Università di Pisa
Dipartimento di Informatica

Peer to Peer Systems and Blockchain

Student: **Samuele Intaschi**
ID: **523864**

Sharing content in IPFS: monitoring the bitswap protocol

1 Introduction

IPFS is the acronym for InterPlanetary File System and it is a protocol and peer-to-peer network for storing and sharing data in a distributed filesystem.
This project aims to monitor some metrics while downloading a content in IPFS, in particular the behavior of the Bitswap protocol, used by IPFS for content discovery and block exchange, and what are the contributions of each peer of the swarm that contributes to the download.

2 Design Choices

There are mainly two libraries that implement the functionalities of IPFS, *js-ipfs*[1] and *go-ipfs*[2], and the choice fell on the second which appears more complete. This library offers a command line interface and exposes an HTTP API that allows us to control the IPFS node from remote and to have informations about its state. Due to the simplicity of making graphs, working with HTTP requests and exposing useful endpoint for a webpage via Flask in Python, it was chosen to write a Python script that uses the API exposed by *go-ipfs*..
There are various metrics to monitor in the download of a content in IPFS, in particular about the Bitswap protocol, and several ways to do it. It was chosen to realize a real-time monitoring, instead of a report when the content download is finished, because the downloadable content could be very large and the download could take several hours, so it was thought that would be more useful to dynamically track the status of the monitored node without waiting for the download to finish. Monitoring during download also allows us to analyze the actual performance of the monitored node, like the actual bandwidth used to download the content.
The downloadable content can be a single single file, but it can also be a folder and it is possible, via IPFS commands, to first know the files that make up a folder and then download them one by one, to have the statistics for each individual file. The initial idea was to do it this way,

[1] `https://github.com/ipfs/js-ipfs`
[2] `https://github.com/ipfs/go-ipfs`

but the decision to perform real-time monitoring (and the lack of documentation on the results obtained from the HTTP requests of *go-ipfs*) forced to avoid it and to consider the metrics for the entire content, even if it was a folder, because otherwise the number of HTTP requests would have been very large and would have grown very quickly, taking too long to handle.
Finally, a simple web user interface is used to enable a user to start the download and see the results.

3 Files

The files that make up the project are:

- `templates folder`: contains the HTML pages used to interact with the user, which are:
  - `index.html`: this is the index HTML page which contains a field that the user can fill with the CID of the content that he is looking for.
  - `plot.html`: this is the HTML page that the system uses to show the results of the monitoring.
- `forms.py`: this is the description of the forms used to pass the CID of the content that the user wants to download from the HTML page to the Python application.
- `flask-app.py`: this is the whole logic of the application.
- `requirements.txt`: this is the the application requirements list.

4 Development Environment

As it was said in the section 2, Python was used as programming language for the application and *go-ipfs* for the IPFS functionalities, accessed via its HTTP API, thanks to the *requests*[3] library. To know the location around the globe of the IP addresses it was used used an HTTP API [4], which is not very accurate, but unfortunately most of these services are paid. To realize the plots it was used *plotly*[5], and to create the dataframes used by *plotly* it was used *pandas*[6]. The webpages showing results are written in HTML and kept up to date with a Javascript script. These pages use the endpoints exposed by the Python application thanks to *Flask*[7], a micro-framework written in Python. The project was entirely developed on the Windows 10 64bit operative system.

5 Implementation and Analysis Performed

All IPFS commands mentioned in this section are executed through the HTTP API exposed by *go-ipfs* and the graphs presented, unless otherwise specified, were created at the end of the download of the sample content with CID `QmNvTjdqEPjZVWCvRWsFJA1vK7TTw1g9JP6we1WBJTRADM`, which consists of "IETF RFC Archive" (it can be found on `https://awesome.ipfs.io/datasets/`) and is 500MB in size. Unfortunately it is possible to correctly monitor the download of a single content at a time, because the features of *go-ipfs* do not cover the possibility of having values for each file or content that the node download, but only for the entire lifecycle of the IPFS node, for example the number of bytes received from a peer are counted since the daemon was started. The values related to the current content download are obtained by the Python program, which can not classify them if multiple content is downloaded at the same time.
The behavior of the application can be divided in the phases described in detail below.

[3] `https://docs.python-requests.org/en/master/`
[4] `https://www.hostip.info/use.php`
[5] `https://plotly.com/`
[6] `https://pandas.pydata.org/`
[7] `https://flask.palletsprojects.com/en/1.1.x/`

5.1 Download of the content

The user can specify the CID of the content he wants to download through the HTML web page of the application, then it is passed to the Python application, which starts the download starting a thread which performs an `ipfs get <CID>` request. There is needed of this because otherwise the whole application hangs waiting for a response, which would come when the content is fully downloaded, so instead a thread waits for the response while the rest of the application deals with the monitoring. Before to start the download this thread saves the actual state of the IPFS node, so that the system can ignore any bytes or blocks that peers have previously exchanged with the monitored IPFS node which are related to previous downloads, and can subsequently start a new count.

5.2 Real-time Monitoring

While a file is being downloaded, the HTML webpage, thanks to Javascript, periodically sends requests for updates to the Python script. These updates consist of json representations of the graphs to show. When the Python application receives such a request, perform these steps:

1. **Update the list of collaborating partners**
   The system sends an `ipfs bitswap stat` request, from which it takes the bitswap partners, then for each of them it sends an `ipfs bitswap ledger <peerId>` query to find out if this peer has ever exchanged blocks with the monitored node until now, if not it is ignored, else the system checks if any blocks have been swapped since the current file download was started. For these interesting peers, the location in the globe is searched, as well as all the information present in the ledger is saved. To find the location, first it is performed a `ipfs dht findpeer <peerId>` to know the addresses associated with the multiaddress of the peer, then IPv6 addresses, private addresses and DNS addresses are ignored and in general only one IPv4 address remains that is submitted via an HTTP API to an online service that communicates the country where the address resides.

2. **Create the bytes received and blocks exchanged plot**
   At this point, with the information retrieved from the ledger and thanks to the functionalities of the *Plotly* library, it is possible to create two graphs which indicates the bytes received (figure 1) from each collaborating peers and the blocks exchanged (figure 2) so far with them.

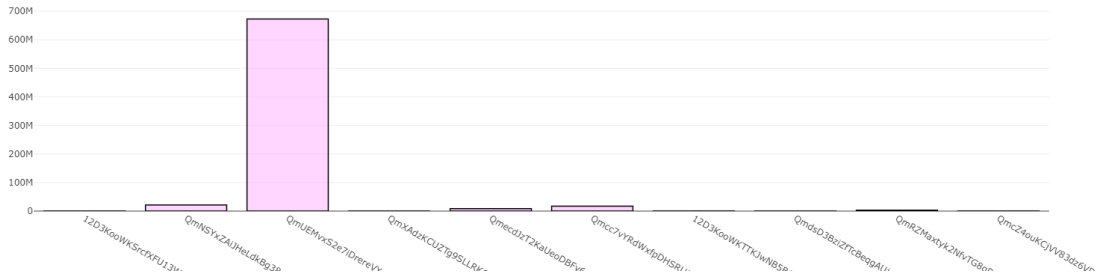

Figure 1: Bytes received from each collaborating peers.

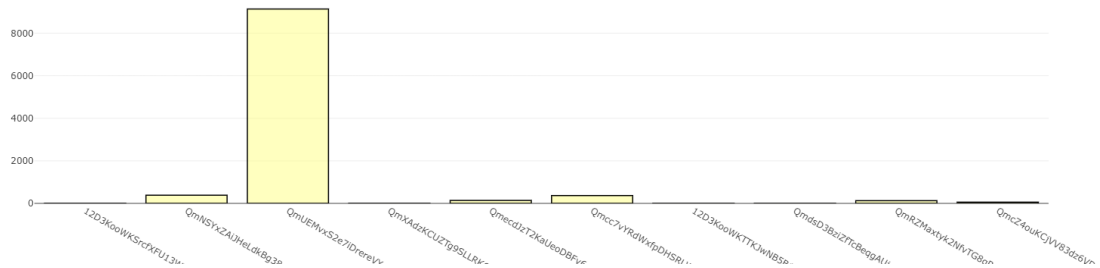

Figure 2: Blocks exchanged with each collaborating peers.

In the figures 1 and 2 it is possible to see that most of the blocks that make up the content come from a few peers. Furthermore, it is possible to observe that the number of exchanged blocks corresponds more or less to the number of bytes received, in fact from the the data it emerges that the blocks that the monitored node sends are negligible compared to those it receives, and this is also evident from the outcoming bandwidth that is negligible compared to the incoming bandwidth. Bytes sent and outcoming bandwidth data are not displayed in the graphs because they don't characterize the download of the content, but is printed to standard output when the application is running. These facts can also be observed in the figures 3 and 4, which represent data taken during the download of `QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D` (Project Apollo Archives).

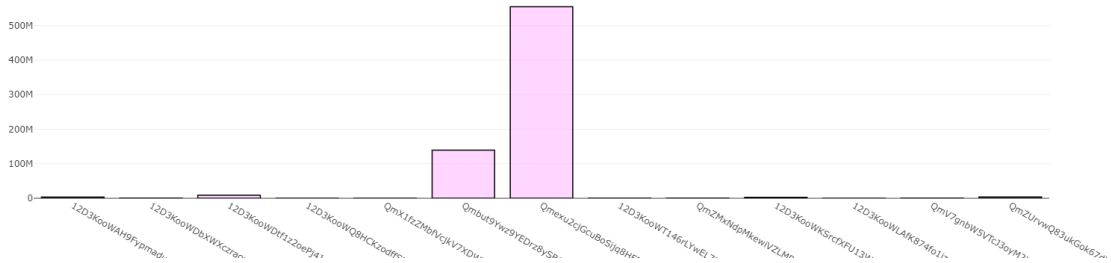

Figure 3: Bytes received from each collaborating peers, detected during the download of `QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D` (Project Apollo Archives).

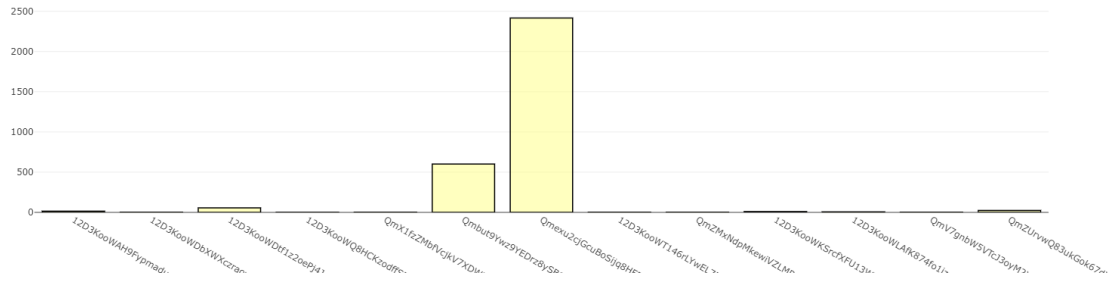

Figure 4: Blocks exchanged with each collaborating peers, detected during the download of `QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D` (Project Apollo Archives).

3. **Create the distribution around the globe**

When information about the collaborating peers is retrieved, the country each peer is from is also included. With this information another feature of the *Plotly* library is used and a visual distribution of the peer on a world map (figure 5) is created. Unfortunately for some peers the free online service that it is used is not able to retrieve the exact country, sometimes only the continent, so these addresses, which are few compared to the others, are ignored by the system.

Figure 5: Distribution of the peers around the globe.

From the figure 5 it emerges that the peers that send this content to the monitored node come from European Union, in particular from Germany (4), Finland (1) and Norway (1).

Figure 6: Distribution of the peers around the globe, detected during the download of `QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D` (Project Apollo Archives).

In the case of figure 6 this content comes to the monitored node from the European Union (Italy, Slovenia and Norway), from the United Kingdom and from the United States.

4. **Create the bandwidth plot**
   Since the monitoring is done while the file is being downloaded, it is also possible to measure the performance, specifically by seeing the actual bandwidth used by the node to download the content. To make this the system sends an `ipfs stats bw` request to the node. This command returns some useful information about the incoming and outcoming bandwidth used by the node, from which the actual incoming bandwidth is taken. The system associates a timestamp with this data and calculates the average value through a mathematical operation. Then these three values are added to a data structure, and with all the saved values a time series graph is created, which indicates the trend of the bandwidth over time, as shown in figure 7. In the figure the last measurement is 0 because it is performed when the downloading is already finished.

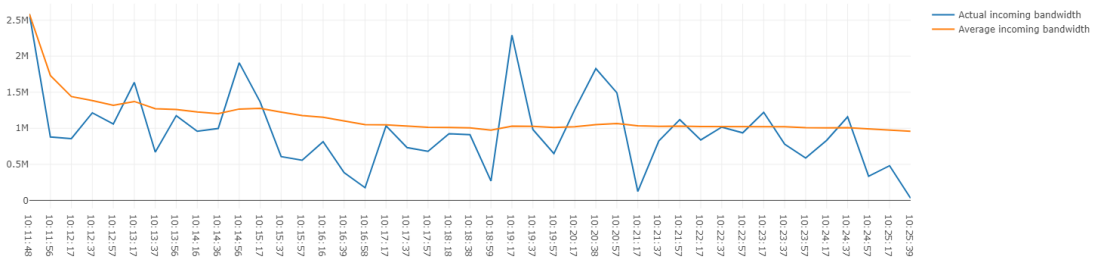

Figure 7: Actual and average bandwidth used by the IPFS node.

5. **Create the collaborating/total peers graph**
   Finally, the system uses the number of total bitswap partners and the number of collaborating partners, associated with a timestamp, to build with *Plotly* a time-series graph which indicates the trend of the total and collaborating partners (figure 8).

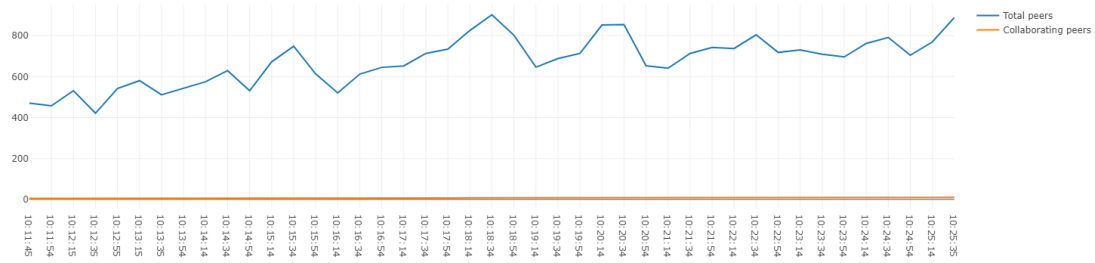

Figure 8: Total partners compared to collaborating partners.

It is probably the least useful graph, but by the figure 8 it is possible to notice that the total number of Bitswap partners changes very frequently, while the number of peers that collaborate with the monitored node remains more or less the same, and it is much smaller than the other.

5.3 Cleanup

This application was designed for statistical purposes only, so there is no point in keeping downloaded blocks when closed, also because they can take up a lot of disk space. For this reason when a signal `SIGINT` is received, that is the right way to stop the application, the system sends an `ipfs repo gc` request to the node that trigger a complete cleanup of the data present in the monitored IPFS node.

6 Usage

All application requirements can be installed with the command:

```
pip install -r requirements.txt
```

which installs all packages specified in the file `requirements.txt`. This operation requires to have Python and its package manager *pip* installed, to install them see `https://realpython.com/installing-python/`.
The application requires also that an IPFS daemon is running on the host to be used, and this can be started with the command:

```
ipfs daemon
```

and this operation requires to have *go-ipfs* installed, to install see `https://github.com/ipfs/go-ipfs#install`.
With a running daemon it is possible to start the application via the command:

```
python flask-app.py
```

or `flask run` (this only if the environment variable `FLASK_APP` is set to `flask-app.py`).
When the application is running it is possible to browse to `http://127.0.0.1:5000/` with any browser to see the web page shown in figure 9, which contains a blank field, which can be filled with the CID of the searched content.

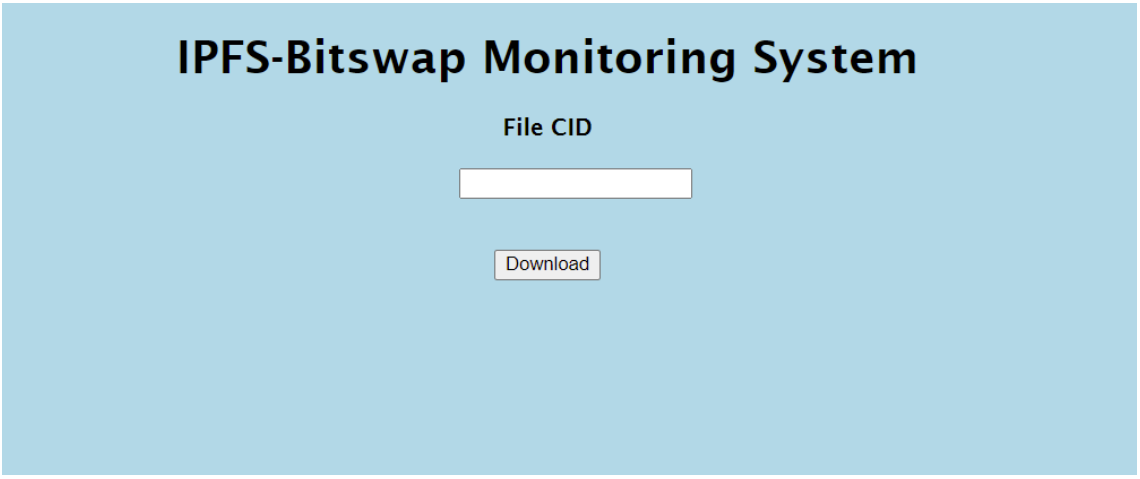

Figure 9: Index page presented by the application.

Then a page is presented that says that the application is loading the graphs (figure 10), to allow the system time to start the download and perform the first surveys.

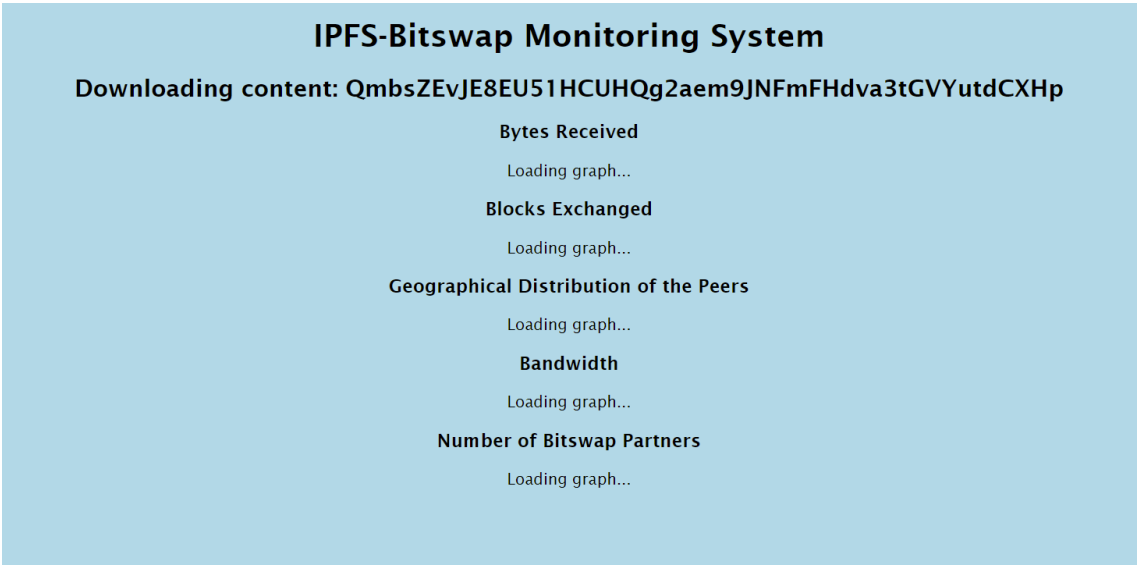

Figure 10: The graphs are loading.

When the graphs is loaded, they are shown on the page (figure 11) and periodically updated when new surveys are performed. Thanks to *Plotly* it is also possible to interact dynamically with the graphs, zooming, panning and cropping them to see more details.

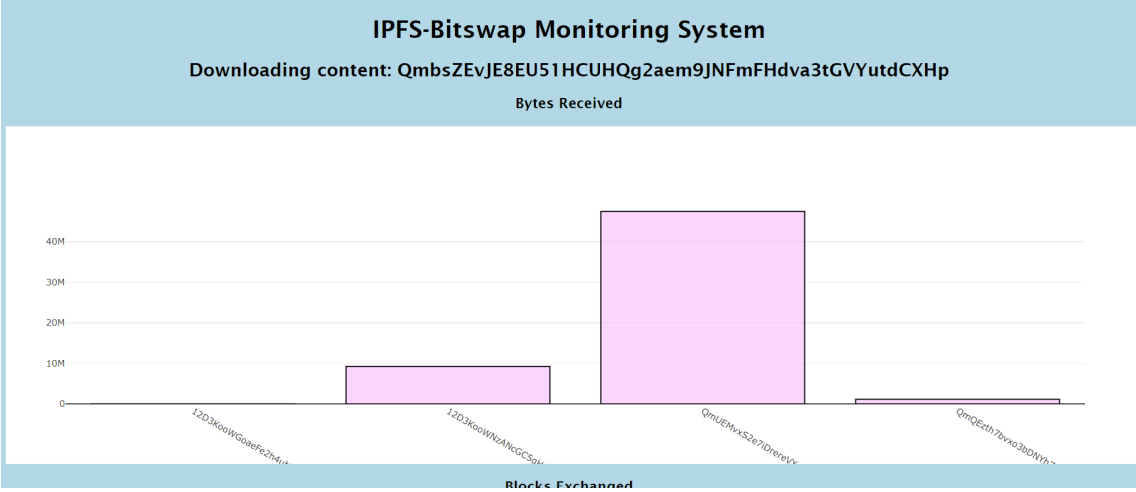

Figure 11: The graphs are finally shown.

When the download is complete, a link to the homepage appears (figure 12), that allows to return to the homepage and download more content from IPFS.

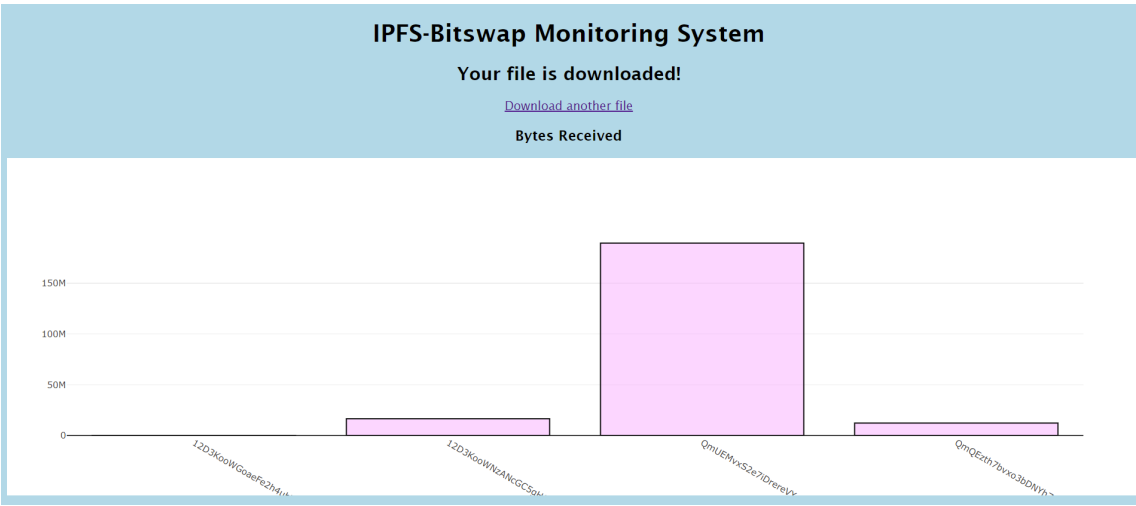

Figure 12: The download is complete, it is possible to start another one.

The application can be closed with a `SIGINT` signal, which can be activated with the `Ctrl+C` key combination, and this trigger the cleanup of the blocks downloaded, as explained in section 5.

7 Questions

This section contains the answers to the questions of the second part of the assignment.

7.1 Describe at least two differences existing between the classical Kademlia protocol and the version of the protocol used by IPFS.

In IPFS, the DHT is used as the main component of the content routing system and maps the content that a user is looking for to the peer that is storing it, in practice to know *who* has *what*. The DHT protocol used by IPFS is a secure key-based protocol based on Kademlia, in particular on S/Kademlia which has a set of security improvements with respect to Kademlia. The first improvement is the limitation of free node identifier generation, using Proof of Works

(PoW) implemented through cryptopuzzles. A PoW is a mechanism that consent to prove to a peer that a certain amount of computational resources has been utilized, for example by solving a cryptopuzzle that requires a considerable effort to be solved and much more less to be verified. The complexity of the puzzle is inversely proportional to the number of people that solves it, the more people solve the puzzle, the harder it gets, and vice versa, so the complexity depends on the average time to produce a valid result. This mechanism is a deterrent for attacks like DOS and SPAM and makes Sybil (the attacker generates a huge number of identifiers gaining a disproportionate control over the network) and Eclipse (the attacker uses multiple identifiers to cut off the traffic that goes to and from a particular node) attacks, common in peer to peer networks, difficult to carry out.
Another improvement of S/Kademlia with respect to the original Kademlia protocol is the parallel lookup over disjoint paths, in order to ensure that honest nodes can connect to each other in presence of a large number of dishonest nodes, otherwise an attacker could reroute packets into a subnet of adversarial nodes.
These are the main differences between S/Kademlia and the original Kademlia, but the protocol used by IPFS implements another extension to the original Kademlia protocol, a distributed sloppy hash table (DSHT) based on Coral CDN, that improves the performance and scalability of content retrieval. This improvement resolves two problems of the classical DHT protocols, the *Hot Spots* and the *Data Locality*.
The first is when a single node is responsible for a popular URL, because of this it stores a huge list of node references and quickly becomes overloaded. With the DSHT each node stores only a maximum number of values for each key, and the data is stored in the responsible peer only if it is not full, else the data is stored in the first free node found on the way to the destination, traveled in reverse. When a node search a content should find the key along the way to the destination, but if it does not find the key there, it goes to the destination. If a content is popular probably many nodes have a copy of it.
To solve the *Data Locality* problem, the nodes are clustered and each cluster contains the peers that have an RTT between them that is below a certain limit. The number of clusters is a parameter of the system and all nodes have the same identifier in all clusters. The level-0 cluster covers the full network and the nodes in the level-i cluster are a subset of the nodes in the level-$i-1$ cluster. The routing starts at the highest level, which clusters the closest nodes, and continues with the lowest level if it does not find the searched key. This mechanism does not increase the lookup time, which remain the same of the routing at level 0.

7.2 Describe the main advantages of the distance metric used by Kademlia with respect to other distances, like the ring distance of Chord.

The distance metric used by Kademlia is the XOR metric, which represents the distance between two objects and it is implemented through a bitwise XOR operation on the object identifiers, interpreted as unsigned integers. With this metric the closest nodes are those which have the longest common identifier prefix, and it is not relative to the real geographical distance or the numerical distance between peer identifiers.
The XOR metric is symmetric, so Kademlia can learn contacts from ordinary queries it receives and this helps in building the routing table. In fact, the symmetric metrics enables each node to enrich its routing table through the query, because $distance(A, B) = distance(B, A)$, so the set of nodes B with small $distance(A, B)$ is the same as the set of nodes B with small $distance(B, A)$, so a node is more likely to receive lookup messages from nodes that are in its routing table. The asymmetric distances does not allow this, and the metric used for example by Chord is asymmetric because it is based on the fact that there is no beginning and no end, the identifiers space is like a ring, so $distance(A, B) = B - A$ for $B \geqslant A$ and $distance(A, B) = B - A + 2^N$ for $B < A$ (in general $distance(A, B) = (B - A + 2^N) \ mod \ 2^N$). In this way if a node A receives a query from B, B has in its table a reference to A, but A may not have a reference for B, so A can not, in general, exploits the information included in the query.

The XOR metric is also unidirectional, because for any given point x and distance $\Delta > 0$, there is exactly one point y such that $distance(x, y) = \Delta$, and it guarantees that there is a single node at minimal distance with the key. Unidirectionality ensures that all lookups for the same key converge along the same path, regardless of the originating node, so it is possible also caching items along this paths to avoid hot spots, because when a key is popular it is probably cached in many nodes and future searches of that key may hit a cached entry before querying the closest node.