Monitoring the Interplanetary File System

P2P and Blockchain Mid-Term Assignment

Luca Roveroni - 606803

4 April 2020

Brief introduction

This assignment consists of two exercises, the first one monitoring the IPFS for at least two days and develop some statistics about the connected peers. In the second exercise we have to answer two questions on how a Kademlia's DHT should behave doing some operations on it. The technologies used to carry out this mid-term assignment are: JavaScript (NodeJS) as main application to retrieve information through the IPFS HTTP APIs and a Python script to read and process all the informations written on a JSON file and finally produce some plots.

1 IPFS Swarm Monitoring

As told before, the NodeJS application reads all the informations about the current swarm of peers and store them in a JSON file. All the data in the output file are stored with a custom structure, defined as follows:

```
{ "Peers": [{
            "Peer": "QmWBNHV8UCAKV1zw9mzrRdHLJy9HmDoaKDFGfTXbDiieaW",
            "Latency": "233.195187ms",
            "Country": "US",
            "City": "Los Angeles",
            "Timestamp": "Sat, 04 Apr 2020 16:46:02 GMT",
            "IPv4": "97.64.19.136"
        }, {
            ...
        }, ...
]}
```

It's basically a JSON array of multiples JavaScript objects where every element of the array contains specific info about that peer. The "Peer", "Latency" and "IPv4" fields are the default values given by the IPFS deamon APIs. Instead the "Country" and "City" values are calculated using a third-party library that invokes an external service to resolve the location of a given IP address.

1.1 NodeJS Script: Invoking IPFS HTTP APIs

Here below the code of the NodeJS application:

```
// Required modules
const superagent = require('superagent');
const geoip = require('geoip-lite');
const fs = require('fs');

// Invoke IPFS Daemon APIs
function readSwarm() {
    const ipfs_path = 'http://127.0.0.1:5001/api/v0/swarm/peers?latency=true';

    try {
        // Call the IPFS API
        const req = await superagent.post(ipfs_path);

        // Check if some peers are read
        if (!req.body.hasOwnProperty('Peers')) {
            console.log('At the moment there aren\'t any peers');
        } else {
            req.body.Peers.forEach(peer => {
                // Retriving the peer's IPv4 address
                const peer_ip = peer['Addr'].split('/')[2];

                // Check if the IPv4 is a valid one
                if (checkIP(peer_ip)) {
                    // Invoke the GeoIP library
                    const geo = geoip.lookup(peer_ip);

                    // Add new info about the peer
                    peer['Country'] = geo.country;
                    peer['City'] = geo.city;
                    peer['Timestamp'] = new Date().toISOString();
                    peer['IPv4'] = peer_ip;

                    // Delete some useless data
                    delete peer['Addr']
                    delete peer['Streams']
```

```
                    delete peer['Direction']
                    delete peer['Muxer']
        } } );

        // Open and read the JSON file and update the array of peers
        fs.readFile('swarm_stats_final1.json', (err, data) => {
                var json = JSON.parse(data);
                var newArray = json.Peers.concat(req.body.Peers);
                var output = {
                    "Peers": newArray
                }
                fs.writeFile('swarm_stats_final1.json',
                             JSON.stringify(output),
                             'utf8', cb => {
                    console.log('Written on file!');
                })
            })
        }
    } catch (err) {
        console.error(err);
    }
}

// Check IPv4 syntax
function checkIP(IPv4) {
    if (/^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
        \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
        \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
        \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/.test(IPv4))
        return true;

    return false;
}

function main() {
    // Recall the function every hour
    readSwarm();
    setInterval(readSwarm, 1000 * 60 * 60);
}

// Start main program
main();
```

1.2 Python Script: Process data and plots

To generate the plots and charts I wrote a Python script to read and process all the data inside the JSON file produced by the NodeJS application. I used the Matplotlib for the plots and print some basic statistics on console. Anyway, all the code is available on Github: https://github.com/LucaRoveroni/IPFS-Peers-Monitoring

1.3 Final plots and charts

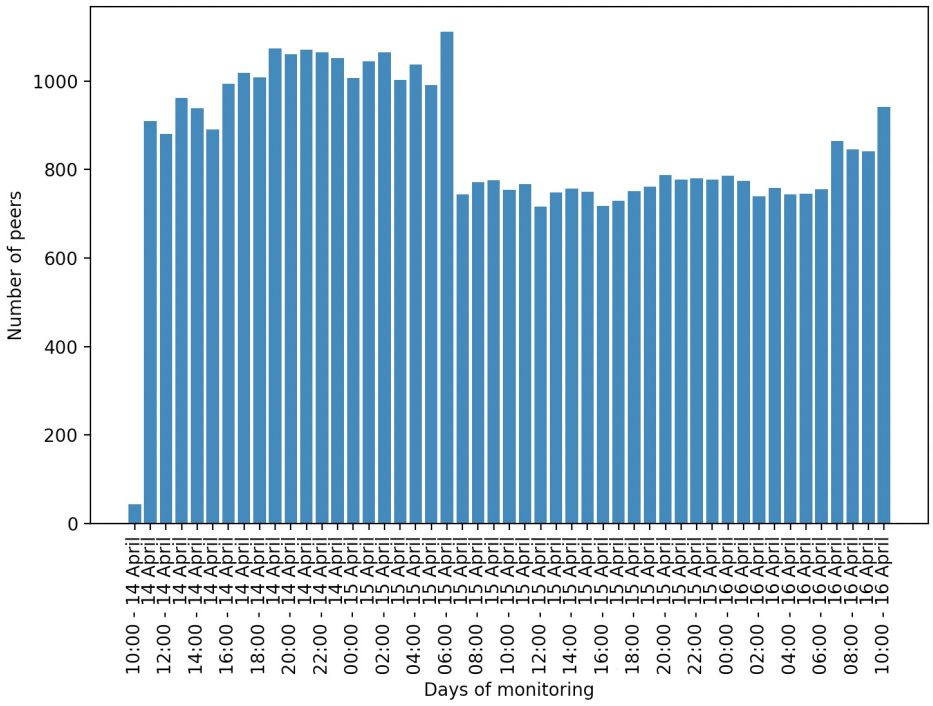
Now the list of statistics derived from the data stored in the JSON file:

1. Total number of peers in the two days of monitoring
2. Average global latency
3. Number of unique peers reached
4. Day behavior of connected peers for the two days
5. Average hour behavior of connected peers between the two days
6. Number of peers per country and city
7. Average latency per country
8. Latency distribution
9. Best and worst latency peer

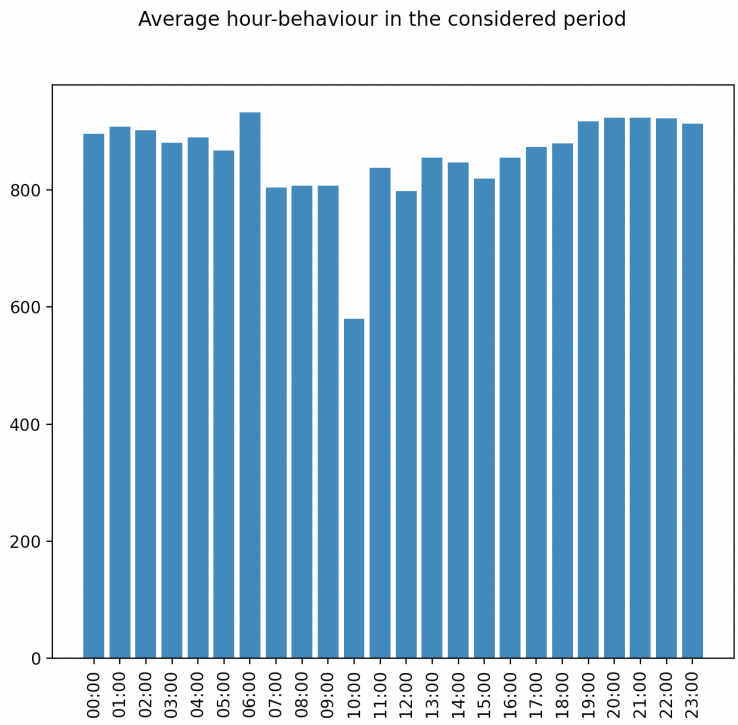
Obtained results:

1. Summing up all the peers in the two days of monitoring my IPFS node reached 41876 peers
2. The average global latency obtained by all the peers with an assigned value is 560ms and a total number 14283 peers with n/a latency
3. Another important statistics is the real number of unique peers connected to the IPFS network, since some peers still online for hours or even days. This cause a multiple read of peers and cause some issue in the monitoring. That's why I checked the real number of peers read multiple times and I obtain: 9904 unique peers and 31972 peer read different times. This means that there are actually a lot of peers connected to the network and probably always available (this is the case of servers).

4. As we can see from the bar plot below, there are some ups and downs values but with a generical average value of 800 peers per hour. Starting from the first monitored hour, where the IPFS node is started, it only reaches a small set of peers ($\tilde{4}0$), but already one hour after, a huge number of peers were reached. Looking in more detail the plot it's easy to see that there aren't considerable differences between the day traffic and during the night.



5. Another bar plot to confirm the generic behavior of peers during the 24 hours is calculating the average between the values of the two days. It follows no significant dissimilarity between day and night (10:00 am value is due to the first ipfs daemon start causing a reduce of the average).

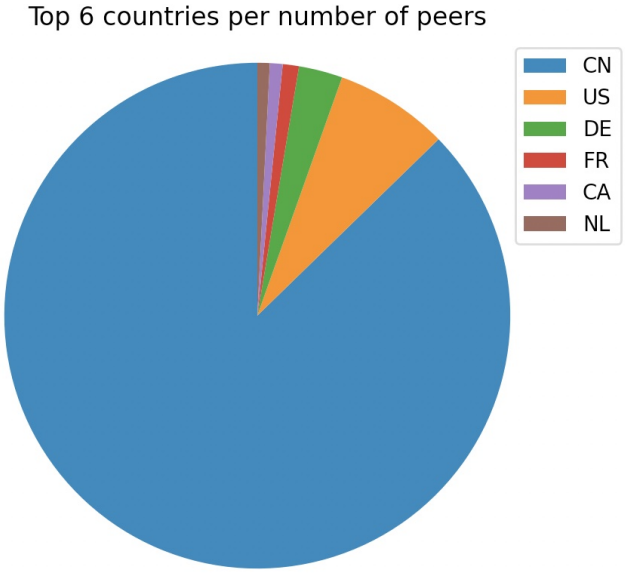


6. In the lists below are listed all the found countries and cities with their number of peers. To retrieve all the peers' locations I used a JavaScript library called GeoIP Lite. Sometimes the library wasn't able to determine the location of a peer using its IPv4 address, but they still a big subset of the entire dataset. In the next page there are two pie charts representing the first 6 contries and cities for number of active peers.

List of peers per county:

China:	34629	Japan:	123	Vietnam:	54	Panama:	19
USA:	2886	South Korea:	108	Ireland:	51	Norway:	17
Germany:	1110	Switzerland:	105	Czechia:	39	Malaysia:	16
France:	411	Australia:	102	Hong Kong:	36	Cuba:	15
Canada:	324	Singapore:	99	Poland:	32	South Africa:	14
Netherlands:	313	Italy:	87	Taiwan:	29	Estonia:	14
Russia:	230	Finland:	83	Denmark:	26	Indonesia:	13
India:	167	Sweden:	74	Ukraine:	21	Malta:	12
Brazil:	133	Spain:	72	Thailand:	20	Belarus:	11
UK:	132	Mexico:	64	Romania:	19	Lithuania:	11

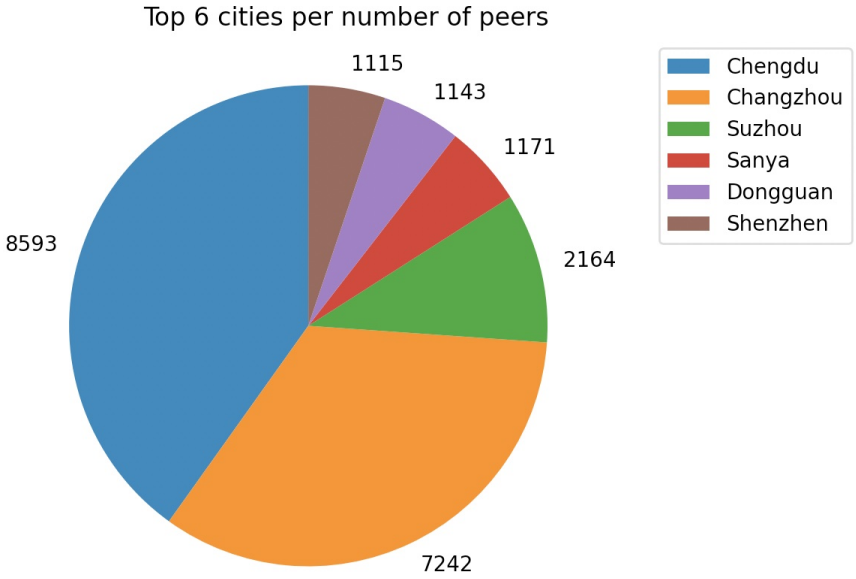
Turkey:	10
Slovenia:	10
Latvia:	9
Dominican Republic:	7
Austria:	7
Bulgaria:	6
Argentina:	6
Colombia:	6
Israel:	6
Philippines:	5
Rwanda:	5
Belgium:	5
Tonga:	5
Tonga:	4
Hungary:	4
New Zeland:	3
Bangladesh:	3
Iran:	3
Luxembourg:	3



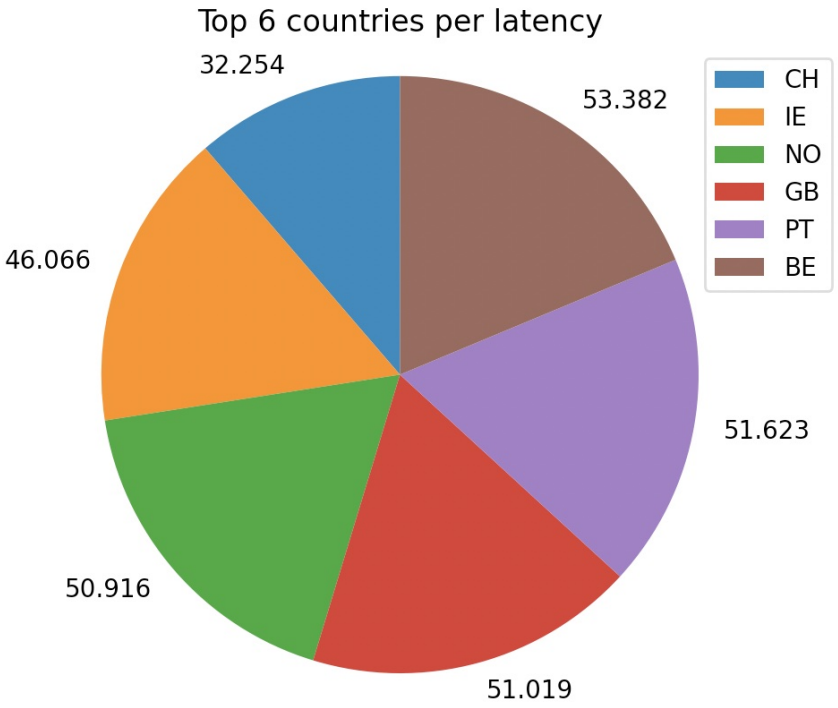
List of peers per city:

Beijing:	1079
Chengxiang:	437
Kunshan:	402
Yulin:	382
Shanghai:	342
Chongqing:	318
Frankfurt:	318
Chanshan:	310
Ashburn:	300
Huilong:	285
Hangzhou:	282
Boardman:	241
Leshan:	195
Amsterdam:	190
Los Angeles:	168
Haikou:	166
Handan:	163
Parsippany:	155
Mountain View:	154
Jiangyin:	150

and many more...



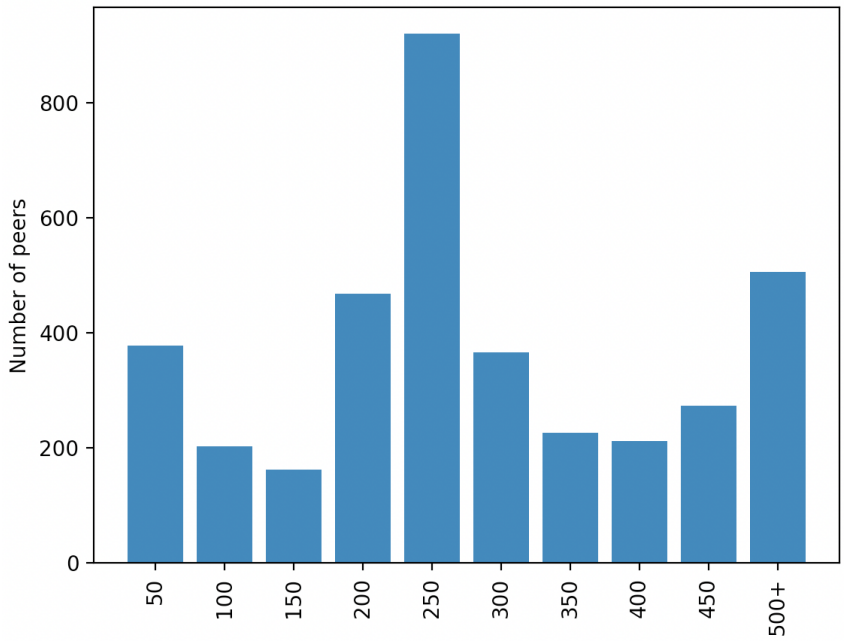
6. Last but not least the pie chart of the average latency for each country that has been found by the GeoIP library. It's easy to see that starting the monitoring from Italy the lower latency values come from the adjacent countries, but not always.



Average latency other countries:

Czech Republic:	60.194	Slovenia:	73.623	Slovakia:	120.02
France:	60.762	Georgia:	83.444	Austria:	128.267
Poland:	61.667	Romania:	87.929	Dominican Republic:	166.711
Lithuania:	62.178	Turkey:	97.385	Sri Lanka:	174.062
Finland:	65.52	Armenia:	100.577	Bangladesh:	195.78
Italy:	66.009	Denmark:	110.02	Panama:	208.344
Belarus:	66.17	Netherland:	118.072	Costa Rica:	209.931
German:	68.655	Estonia:	118.178	and many more...	

7. Distribution of latencies from 50ms to 450ms, where the 36% of the peers have a latency of 200/250ms approximately and a 13% with more than 450ms. Merging together these data we obtain that half the peers have a high latency! This could be explained because having a good internet connection is highly probable to reach more peers than someone with a slower connection an this cause a discover of a larger number of peers, mostly peers far away from me, as result a high global average latency.



8. The best peer according to the monitored latency was:
   Peer ID: 'QmaaQHhykT1XEqNt2X2peVXegiMF2jEhSVWjLGZ3aQzFmZ'
   Latency: 16.125ms - Country: France

   Instead the worst peer was:
   Peer ID: 'QmbhtyYqqEKSv566eUuYPiQK6sCW4QbnieajmgzwH8PQ9G'
   Latency: 47154.255ms - Country: China

2 The Kademlia DHT

- Node 11111101 inserts the content identified by the identifier 01000001. Which node is responsible for that content?

  Kademlia protocol defines an exact procedure to follow when a content is inserted by a node. First, it's generated the hash value of the content, which in this case is 01000001 and then it's assigned to the closer node. So, the closer node is searched checking the k-bucket with the maximum sharing prefix in the local node's bucket-list. Then, the node will perform, recursively, a look-up operation on α peers nodes in that k-bucket. Once the closer nodes are found, it's invoked a STORE RPCs on them to save the key/value pair. According to the given snapshot of the Kademlia overlay, the node 11111101 will performs a look-up on its most-left k-bucket (in this case representing the left subtree starting from the root). In this case, the two nodes that will take care of this content are the 4th and 5th starting from left. After retrieved the closer set of peers to the content id, it's performed the STORE of (Key = 01000001, Value = A block of that content) on them.

- Node 11010101 joins the network. Node 00110011 is the bootstrap peer that joining node uses to join. Describe the join operation step by step, describing how the buckets of the joining peer are filled over time.

  The node 11010101 to start joining the network must know a starting node (called bootstrap node) to call and retrieve some informations about the network. To do so, the first operations is to perform a FIND_NODE of itself to the bootstrap node, letting the joining node to have a first understanding of its neighbours peers and fill its k-buckets. During this operation the bootstrap node invokes other peers before returning the result to the joining node and this cause a knowledge to the other nodes in the network of the new peer. Now that the new node is actually a member of the network, it stats filling up its remaining k-buckets performing FIND_NODE RPCs to some random generated IDs that belong to the remaining k-buckets. In this case, after generating a possible ID, the peer will send a FIND_NODE(generated_ID) to the α known peers in the corresponding k-bucket and receive some closer peers to it. Now the sender peer try to insert those responses into its k-bucket applying a specific algorithm to maintain the least recently contacted nodes in the first positions: If the received node already exists in the k-bucket it is moved to the tail otherwise if the k-bucket is not full, the received node is inserted at the tail of the list. If the k-bucket is full, it will ping the least recently seen node in that k-bucket and if it responds, the received node is discarded an the contacted node is put at the tail of the list otherwise we swap the two nodes.