IPFS Monitor

Santi Gianmarco - 607016

A.A. 2019-2020

Repo: https://github.com/gianmarcosanti/IPFS_Monitor

1 Abstract

The aim of the present document is to illustrate the work done for the first assignment of the course *Peer to Peer Systems and Bolckchains.* In the first place, there will be an introduction to the methodologies and the tools which have been used to accomplish the goal of this assignment. In the second place, the results of the **IPFS Monitor** exercise will be discussed. Lastly, the question of the **The Kademlia DHT** section will be answered.

2 IPFS Monitor

2.1 Environment & Tools

The main purpose of the assignment was to build a program which will have monitored the usage of *IPFS*, collecting data about *Swarm*, bandwidth, data storage and network performance. Once the data collection phase finished, metrics should have been calculated and the result of which plotted in order to allow a good data visualization.

In order to accomplish this goal I wrote a **Python3** program which is composed of two main part. The first one takes care of the data collection phase and the second one deals with the plotting of the metrics' results. Both of the two parts have been developed using *PyCharm IDE.*

The monitoring function (`main.py`), exploits the *IPFS HTTP API* to collect data and storing them in ad-hoc data structures. On the other side, the plotting function (`plotter.py`) takes data previously saved and process them calculating the metrics. Once these have been calculated, the results will be organized in order to be able to use the library `MatPlotLib` for plotting them.

Since my conncection tends to be very instable and might have undermined the monitoring activity, the *IPFS* node have been mounted in *Digital Ocean* serve located in New York. The program have been running for 48 hours with a monitor interval of 15 minutes for a total of 192 monitor. The outcomes were elaborated succesfully.

2.2 Graphs & results

After the 48 hours of monitoring, all the collected data were stored in some `JSON` files. As second step, the plotter script took care of calculating and plotting the results of the following metrics:

- Number of Peers
- Distribution of peers' location
- Up time for each peer
- Bandwidth consumption
- Number of stored blocks
- Average latency per peer
- Average latency per country

2.2.1 Number of Peers

This metric's result shows the number of peers present in my node's swarm during the 48 hours of monitoring. We can see how this number is very high from the very first monitor. This is because the *IPFS daemon* was active before the monitor script were launched, so my node's swarm was already populated. As predictable, since the node were hosted in a farm there is not a huge variance, with a deviation of maximum ~ 150 peer from the average except for a peak during the 180^{th} monitor in which the peers were ~ 1100. In general, as visible in the graph, among in the first 100 ones the oscillation was not large between a monitor and the other. At the contrary in the last part of the activity this oscillation were much larger. For how concerns the average number of peer for each hour of the day we can see that all hours floats from ~ 700 to ~ 900.

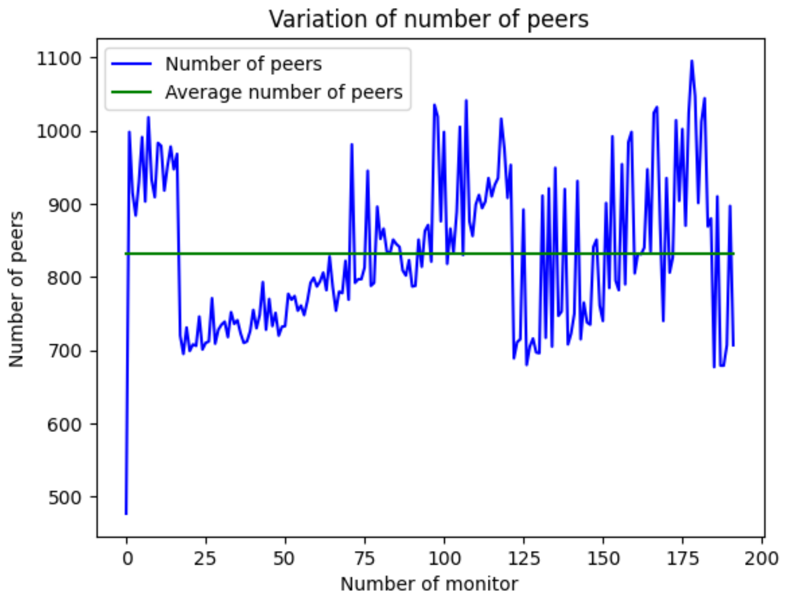

Figura 1: Number of Peers

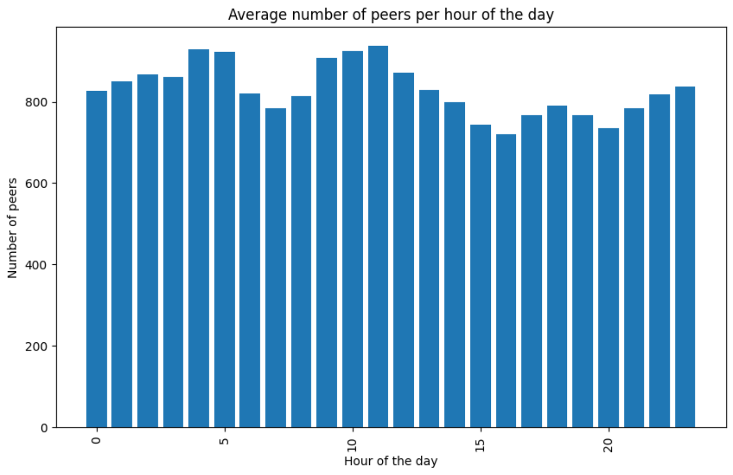

Figura 2: Hourly average number of Peers

2.2.2 Distribution of peers' location

The distribution of my swarm's peers during the monitoring phase, as evincible from the graph, was for almost an half located in the United States with $\sim$ 6000 peers detected over the $\sim$ 14000 totals. At the second place there is China, followed by Germany and France. For seek of clarity I preferred to split the original graph in two parts, in fact, part 1 shows the 50 most crowded countries and part 2 the least ones. I also preferred to avoid to show all those states which had less than 4 peers detected.

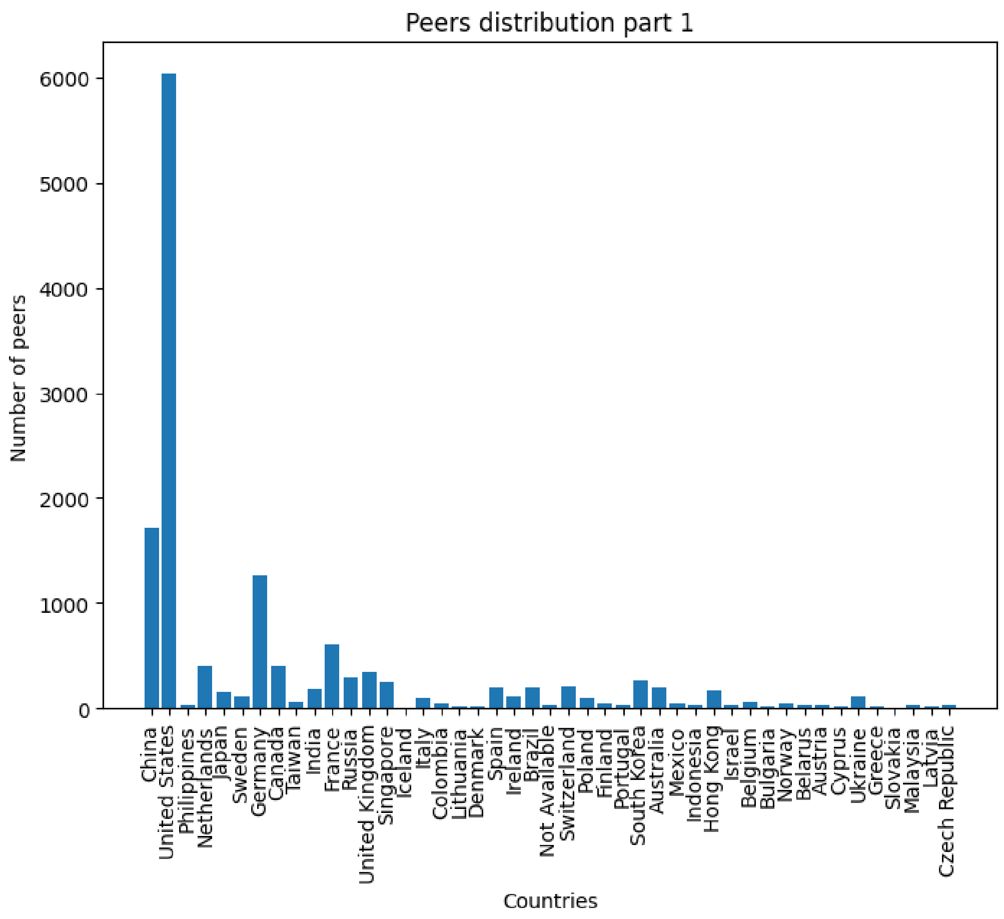

Figura 3: Distribution of peers' location part 1

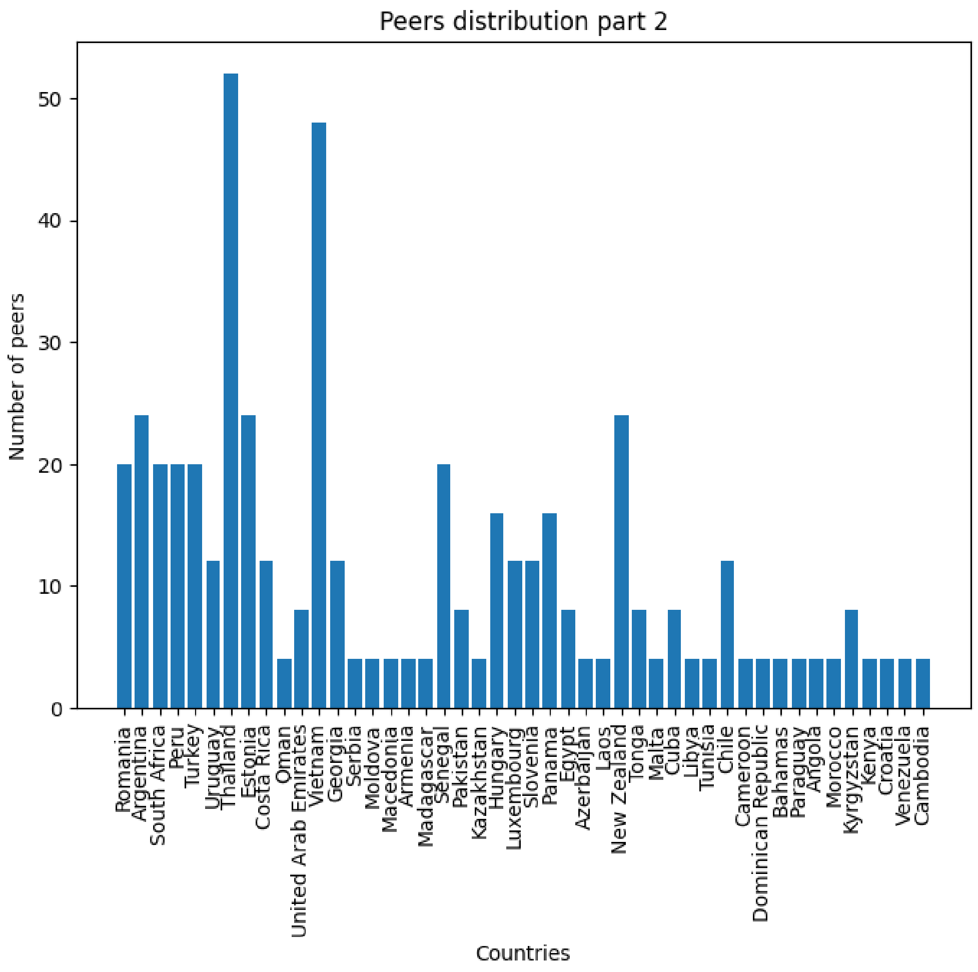

Figura 4: Distribution of peers' location part 2

2.2.3 Up time for each peer

The results of this metric show how over the half of the nodes detected were in the swarm for less than 15 minutes, this explains the oscillation in the number of peer metric. Only a small part of the nodes stayed in the swarm for more tha 10 hours.

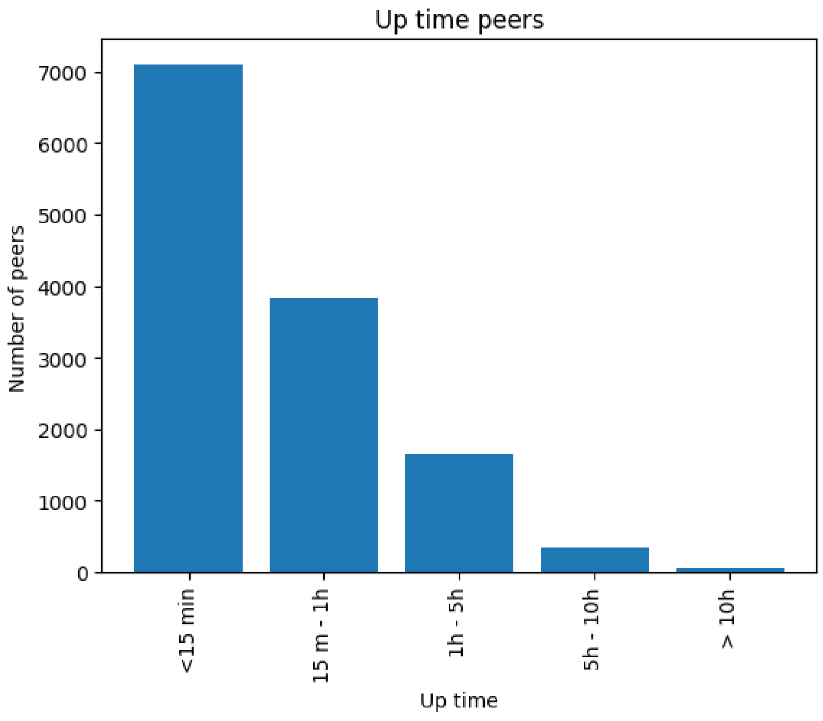

Figura 5: Up time for each peer

2.2.4 Bandwidth consumption

This metric shows the incoming and outgoing bandwidth consumption during the monitoring phase. For how concerns the incoming one, data denote a correlation between the raise of the number of nodes in the swarm and also of the bandwidth consumption, probably because the major exchange of data. The outgoing bandwidth, instead, fluctuated regularly between 5 and 25 megabytes.

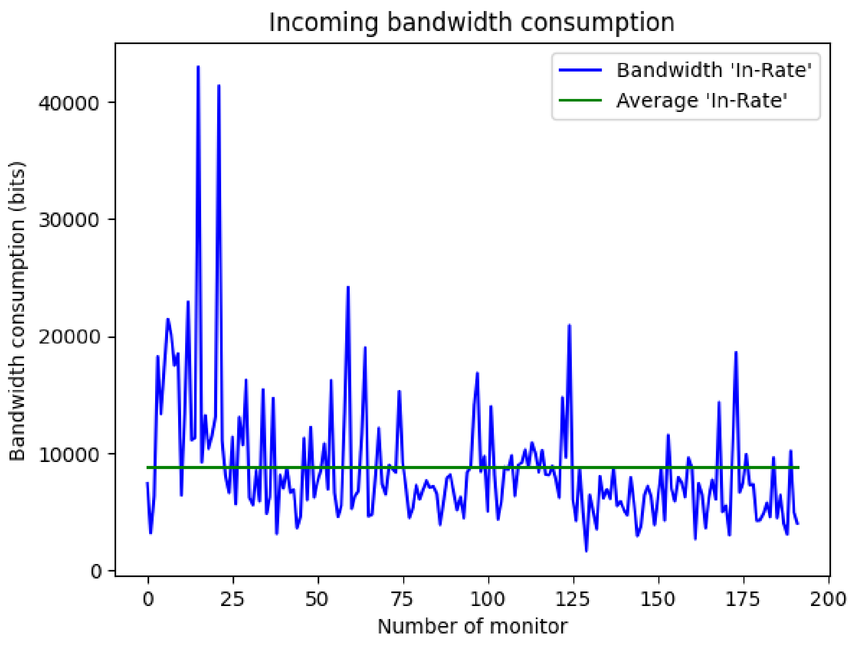

Figura 6: Incoming bandwidth consumption

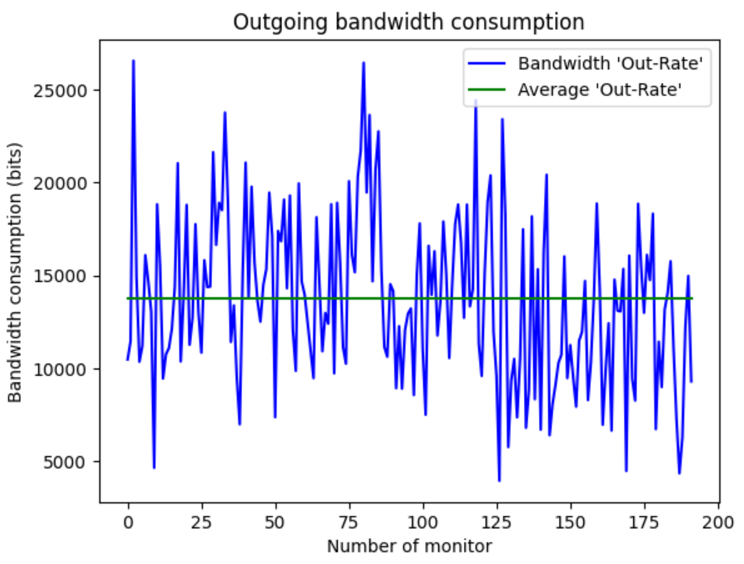

Figura 7: Outgoing bandwidth consumption

2.2.5 Number of stored blocks

The number of stored blocks metric shows that my node's storage amount kept growing until reached almost 16 millions of blocks then falling to 11 millions. This process of growing and falling was repeated twice during the monitoring period and, as we can see in the graph, at the end was restarting growing after a sharp drop.

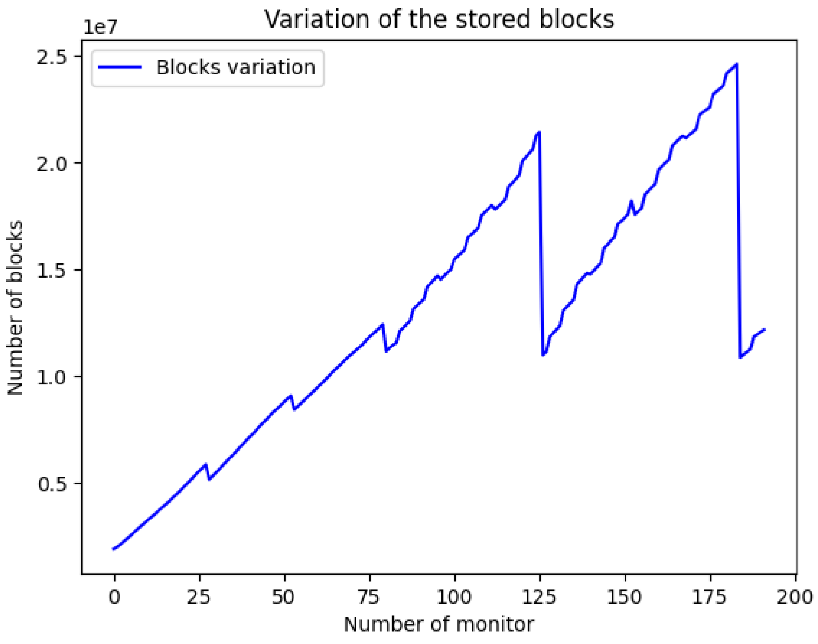

Figura 8: Number of stored blocks

2.2.6 Average latency per country

The last metric measured the average latency for each country. The lowest latency has been recorded in Mexico and the highest one in Czech Repulic and Spain. A consistent part of the detected latency's country, unfortunately was not available and reported in the graph with the label "Not Available".

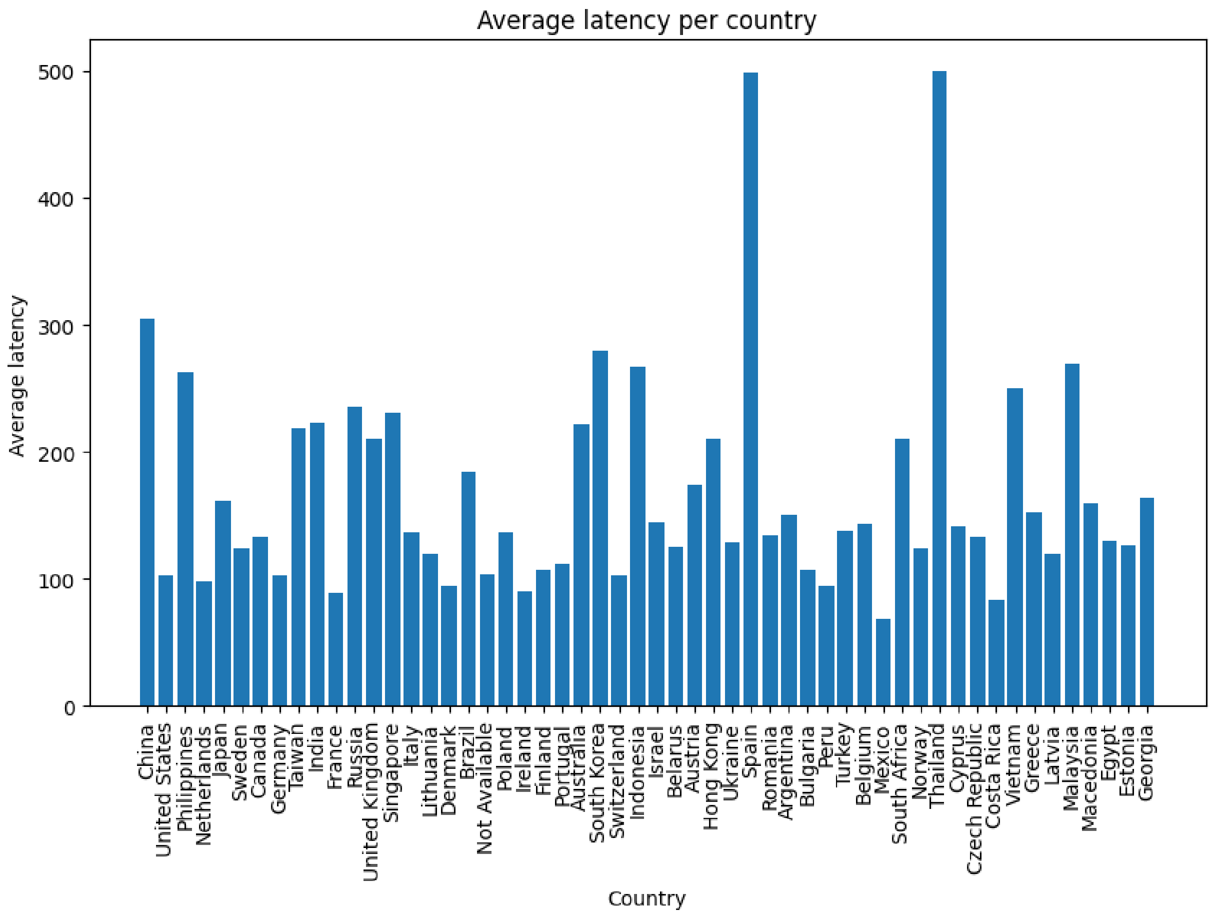

Figura 9: Average latency per country

3 The Kademlia DHT

3.1 Parameters

- $\alpha = 2$
- k = 2
- m = 8

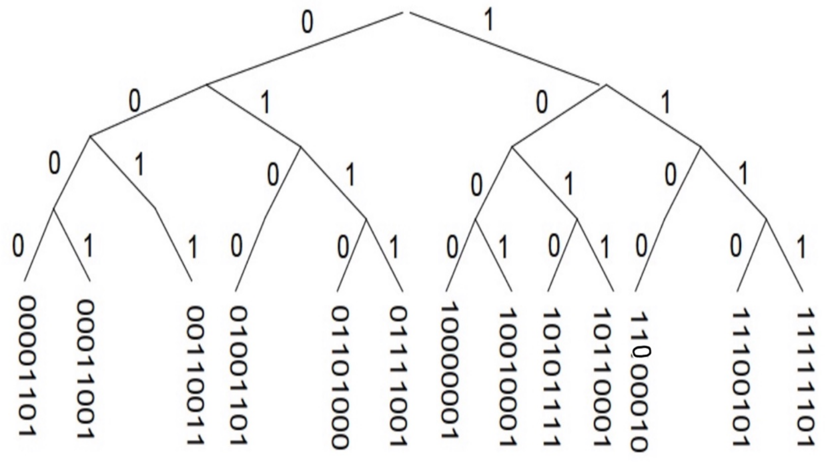

Figura 10: A Snapshot of a Kademlia overlay

Let us assume that the nodes in the Kademlia overlay, showed in the figure, are named with numbers from 0 to 12, starting from the leftmost leaf of the tree to the rightmost one.

3.2 Store of content

In the current scenario, there is n_{12} which inserts a content identified by 01000001. Let us assume that the routing table of n_{12} is the following:

0	n_0,n_3
1	n_6,n_8
2	n_{10}
3	n_{11}

- The first step of the store procedure is to understand how long is the common prefix between n_{12} and the key that it wants to store. Since the common prefix has length 0 the nodes responsible for the key will be in the leftmost sub-tree.
- The second step of the store procedure is to do a node `LOOK_UP`, which consists in finding the `k-closest` nodes to the key that need to be inserted. Firstly, n_{12} needs to take α nodes from his 0^{th} `k-bucket` (since the length of the common prefix is 0). Once these two nodes have been chosen, n_{12} will proceed by invoking a `FIND_NODE(key)` on each of them. Since k = α and the 0^{th} bucket of the routing table is full, the chosen nodes will be n_0, n_3. Let us now assume the following:

$$\texttt{FIND\_NODE(key)} \rightarrow n_0 \Rightarrow \{n_3, n_4\}$$

$$\texttt{FIND\_NODE(key)} \rightarrow n_3 \Rightarrow \{n_4, n_5\}$$

At this point n_{12} needs to calculate the distance between the key and $\{n_3, n_4, n_5\}$, the ones received as result of the previous `FIND_NODE` calls. In order to accomplish this task the `XOR-metric` will be used.

- In the end, once identified the k-closest nodes, n_{12} will use the `STORE RPC` to store the key in these nodes.

3.3 Join node

In the current scenario, there is the node 11010101 (which from now will be called `new` for simplicity) who wants to join the network. In order to accomplish this task it needs a boostrap node n_2, which will help `new` to build its own routing table. The `JOIN procedure` is composed by the following steps:

- `New` sends a `FIND_NODE(new)` to n_2 and will receive a response with its `k-closest` nodes.

$$\texttt{FIND\_NODE(new)} \rightarrow n_2 \Rightarrow \{n_{10}, n_{11}\}$$

- N_2 will add `new` to his routing table and at this point `new` has joint the network.
- `New` adds n_2 to its routing table and starts to `PING` those nodes given as result from the previous `FIND_NODE` call. The `PING` action is needed to be sure that such node are alive and also to let them know about the existence of the `new` node. Assuming that all the `PING` calls have returned positive results, `new`'s routing table will be this:

0	n_2
1	
2	n_{11}
3	n_{10}
4	
5	
6	
7	

- At this point, `new` needs to fill its own routing table and in order to do this it will generate a random, but consistent, identifier for each `k-bucket` and will actuate a `LOOK_UP` for this fake id. As a reminder, the `LOOK_UP` procedure consists in finding the k-closest nodes in our own routing table and then making a `FIND_NODE(key)` for each of them. The simulation of this procedure is the following:
  - `k-bucket` number 0, k-closest nodes with `XOR metric` = $\{n_2$, $n_{10}\}$:

$$\texttt{FIND\_NODE(random\_id = 00000000)} \rightarrow n_2 \Rightarrow \{n_3, n_4\}$$

$$\texttt{FIND\_NODE(random\_id = 00000000)} \rightarrow n_{10} \Rightarrow \{n_2, n_5\}$$

Choose α nodes from $\{n_2, n_3, n_4, n_5\}$ with `XOR metric` $\Rightarrow\{$ n$_2, n_3\}$

`PING(`n_2`)`

`PING(`n_3`)`

Add $\{n_2, n_3\}$ to the routing table

0	n_2, n_3
1	
2	n_{11}
3	n_{10}
4	
5	
6	
7	

– `k-bucket` number 1, k-closest nodes with `XOR metric` $= \{n_{10},\, n_{11}\}$:

`FIND_NODE(random_id = 10000000)` $\rightarrow n_{10} \Rightarrow \{n_6, n_7\}$

`FIND_NODE(random_id = 10000000)` $\rightarrow n_{11} \Rightarrow \{n_8, n_9\}$

Choose α nodes from $\{n_6, n_7, n_8, n_9\}$ with `XOR metric` $\Rightarrow\{$ n$_6, n_7\}$

`PING(`n_6`)`

`PING(`n_7`)`

Add $\{n_6, n_7\}$ to the routing table

0	n_2, n_3
1	n_6, n_7
2	n_{11}
3	n_{10}
4	
5	
6	
7	

– `k-bucket` number 2, k-closest nodes with `XOR metric` $= \{n_{10},\, n_{11}\}$:

`FIND_NODE(random_id = 11000000)` $\rightarrow n_{10} \Rightarrow \{n_{11}, n_{12}\}$

`FIND_NODE(random_id = 11000000)` $\rightarrow n_{11} \Rightarrow \{n_{10}, n_{12}\}$

Choose α nodes from $\{n_{10}, n_{11}, n_{12}\}$ with `XOR metric` $\Rightarrow\{$ n$_{11}, n_{12}\}$

`PING(`n_{11}`)`

`PING(`n_{12}`)`

Add $\{n_{11}, n_{12}\}$ to the routing table

0	n_2, n_3
1	n_6, n_7
2	n_{11}, n_{12}
3	n_{10}
4	
5	
6	
7	

– `k-bucket` number 3, k-closest nodes with `XOR metric` = $\{n_{10}, n_{11}\}$:

`FIND_NODE(random_id = 11000000)` $\rightarrow n_{10} \Rightarrow \{n_{11}, n_{12}\}$

`FIND_NODE(random_id = 11000000)` $\rightarrow n_{11} \Rightarrow \{n_{10}, n_{12}\}$

Choose α nodes from $\{n_{10}, n_{11}, n_{12}\}$ with `XOR metric` $\Rightarrow$\{ n_{10}\} *

`PING(`n_{11}`)`

`PING(`n_{12}`)`

Add $\{n_{11}, n_{12}\}$ to the routing table

0	n_2, n_3
1	n_6, n_7
2	n_{11}, n_{12}
3	n_{10}
4	
5	
6	
7	

* *The only node with prefix length 3 is 10 and since both nodes 11 and 12 are already present in the routing table they won't be took in consideration.*

The final routing table of node `new` will be:

0	n_2, n_3
1	n_6, n_7
2	n_{11}, n_{12}
3	n_{10}
4	
5	
6	
7	