Ex 1 (through IPFS CLI)

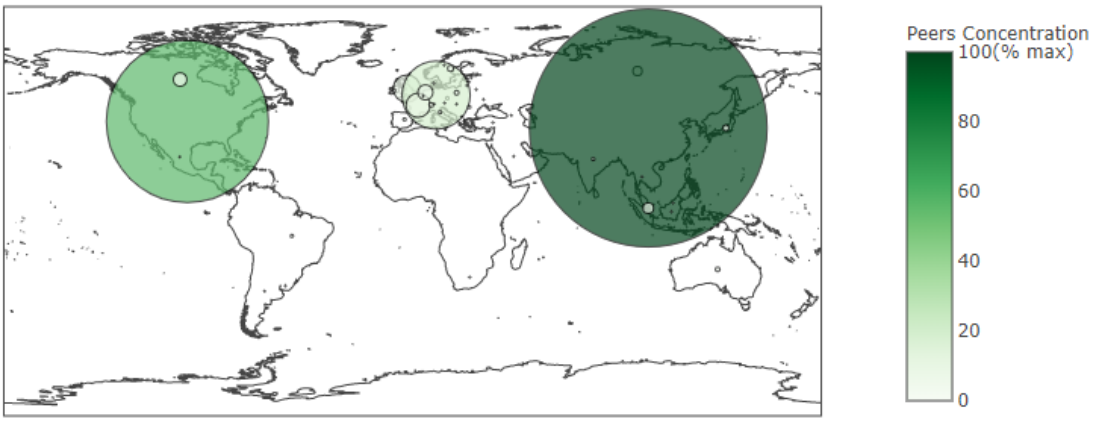
Little code's explanation (per folder)

- **Logging:** It will build an OS-dependant process that will invoke the given command into the console and then redirect the output into log files. **[Tested** on Windows 10, Arch Linux and MacOs**]**
- **Analysis:** once the logs are done, it simply parses what I thought were relevant pieces of information for the assignment. Once the data is elaborated, it will create a minified version of JavaScript code to show the results.
- **Configurations:** from Configurations.java it's possible to change many things, like the command to be invoked in the command-line, the amounts of cycles, etc. There's also an enum containing the countries' 2 letters and 3 letters codes.

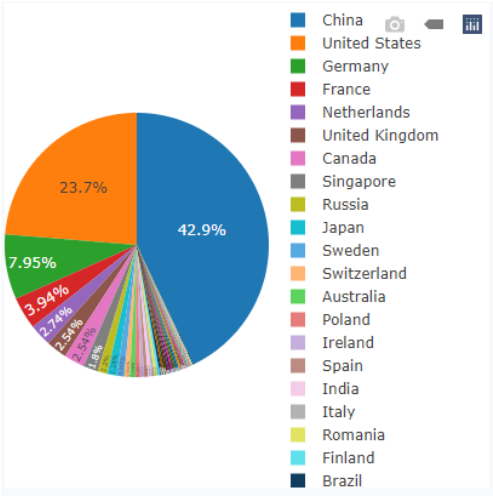
Results

I've chosen a simple and direct way of showing the results of this monitoring.

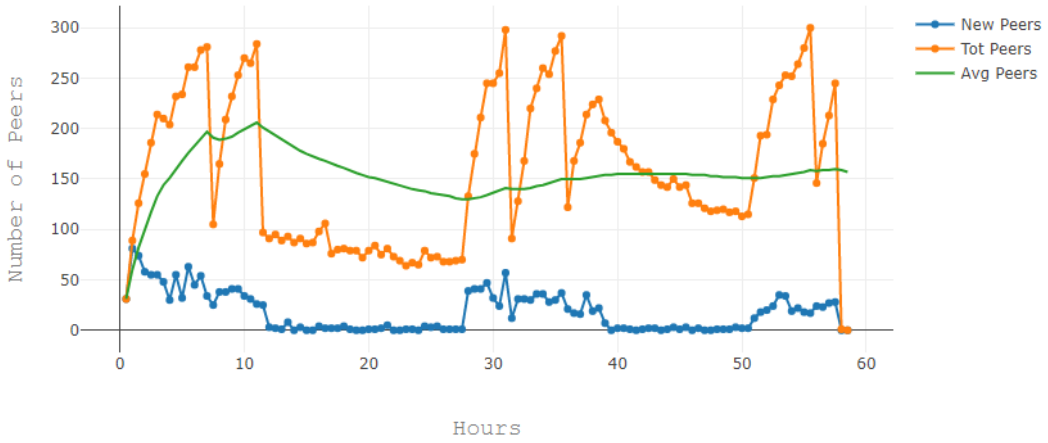
At first glance, we can see the world's representation: the bigger and greener is the circle, the higher quantity of peers was found to be analyzed in that location.
Note: *it's possible to zoom in/out and drag the world view*



It's immediate to see that China and the United States had far more people than any other country. These values are better seen as percentages in the lower pie chart, where it shows that nearly 2 peers out of 3 were from these two countries alone. By hovering over, it's possible to see the exact amount (e.g. China 642 peers).



In the middle of the two graphs there's a final one, showing the changing in the number of total, new and average number of peers every half an hour. The total time of consecutive monitoring was almost 2.5 days.



The **bootstrap phase**, initiated at 9 PM of April's 3rd, took off very quickly: in less than 3 hours I was already at the average level I'd stay for the rest of the monitored period, which is around 150 peers.

After nearly 4 hours there was a huge drop in connections, but that got quickly recovered. To be noted is that at that moment I gained 38 new peers but total increased from 105 to 165, so many of the "new" connections were peers met in the past. Then, after another quick increase, a very long period (nearly 8 hours) where the peers would stay almost constant: close to 0 new peers, while total peers are nearly the same all the time. Then again, 39 new peers and a 63 total increase: this suggests some kind of pattern that lets my average constantly sit at nearly the same level all the time. This is suggested by the Churn rate too, that is around 1%, meaning that the network doesn't suffer from constant disconnections and it's highly adaptive.

General stats that I thought were relevant, visible in the bottom right of the page, are:

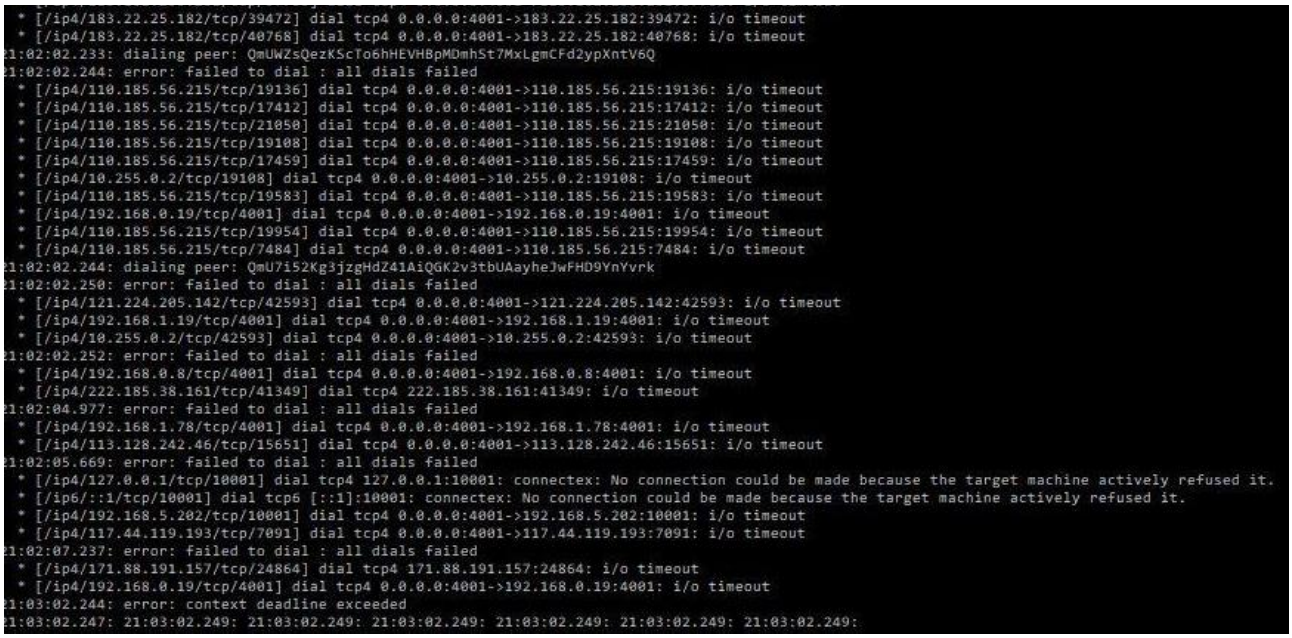
- Addresses collisions: 1/3 of the peers shared the same IP address, but different ports (protocol could have been another possibility, but TCP users were 100% so it's not the case).
- CID Version: there's still a majority of CID version 0, but Version 1 is being used enough to give consistent presence.
- IPV6 addresses: the total absence of this is because my internet connection doesn't support IPV6.
- The connections' direction huge difference between in and outs can be explained by the fact that I've never actively uploaded anything into the network, causing the directions to be mostly one-way.
- The "Fast, Ok, Slow" peers were designed to manually classify peers' latency: I've decided that people with less than 300ms were considered fast, peers with latency lower than 1 second were ok, and finally all the others are considered slow. That's what I felt like a good value would be, and the average proved in my favor sitting at 292ms.
- Churn, simply because it's one of the most important stats in distributed networks.

Optional Part

I've tried to use the command as requested, but I had some trouble because the command was behaving differently from what I thought it would do. So I've decided to do some online investigations, where I've found that the official documentation of the command *"ipfs-dht-query"* was scarce, but stated that it's used to "Run a 'findClosestPeers' query through the DHT". So I've decided to manually test this command and

see what it was about, so all of the things I'm going to say are from my personal experience of the command's usage.

The first thing that we can see that this command does is requesting an ID: both searching for myself and other IDs would result in some sort of cycle. I tried to understand first the type of iteration and I noticed that it's done iteratively, and I've proved this by looking at what nodes are inspected cycle after cycle. It looks like it tries to reach some nodes and, if any fails to respond, it will simply "rollback" of one iteration and ask other peers. The problem is that every time I tried to execute the command, it'd fail due to a timeout error for all the nodes, ending in no results at all. (Proof in the image)



Assuming that it's a problem from my side and that it'd actually "find the closest peers", then the only usage of this command that I can come up with is to calculate the number of hops between the sender and the final node: maybe this can be used in order to dynamically increase and decrease the size of the k-buckets, to maintain a good trade-off between the space occupation and the time requested to query something that is stored way outside of the sender's range (using the properties of the XOR metric).

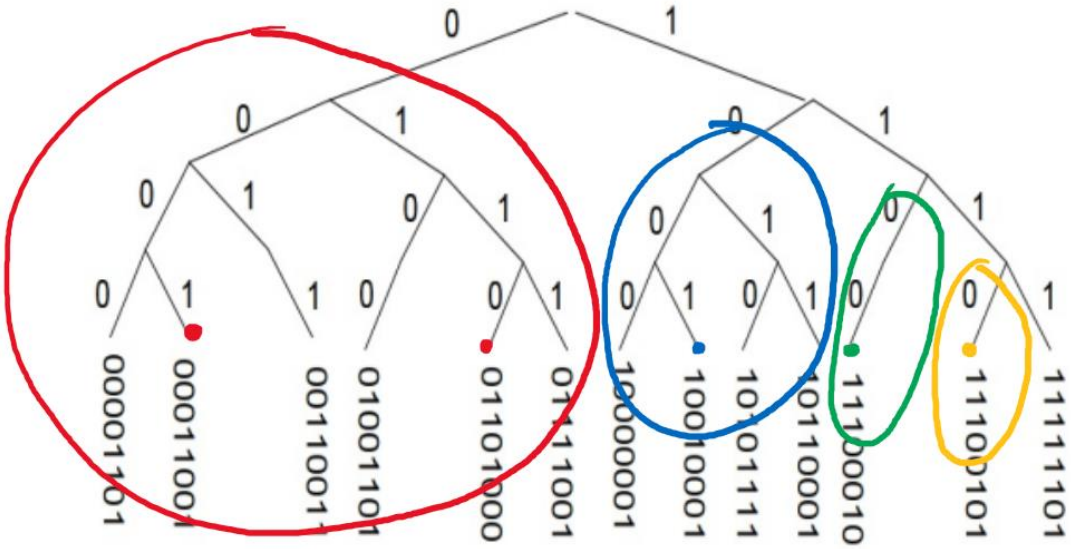
Ex 2

Question 1.

Node *111111101* wants to insert the pair *<01000001, some_value>*. This node is already belonging to the Kademlia tree, so he simply has to localize the k closest nodes to the key *01000001* and send a store to each one of them.

*Sender will be called **S**, the key will be called **K**.*

To achieve this (through an **iterative** algorithm), we have to see the Kamdelia tree from the node's view: we divide the tree into different decrementing sets that do not contain the node (see image). We know that the protocol assures that any node will know <u>at least</u> one node on each subtree *(in this exercise I've assumed the colored dots are the known nodes from the sender node)*.



To identify the k closest nodes to **K**, **S** will use the XOR distance and will send in parallel two (α) asynchronous query messages to the two red nodes (identified from left to right as *R1* and *R2*).
Note: if there was only one red node, then the other one would have been the blue node.

Both *R1* and *R2* will return 2 nodes' IDs that they think are closer to the requested key **K**. From our point of view, this means that the best targets are the nodes between *R1* and *R2* (inclusive).

Assuming that *R1* knows *00110011* and *01001101*, *R1* will return them to the sender.

Assuming that *R2* knows *R1* and *01001101*, *R2* will return them to the sender.

Then, the sender **S** will ask the same query messages I just explained to these newly retrieved nodes, α at a time, in order to find closer values to the key **K**. Once the returning nodes from these calls aren't better than the ones **S** was saving as "closer ones", it stops.

Due to different hops, there will be more or less delay in each "path": anyway, the resulting pair of IDs will be *01001101* and *01101000* as the closest ones.

Finally, the sender will send a **store** message to these 2 nodes.

Question 2.

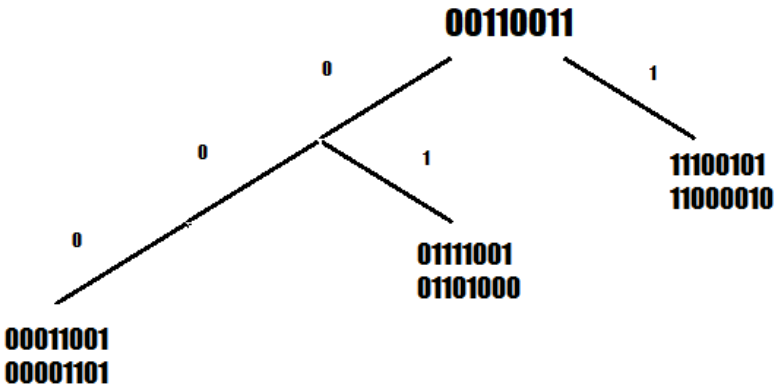
11010101 (N) wants to **join** the network through the **bootstrap** node 00110011 (BS).

Assumptions:

- Every node will answer all the possible requests without failing (pings, etc..) unless stated differently
- The tree has depth 4, so I'll simply talk about the relevant non-empty buckets.

1st step:

- **N** has in its Routing Table the **BS** node.
- The Routing Table of **BS** contains:



The Routing Table is a tree that has as leaves the node's k-buckets: since k is 2, there are at most 2 nodes for each bucket. Each path represents the Longest Common Prefix with the nodes.

2nd step:

- **N** sends a **FIND_NODE(N)** to **BS**, and **BS** will return the k closest nodes to that ID that he knows. Considering the previous image, it will return the 2 nodes on its right, so 11100101 and 11000010.

3rd step:

- **N** adds the returning nodes to its Routing Table
- **N** then proceeds to generate random IDs for each empty range, in order to fill up both its own view and to possibly populate others' k-buckets, depending if they're full

Note: The protocol states that the node is added if the k-bucket doesn't contain it yet and is not full. If it's full, it will ping the head of the list for up to 5 times before replacing it: this prevents DoS attacks and asserts a good behavior of the whole network, due to the fact that older nodes are more stable than new ones. More than that, this is the moment when N gets to know the keys he has to take care of, by the implicit knowledge of the neighbors.

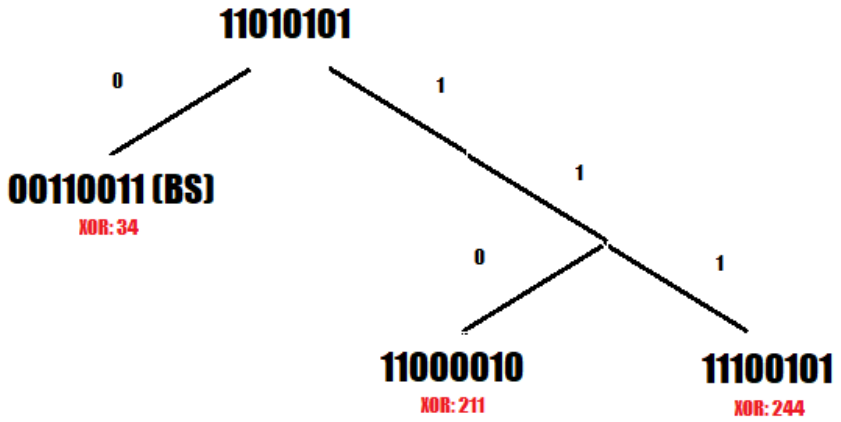
Possible outcome for the first iteration:

For example, **N** will generate random IDs for each bucket (classic DFS ordering), like:

- **0**0010001 for the bucket 0
- **10**101111 for the bucket 1
- **111**00000 for the bucket 2
- **1100**0011 for the bucket 3
- And so on, as stated in *Assumptions* I stop at this level but it can be expanded.

Generated IDs may actually exist in the Kademlia tree or may not, this step is just to populate the routing tables.

So N will ask to its k closest nodes to the generated key “00010001”, depending on the XOR metric. At the moment, the routing table of **N** is presented in the image *(in red the XOR distance from the random ID):*



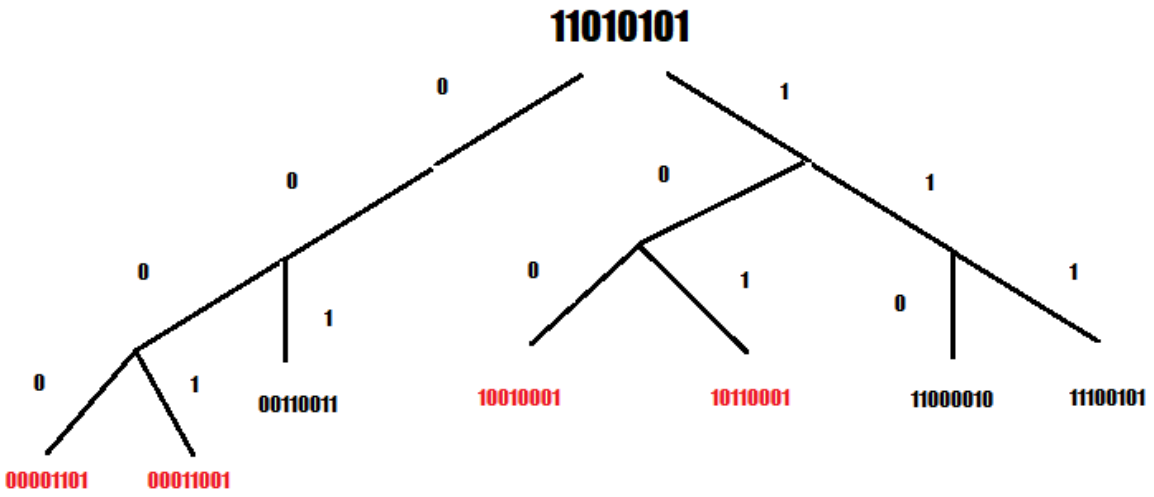
Looking at the XOR distances, the 2 closest nodes are the one with XOR distance equal to 34 and the one with XOR distance equal to 211. So, it will send two message queries:

- FIND_NODE(00010001) to 00110011
- FIND_NODE(00010001) to 11000010

Both of them will then return the k closest nodes to that requested ID, which may be like:

- 00001101 and 00011001 for the BS node
- 10101111 and 10000001 for the other one

Every returning ID will then be added to the Routing Table, following the previous protocol for adding nodes into the appropriate k-buckets, resulting in *(in red the newly added nodes)*:



This is how it is possibly populated after one search. In this case, since all returned IDs were unknown, they got all added.

This is then repeated for each message that I listed earlier: *10101111, 11100000, 11000011.*