Monitoring the Interplanetary File System

Jacopo Massa - 543870

April 2020

1 IPFS Swarm Monitoring

1.1 Code

The Python script used to analyze the IPFS swarm exploits the HTTP API provided by IPFS (in particular the `ipfs swarm peers` one). the script is made by two main modules:

main.py: the module that effectively logs data, manipulating them a bit before writing them on a CSV file.

graphs.py: the module that plots the data logged by *main.py* into different types of charts, explained in the next section.

For all the other *sub-modules* see the docs and comments into the files themselves.

How to run the code

I've set up a `requirements.txt` file to install all the dependecies, so my advice is to use a virtual environment in which to install these requirements:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt

python main.py  # to start logging
python graphs.py # to see the results for the logged data
                 # (it can take a while to start)
```

1.2 Graphs and Results

In this section I'll list and briefly explain/comment the 5 kinds of graphs that I've built on the data collected in 4 days of monitoring (April 5 - April 9).

1.2.1 Geo-localization of the swarm

In the first graph we can see how the swarm is almost entirely located in China (2020 peers) and in the United States (820 peers). The other main groups of peers are in Western Europe, followed by small (or single) peers scattered throughout the rest of the world. The amount of peers per country is shown by the size of the several pops in the figure.

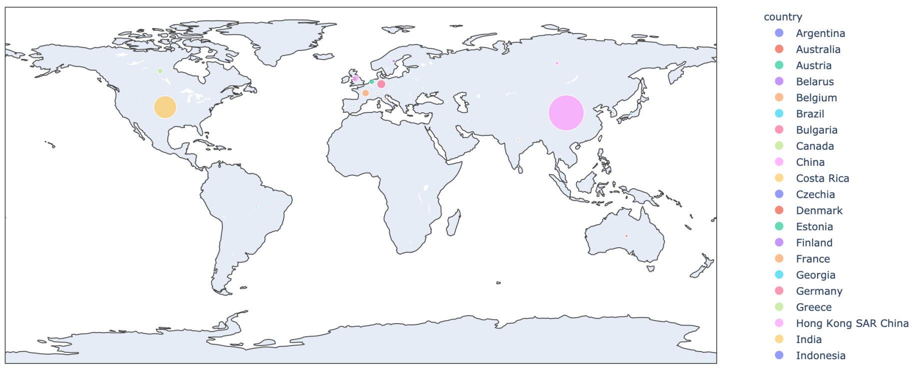

Figure 1: World representation with number of peers per country

The following pie chart shows the previous distribution in terms of percentage for each country:

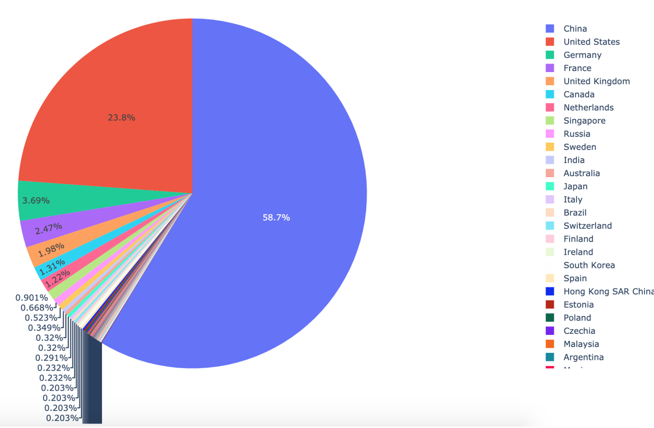

Figure 2: Pie chart with percentages per country

These two graphs consider the unique peers connected to the swarm, so if a peer was logged twice or more, it has been counted only once.

1.2.2 Number of peers and Latency variation

The initial bootstrap phase was almost imperceptible, since after only an hour my node had reached the average number of connected peers (more or less 150, as shown in Figure 4). For the rest of the monitoring period the number of peers grew and decreased regularly, with peaks especially in the morning hours (GMT time). The latency (obvioulsy) changed proportionally to the number of peers, keeping a value around 500ms on average.

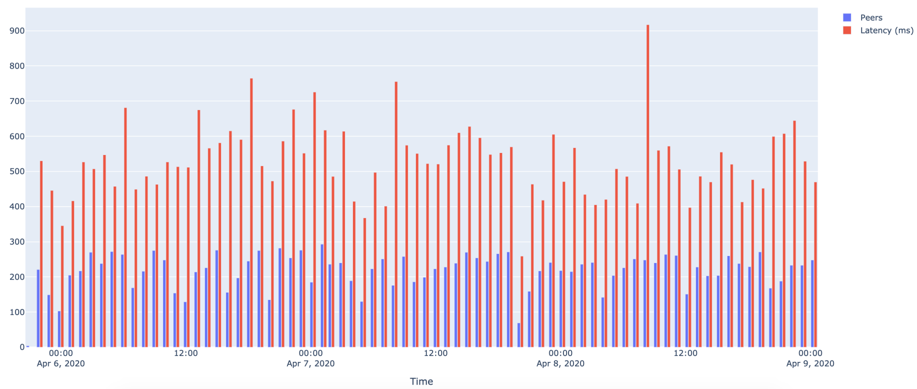

Figure 3: Hour-daily behaviour

On April 7 at 8 P.M., I had some problems with my internet connection, so the data are a bit distorted around that period, but they came back to average after a couple of hours.

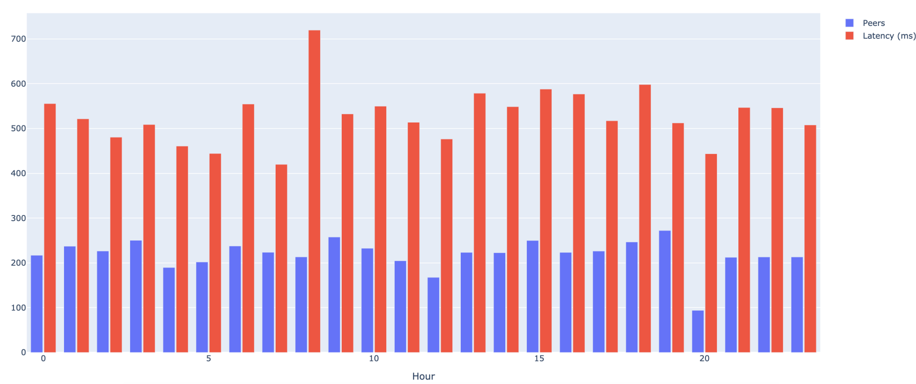

Figure 4: Average-hour behaviour

1.2.3 Protocols usage

The last graph shows which protocols were present in the registered multiaddresses, showing that:

- `tcp` was almost always present
- `ipv6` nodes initially prevail, then attenuate and equalize with `ipv4` ones.
- there are some other protocols like `p2p-circuit` (the one used for circuit-relay) or `p2p` (the newer identifier for `ipfs`, as shown in this table) which are less used.

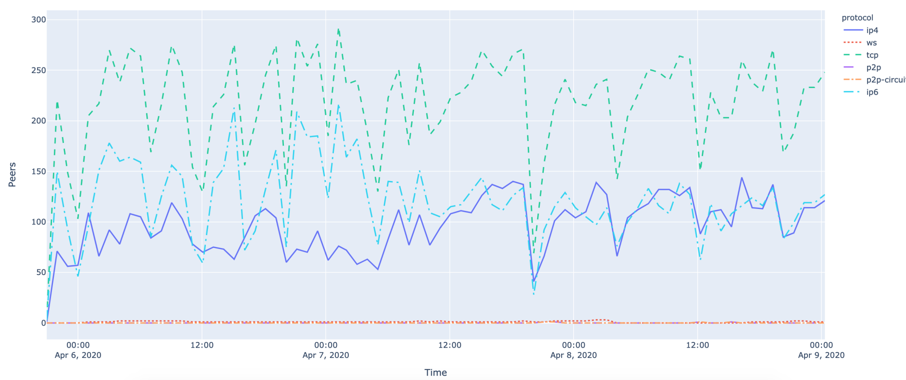

Figure 5: Protocols usage

1.3 Additional Remarks

I haven't made statistics on the CID version of the peers, since I've noticed an inconsistency between the CID documentation and the results retrieved via the py-cid module (implemented on the basis of the previously linked documentation). The module always retrieved 0 as CID version, both for the CID starting with `Qm` or not (for instance `12D3`), showing that only the multihash was changing (ex. `sha256` for the first, `identity` for `12D3`). The same results were obtained on the CID inspector.
However, if we consider *version 0* only the CID starting with `Qm`, these prevail in number over the other types of CID, as shown in the log file attached to the code.

For a better and more detailed visualization of the graphs I recommend to run the `graphs.py` web service, that will open a web page with a complete list of the graphs and also additional statistics (do not remove/change the `log.csv` file if you want to see my results).

2 The Kademlia DHT

Known variables:

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

N.B. I've named the nodes of the given Kademlia Snapshot with $\texttt{n}_\texttt{i}$ where i is the position of the node starting from left (for instance: `00001101` is $\texttt{n}_1$ and `11111101` is $\texttt{n}_{13}$)

2.1 STORE

Calling $\texttt{s}_\texttt{node}$ the node that inserts the content, and `key` the identifier of the content, the steps to understand which node(s) will be responsible for the content identified by `key` are the following:

- **Find α nodes from the k-bucket closest to** `key` **:**
  This is the starting point to perform a **node look-up**.
  The common prefix between $\texttt{s}_\texttt{node}$ and `key` is 0, so $\texttt{s}_\texttt{node}$ picks two (because the degree of parallelism, so α, is 2) nodes from its 0-bucket.
  Assuming that there are two nodes in the bucket ($\texttt{n}_1$, $\texttt{n}_5$), we insert them into a list ordered by the distance (XOR metric) between the inserted node and $\texttt{s}_\texttt{node}$. So we obtain `k-closest = [`$\texttt{n}_5$`, `$\texttt{n}_1$`]`. Now, considering the *closest node* as the first node in `k-closest`:

- **Make the recursive step of the node lookup, until the closest node is no longer updated:**
  Selecting the only 2 nodes in `k-closest`, we send them a `FIND_NODE`, updating `k-closest` by inserting the new retrieved nodes. Assuming that the calls have the following results:

  - $\texttt{FIND\_NODE(key)} \rightarrow \texttt{n}_5 \Longrightarrow \texttt{n}_4, \texttt{n}_6$
  - $\texttt{FIND\_NODE(key)} \rightarrow \texttt{n}_1 \Longrightarrow \texttt{n}_4, \texttt{n}_5$

  Now `k-closest` = `[`$\texttt{n}_4$`, `$\texttt{n}_5$`, `$\texttt{n}_6$`, `$\texttt{n}_1$`]`. The second iteration of the recursive step, consist of querying the two nodes that has not been queried yet (as before, we're assuming which are the results):

  - $\texttt{FIND\_NODE(key)} \rightarrow \texttt{n}_4 \Longrightarrow \texttt{n}_5, \texttt{n}_6$
  - $\texttt{FIND\_NODE(key)} \rightarrow \texttt{n}_6 \Longrightarrow \texttt{n}_4, \texttt{n}_5$

  `k-closest` (so the *closest node*) has not changed, so we can go towards the next step:

- **send** `STORE(key,`*`some_value`*`)` to $\texttt{n}_4$, $\texttt{n}_5$ which are the α closest nodes to `key`, so they are the responsibles for the content.

2.2 JOIN

N.B. *The routing table should have 8 rows because the identifiers are 8 bit long. But I'll show only the first 4 rows, because the given tree has depth 4, so the other 4 rows will be empty.*

N.B. *The routing table of all nodes are assumed to be filled with random (but consistent) nodes, so also the retrieved nodes to the* `FIND_NODE` *RPCs are randomic.*

Calling `new` the node that joins the network and `boot` the bootstrap node, the joining process follows these steps (the filling of the buckets of `new` is shown step by step):

- **`new` sends `FIND_NODE(new)` to `boot`.** `boot` will now add `new` to its routing table, so `new` is finally in the network. Assuming the answer to the `FIND_NODE` is:

  $$\texttt{FIND\_NODE}(\texttt{new}) \rightarrow \texttt{boot} \Longrightarrow \texttt{n}_{13}, \texttt{n}_8$$

  We insert $\texttt{n}_{13}$ in the 2-row and $\texttt{n}_8$ in the 1-row (assuming that both have successfully replied to the PING sent from `new`):

0	boot
1	$\texttt{n}_8$
2	$\texttt{n}_{13}$
3	

  `new` has also added `boot` to its routing table.

- **`new` has to fill all the rows of its routing table.**
  To do this, for each i-row it generates at random an identifier that will be in the i-row, and sends a `FIND_NODE` to the `k-closest` nodes (k = 2) to the generated ID. If in the i-row there are fewer than k nodes, `new` will chose the remaining ones by using the `XOR` metric, calculated between the generated ID and the peers'IDs:

  - **0-row:**
    - $\texttt{id}_0 = 01101101$
    - $\texttt{FIND\_NODE}(\texttt{id}_0) \rightarrow \texttt{boot} \Longrightarrow \texttt{n}_5, \texttt{n}_6$
    - $\texttt{FIND\_NODE}(\texttt{id}_0) \rightarrow \texttt{n}_{13} \Longrightarrow \texttt{n}_1, \texttt{n}_3$

    The second chosen node is $\texttt{n}_{13}$ because:
    $\texttt{id}_0 \oplus \texttt{n}_8 = 252.$
    $\texttt{id}_0 \oplus \texttt{n}_{13} = 144.$

    $\texttt{n}_5$ is inserted in the 0-row, then `new` pings `boot` which succesfully responds, so it is moved to the tail of the 0-row and $\texttt{n}_6$ is discarded. Now `new` pings $\texttt{n}_5$ which also responds, so it's moved to the tail and $\texttt{n}_1$ is discarded. Finally, to insert $\texttt{n}_3$, `boot` in pinged, but this time it fails to respond, so `boot` is removed and $\texttt{n}_3$ is inserted at the tail.

0	$\texttt{n}_5$, $\texttt{n}_3$
1	$\texttt{n}_8$
2	$\texttt{n}_{13}$
3	

- **1-row:**
  - $\texttt{id}_1 = 10111100$
  - $\texttt{FIND\_NODE}(\texttt{id}_1) \rightarrow \texttt{n}_8 \Longrightarrow \texttt{n}_9, \texttt{n}_{10}$
  - $\texttt{FIND\_NODE}(\texttt{id}_1) \rightarrow \texttt{n}_{13} \Longrightarrow \texttt{n}_7, \texttt{n}_{10}$

  The second choosen node is $\texttt{n}_{13}$ because:
  $\texttt{id}_1 \oplus \texttt{n}_5 = 211$.
  $\texttt{id}_1 \oplus \texttt{n}_3 = 143$.
  $\texttt{id}_1 \oplus \texttt{n}_{13} = 65$.

  $\texttt{n}_9$ is inserted in the 1-row, then `new` pings $\texttt{n}_8$, which fails to respond, so $\texttt{n}_{10}$ is moved to the tail of the 1-row and $\texttt{n}_8$ is removed. $\texttt{n}_7$ is not inserted because $\texttt{n}_9$ responds to the ping, and finally $\texttt{n}_{10}$ which was already in the row, is moved to the tail.

0	$\texttt{n}_5$, $\texttt{n}_3$
1	$\texttt{n}_9$, $\texttt{n}_{10}$
2	$\texttt{n}_{13}$
3	

- **2-row:**
  - $\texttt{id}_2 = 11110000$
  - $\texttt{FIND\_NODE}(\texttt{id}_2) \rightarrow \texttt{n}_{13} \Longrightarrow \texttt{n}_{12}, \texttt{n}_{11}$ (from 2 different rows of $\texttt{n}_{13}$)
  - $\texttt{FIND\_NODE}(\texttt{id}_2) \rightarrow \texttt{n}_{10} \Longrightarrow \texttt{n}_{11}, \texttt{n}_{13}$

  The second choosen node is $\texttt{n}_{10}$ because:
  $\texttt{id}_2 \oplus \texttt{n}_9 = 95$.
  $\texttt{id}_2 \oplus \texttt{n}_{10} = 65$.

  $\texttt{n}_{12}$ is inserted in the 2-row, $\texttt{n}_{11}$ in the 3-row (due to the common prefix between it and `new`). $\texttt{n}_{11}$ and $\texttt{n}_{13}$ are already present in the routing table, so they're only moved at the tail of their respective rows.

0	$\mathtt{n}_5$, $\mathtt{n}_3$
1	$\mathtt{n}_9$, $\mathtt{n}_{10}$
2	$\mathtt{n}_{12}$, $\mathtt{n}_{13}$
3	$\mathtt{n}_{11}$

- **3-row:**
  - $\mathtt{id}_3 = 11000110$
  - $\mathtt{FIND\_NODE}(\mathtt{id}_3) \rightarrow \mathtt{n}_{11} \Longrightarrow \mathtt{n}_{12}, \mathtt{n}_{13}$
  - $\mathtt{FIND\_NODE}(\mathtt{id}_3) \rightarrow \mathtt{n}_{12} \Longrightarrow \mathtt{n}_{11}, \mathtt{n}_{13}$ (from 2 different rows of $\mathtt{n}_{12}$)

  The second choosen node is $\mathtt{n}_{12}$ because:
  $\mathtt{id}_3 \oplus \mathtt{n}_{13} = 59$.
  $\mathtt{id}_3 \oplus \mathtt{n}_{12} = 35$.

  This step doesn't change the routing table, because all the retrieved nodes are already present, even after moving the nodes to the tail of the rows.

The routing table at the end of the joining process is:

0	$\mathtt{n}_5$, $\mathtt{n}_3$
1	$\mathtt{n}_9$, $\mathtt{n}_{10}$
2	$\mathtt{n}_{12}$, $\mathtt{n}_{13}$
3	$\mathtt{n}_{11}$