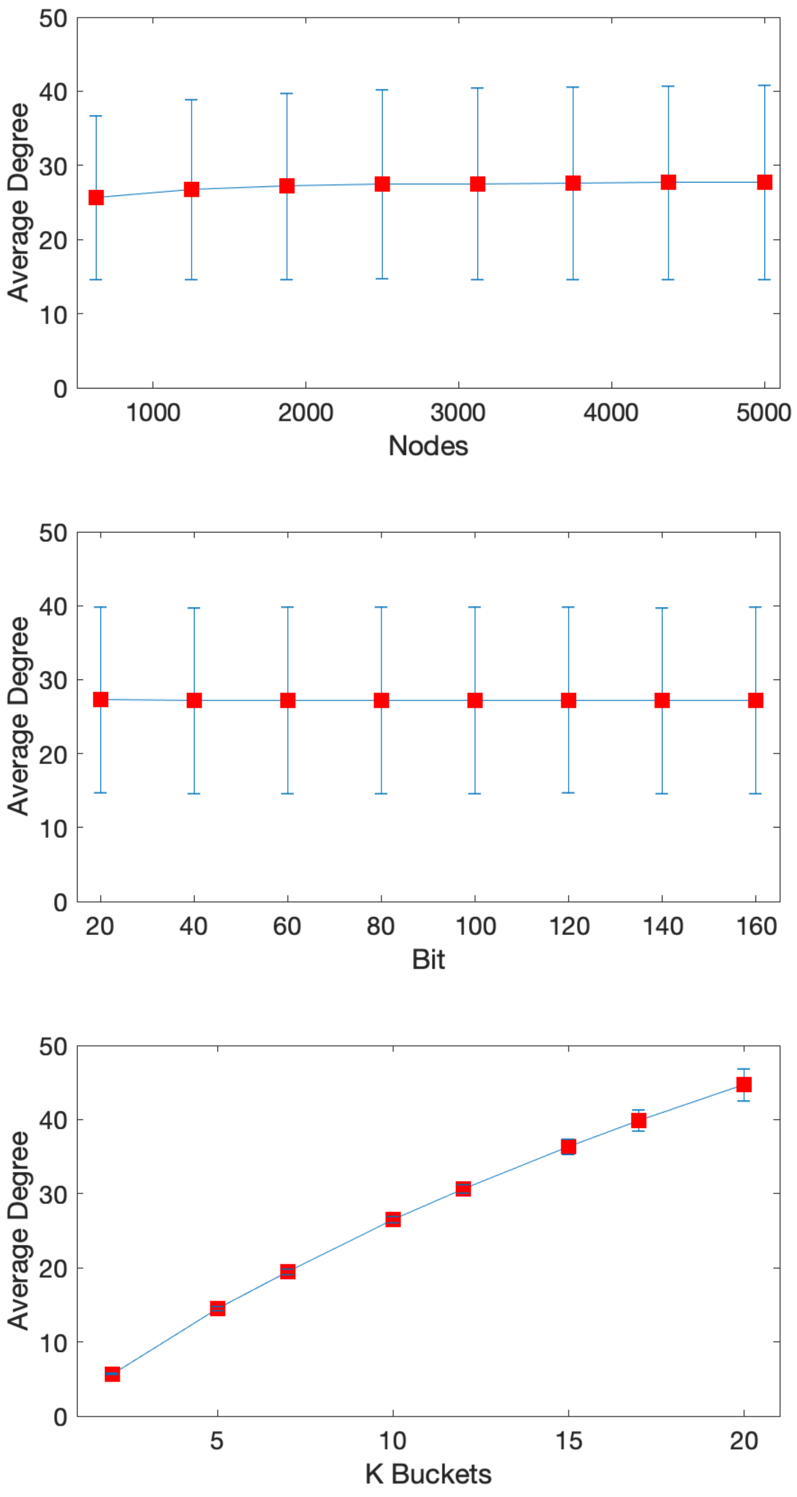


Figure 9: Line plot with errors of the Average Degree

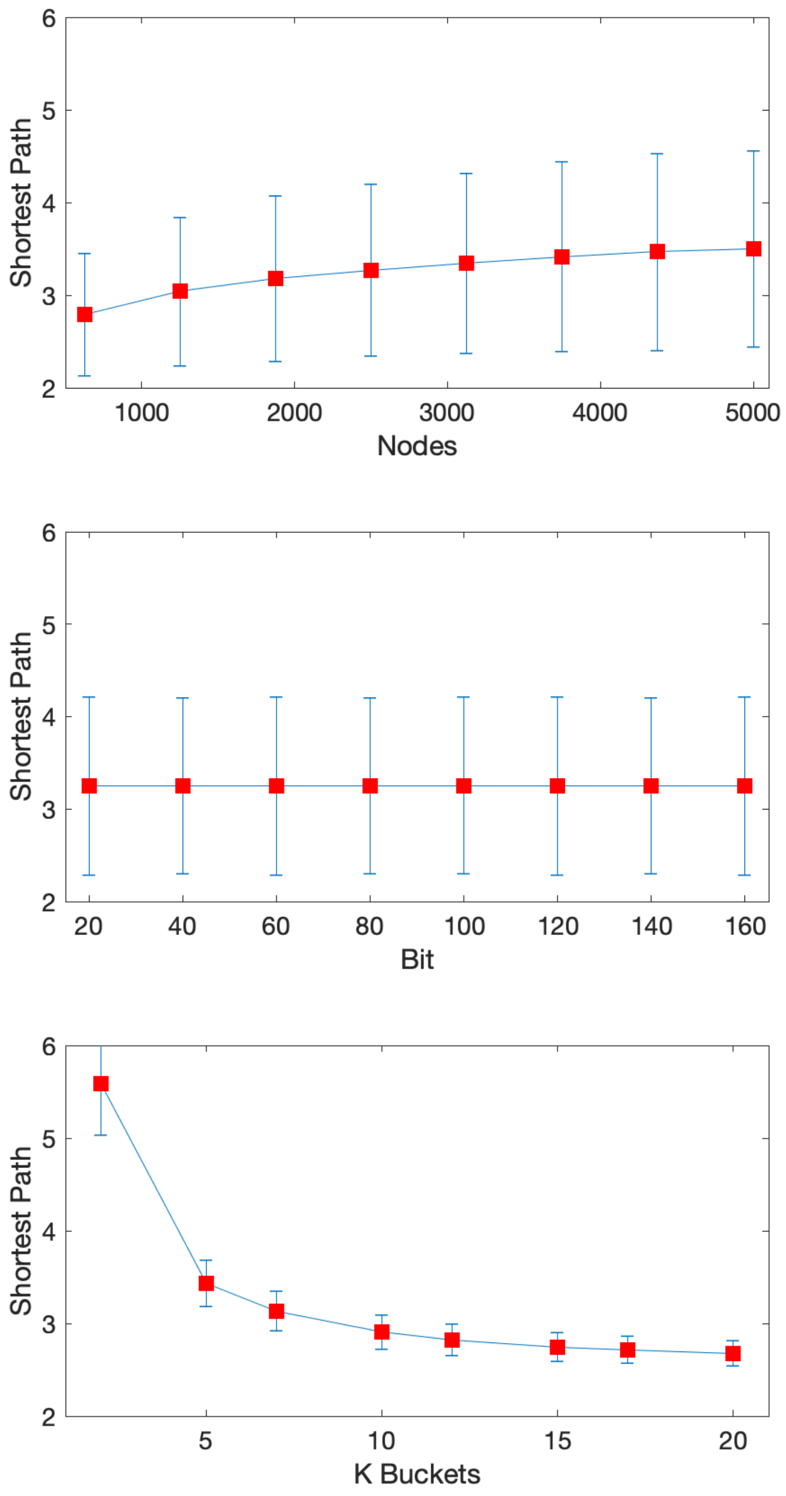


Figure 10: Line plot with errors of the Average Shortest Path

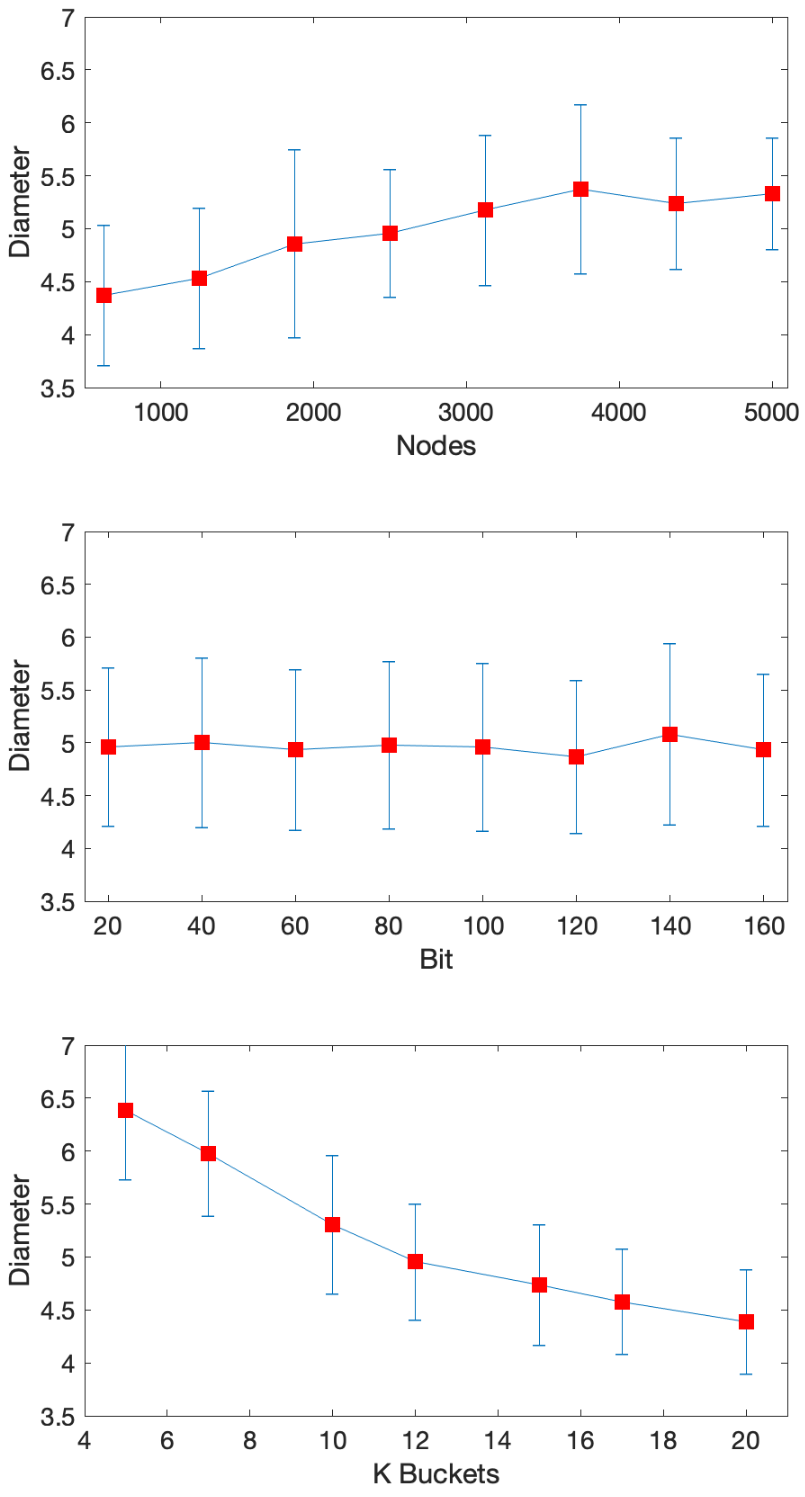


Figure 11: Line plot with errors of the Average Diameter

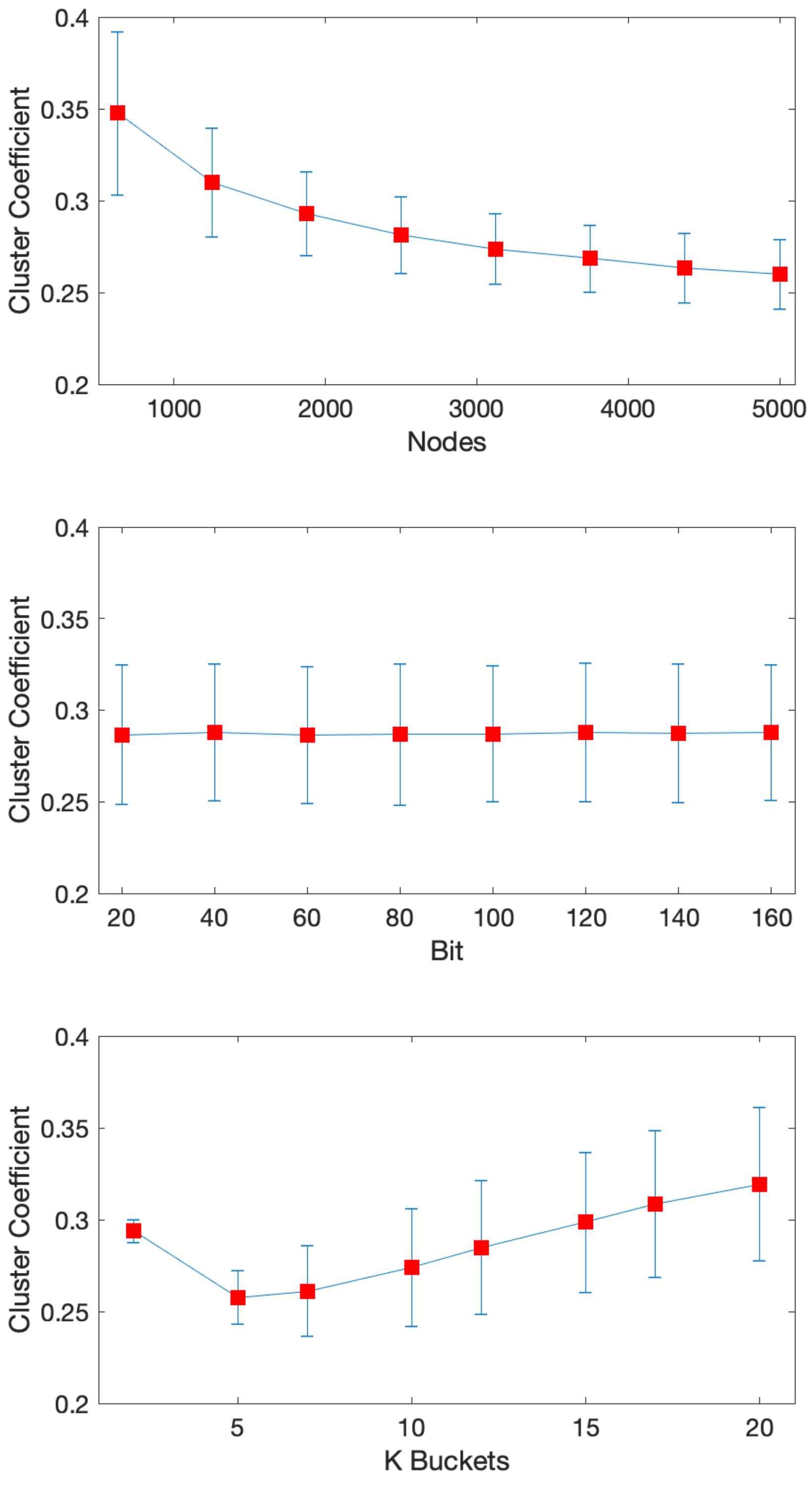


Figure 12: Line plot with errors of the Average Clustering Coefficient

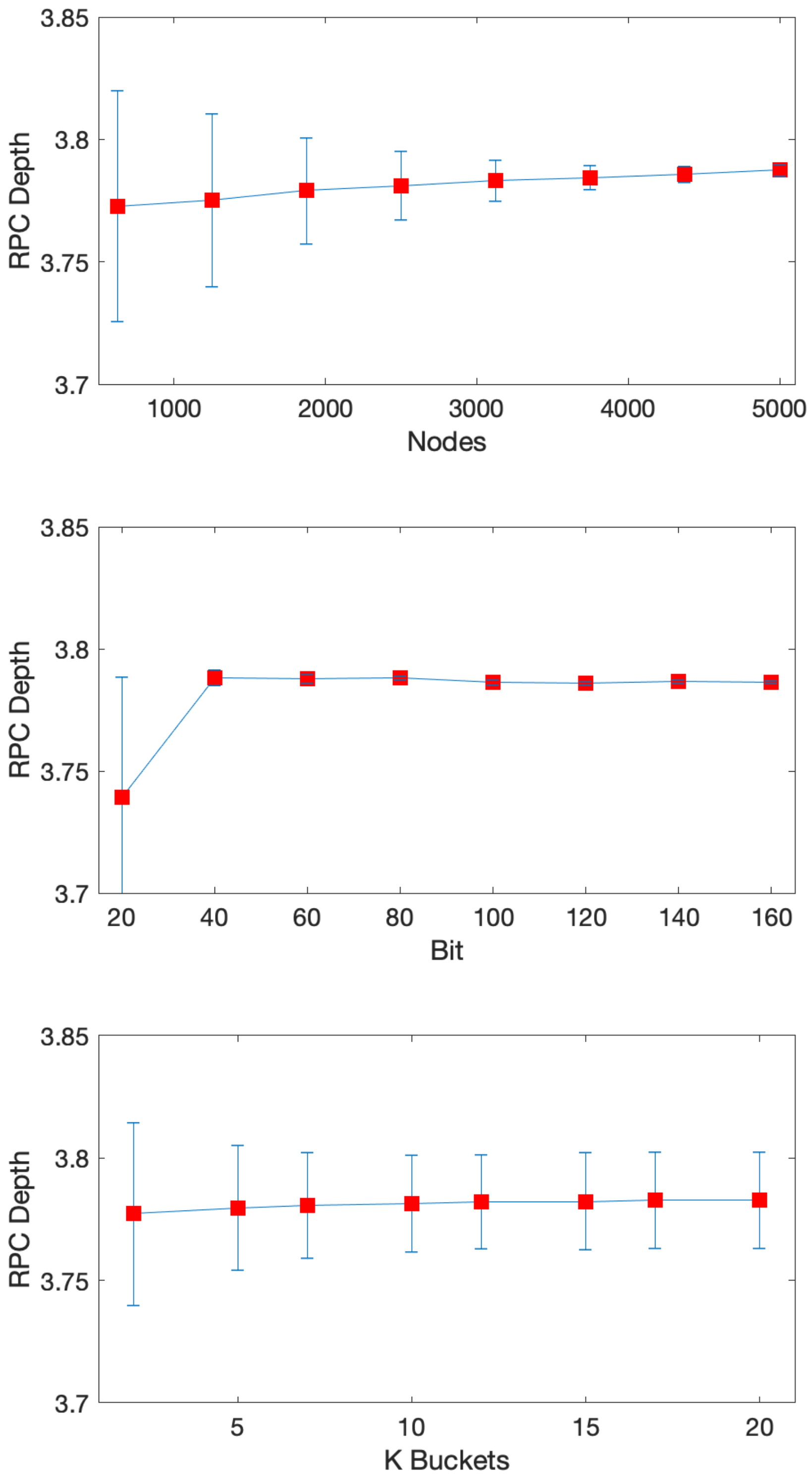


Figure 13: Line plot with errors of the Average RPC Depth

7 Data analysis

This section describes the analysis of the above results, by analysing each of the metrics previously discussed in section 5.

7.1 Average degree

As Figure 9 clearly suggests, the average degree depends almost exclusively by the size of the bucket (k). This is because the parameter of k is essential within the algorithms of Kademlia. Indeed, albeit we change the number of buckets b, what really matters is k. The main reason could be connected to the lookup procedure. This latter as the original paper says, will start by calling the α-closest contacts and will finish by returning the k-closest nodes found along the way. Hence, as a matter of fact, there is no connection between the lookup procedure and the number of buckets b.

7.2 Shortest path

The average shortest path can be seen as the number of people you will have to communicate through, on an average, to contact a complete stranger. By increasing the size of the buckets, it is easier to contact a "stranger" node. This implies that the lookup procedure converges faster with larger buckets size. Furthermore, it can be noticed a slight correlation between the average shortest path and the number of the nodes.

7.3 Diameter

The data extracted show us that the diameter is in inverse relation to the size of the buckets, whereas is directly proportional to the number of nodes within the network. The number of bits does not act in precise way, in fact, the diameter follows an oscillating trend.

7.4 Clustering coefficient

Our tests show that nodes are not highly clustered, but instead, there is a uniform distribution of nodes. Clearly the growing of the number of the nodes makes the clustering coefficient smaller, whereas, the growing of k makes the coefficient larger. Similar results have been discovered after the analysis of Gnutella[23]. This latter strengthens both the correctness of our work, as well as the fair distribution of the nodes within networks who are using distributed data structures.

7.5 RPC Depth

This ad-hoc metric has been created in order to count the number of recursive calls during the lookup procedure. As Figure 13 shows, inside the three graphs, our metric tends towards $\alpha + some\ costant$. More precisely, is important to notice the standard deviation with respect to the growing of the nodes number. Indeed, there is a higher degree of uncertainty at the beginning, while at the end, there's a stable trend towards $\alpha + some\ costant$.

8 Conclusions

Although nowadays the number of nodes is reaching the million's threshold, this study considered a set of parameters that respect entirely the original protocol use cases during the first years of work. The choice to fix the maximum number of nodes to 5000, has been dictated mostly by the poor computing resources available, but anyway, the results seems to follow the aims and the rules of the original paper.

9 References

[1] "Global consumer internet subsegment traffic 2022," *Statista*, 2019. [Online]. Available: https://www.statista.com/statistics/267194/forecast-of-internet-traffic-by-subsegment/.

[2] Cisco Systems Inc., "Cisco VNI: Forecast and Trends, 2017–2022 White Paper," 2019. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.

[3] R. Want, B. N. Schilit, and S. Jenson, "Enabling the internet of things," *Computer*, no. 1, pp. 28–35, 2015.

[4] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-addressable Network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, New York, NY, USA, 2001, pp. 161–172.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Middleware 2001*, 2001, pp. 329–350.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, New York, NY, USA, 2001, pp. 149–160.

[8] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Peer-to-Peer Systems*, vol. 2429, P. Druschel, F. Kaashoek, and A. Rowstron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65.

[9] BitTorrent Inc, "BitTorrent." [Online]. Available: www.bittorrent.com.

[10] D. G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," p. 32.

[11] Ethereum Foundation, "Ethereum Wiki," 2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/Kademlia-Peer-Selection.

[12] M. Steiner, T. En-Najjary, and E. W. Biersack, "A global view of kad," 2007, p. 117.

[13] S. A. Crosby and D. S. Wallach, "An Analysis of BitTorrent's Two Kademlia-Based DHTs," p. 29.

[14] H. Balakrishnan, M. F. Kaashoek, D. Karger, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems," *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, 2003.

[15] D. Eastlake 3rd and P. Jones, “US secure hash algorithm 1 (SHA1),” 2001.

[16] “HashSet (Java Platform SE 7).” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html. [Accessed: 29-Apr-2019].

[17] “ArrayList (Java Platform SE 7).” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html. [Accessed: 29-Apr-2019].

[18] “Amazon EC2 M5 Instances - general purpose compute workloads,” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/ec2/instance-types/m5/. [Accessed: 27-Apr-2019].

[19] “Intel® Turbo Boost Technology 2.0,” *Intel.* [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. [Accessed: 27-Apr-2019].

[20] “Java (programming language),” *Wikipedia*. 23-Apr-2019.

[21] “MATLAB - MathWorks.” [Online]. Available: https://www.mathworks.com/products/matlab.html. [Accessed: 27-Apr-2019].

[22] “NetworkX — NetworkX.” [Online]. Available: https://networkx.github.io/. [Accessed: 27-Apr-2019].

[23] “Gnutella - Network analysis of Gnutella - KONECT.” [Online]. Available: http://konect.uni-koblenz.de/networks/p2p-Gnutella31. [Accessed: 30-Apr-2019].