P2P Report

Group “Charlie”
Matteo Brucato, Boris Maguet, Enrique Muñoz

I. INTRODUCTION

The system we decided to implement in the P2P Network course is an unstructured Peer-To-Peer network.
After having created such a Network, the intended use of it would be, from an user point of view, to be able to Find and Get data, in the most efficient way possible (for simplicity, users will be able to look for peer names, such as “P20”). From a peer point of view, the intended use would be to be to have an optimized and stable connection to the Overlay Network.
To present our work, we’ll be led by the implementation choices we made all along the Milestones and the problems we encountered.
In this report, we’ll first present the choices we made in terms of Peer Discovery, which is a crucial step to initiate the network. After the initial discovery, we’ll discuss about the Construction of our Overlay Network -- the global architecture we built. To end, two different Search Algorithms and their efficiency will be discussed and compared.

II. PEER DISCOVERY

II.1. Bootstrapping

As we are making a peer-to-peer network, we have the problem of bootstrapping. This means that at the beginning, when a new peer is intending to connect to a network, it initially has to know at least one other peer that is already part of this network.
For our experiments, we decided to adopt the idea of a “Super Peer”: one peer that all others peers know, and which never leaves the network.
Given each peer knows the location of the “Super Peer”, it is thus trivial to initiate the Discovery Protocol.

II.2. Discovery Protocol

We use the same flooding protocol as Gnutella but, with a slight difference: while in Gnutella answers are forwarded back through the query path, we instead send the response directly back to the source.

The protocol uses two different messages, namely:

- ping: Used to discover peers in the network.
- pong: The response to a ping.

Ping

Used to actively discover peers in the given network. A peer receiving a Ping is expected to send back a Pong to the sender.

The Ping packet includes:

- *sourcepid*: the IP address of the peer that started the discovery process
- *name*: the name of the peer that started the discovery process

- *msgid*: the id of the ping message
- *TTL*: the Time To Live of the message
- *senderpid*: the pid of the last peer that sent the message

Ping algorithm:

```
-----------------------------------------------------------------------
update logging info about sourcepid peer
IF I have seen this message THEN
      RETURN
ELSE
      TTL--
      IF TTL > 0 THEN
            FOR each peer P in plist
                  IF P is the sender of this message THEN
                        CONTINUE
                  forward the ping message to P
      send the pong message to the sourcepid peer
-----------------------------------------------------------------------
```

Pong

The response to a Ping. A peer receiving a Pong message is expected to add the new discovered peer on its *plist* (list of known peers).

The Pong packet includes:

- *pid*: the IP address of the discovered peer
- *name*: the name of the discovered peer

The Pong algorithm is straightforward: it simply adds or updates the newly known peer’s IP address into plist.

III. BUILDING THE OVERLAY NETWORK

We decided to adopt the main ideas of the Gia protocol, in order to build an unstructured overlay network that is capable to adapt itself based on the capacity of the peers. Our protocol utilizes a concept of “satisfaction” similar to that of Gia. In our protocol, a peer is satisfied if the number of its neighbours equals its capacity *nmax*, i.e. when $|nlist| = nmax$. Following this idea, the satisfaction level is defined as $\frac{|nlist|}{nmax}$. Moreover, we decided to implement a 2-way handshake protocol to establish the connections.

We identified five main aspects of the protocol to be addressed:

1. Selecting the peers which we want to connect to: this should change over time, as our level of “satisfaction” increases (or decreases).
2. Responding to connection requests.
3. Reducing the network traffic.
4. Coping with churn.
5. Adjusting inconsistencies in the network.

III.1. Selection of peers

Each peer has an internal routine (running in a separate thread) that checks whether the peer is satisfied or not with the current neighbourhood. If the peer needs to find some more neighbours, then its routine will have to select which peer (among the list of known peers) to ask for connection. Differently from Gia, which utilizes a random selection (filtered by the peers capacity), we decided to implement a selection based on a score. We rank the entire list of known peers (*plist*) and ask to the top peer to which we are

not connected yet. We ask to one peer at a time, and wait for its response before trying to connect to any other. If the peer does not respond within reasonable time, we rank again the peers and make another request.

We rank peers based on the following score:

$$Score_i = \frac{nmax - |nlist|}{nmax} \frac{nmax_i}{MAX - NB} + \frac{|nlist|}{nmax}\left(1 - \frac{nmax_i}{MAX - NB}\right)$$

where $nmax$ is the capacity of the local peer, $|nlist|$ is the current number of neighbours of the local peer, $nmax_i$ is the capacity of the i-th peer, and $MAX - NB$ is the maximum allowed capacity in the entire network. The idea of the score is that it is higher if one of the following conditions is true:

- The local peer is not well-connected (i.e. $|nlist| << nmax$) and the i-th peer has high capacity
- The local peer is well connected (i.e. $|nlist| \approx nmax$) and the i-th peer has low capacity

This models the fact that we want to be greedy at the beginning, but as soon as we get more satisfied, we start being generous to low capacity peers, by actually asking them to join our neighbourhood. This way, a high capacity peer will ask to peers that cover the whole spectrum of capacities, from high (at the beginning), to medium, to low (at the end). This fact has the following nice property: it allows for a driven construction of a network that embodies a reasonable number of high-capacity peers highly connected to each other (which form the backbone of the network), yet allowing every peer to eventually join the network. Another nice property of not using randomness is that our protocol is deterministic (given that we know each peer’s capacity).
Additionally, we implemented a counter of rejections, for each peer the local peer tries to connect to. This is necessary in order to avoid asking the same peer over and over again if we already received some rejections from it. The final ranking function of known peers is as follows:

$$\frac{Score_i}{\sqrt{r_i + 1}}$$

where r_i is the number of rejections the local peer received from the i-th peer. We take the square root of it in order to dampen its contribution to the final score.

III.2. Response to connection requests

The response to a request of connection is handled through the same algorithm Gia uses for topology adaptation. We employed a value of $H = 0$, in order to ensure that every peer with smallest capacity has a chance to enter the network.

III.3. Reduction of the network traffic

As we said, the thread responsible for fulfilling the local peer’s neighbourhood asks to one peer at a time for connection. However, the frequency of requests are dynamically changed based on the level of satisfaction. That is to say, the thread sleeps for a time equals to $(0.1 + 10\frac{|nlist|}{nmax})$ seconds. This has the desirable property that as soon as a peer gets more and more satisfied, it will utilize the network less and less frequently. At the beginning, it will try to establish a connection every 100 milliseconds, and once it is fully satisfied it will check again only every 10 seconds.

III.4. Churn

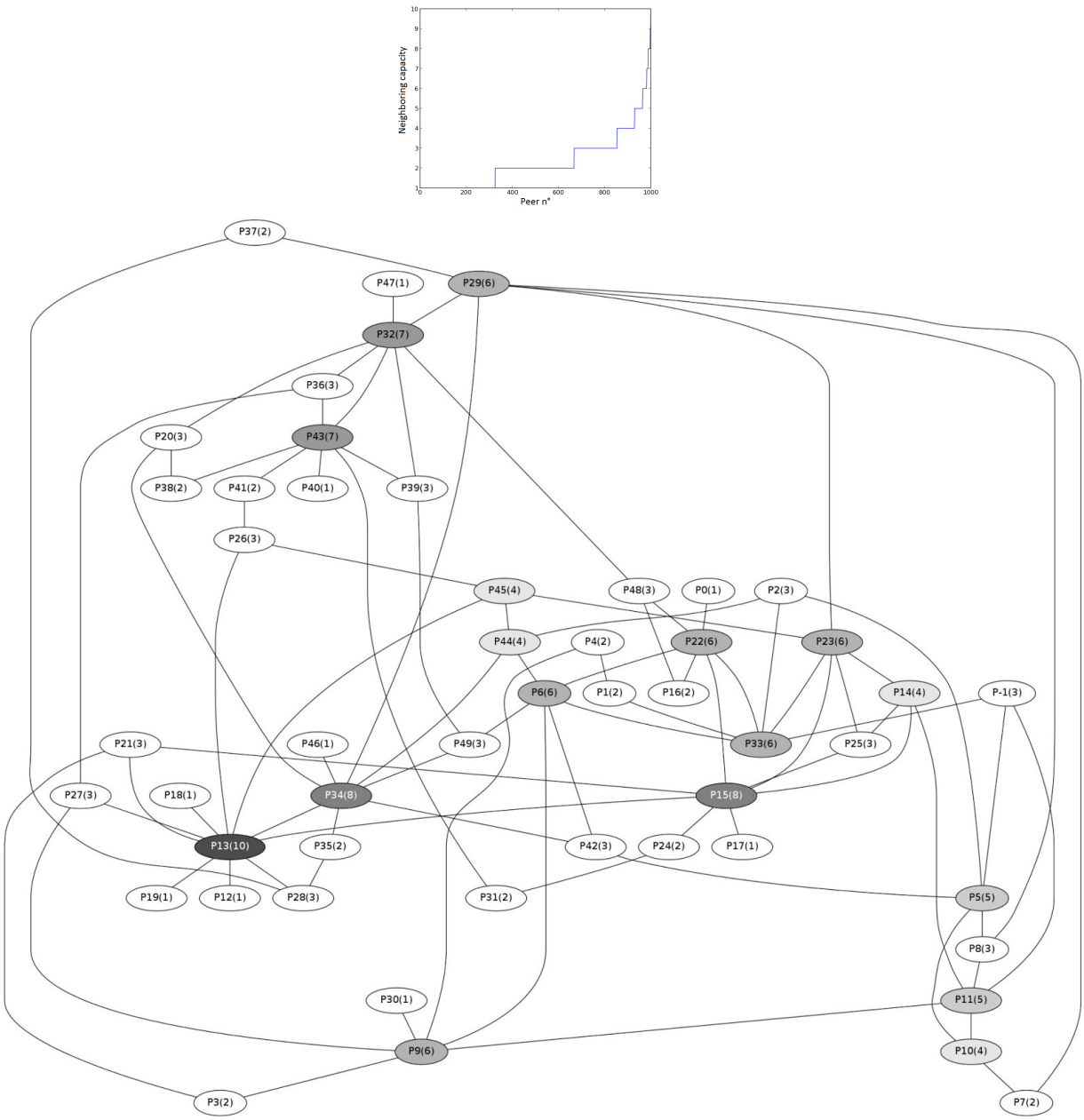
Our implementation deals with churn in the following way: a special thread wakes up every 2 seconds in order to check if there are peers that it considers “probably off”. A peer is considered probably off if the local peer has not seen any message coming from it for at least 5 seconds. For each of these peers, the local peer sends a message “are you still alive?”, and it waits an additional 2 seconds. If after 2 seconds the peer is still in the “probably off” condition, it gets removed from the local peer’s list of peers, as well as from its neighbour list.

III.5. Removing inconsistency in the neighbour lists

Due to the distributed nature of the network, some inconsistency may arise during the lifetime of the network. An inconsistency is when the peer i has peer j in its neighbour list, whereas peer j does not have peer i. This happens for two reasons: (1) we use a 2-way handshake protocol for connection (hence, no acknowledgement), and (2) we drop neighbours without informing them. To cope with this, we implemented this simple mechanism: every so often the local peer sends to each of its neighbours a message “are you still my neighbour?”, and it adjusts its neighbour list accordingly.

III.6. Example

In our experiments, we created 1 Super Peer with $nmax = 3$ and we automatically launched 50 peers whose capacities followed an exponential distribution. The exponential distribution is shown in the following figure, in which 1000 peers have been generated. The figure underneath is an example of a 51 peers overlay network.



Example of 51 peer overlay network (one initial “known peer” and 50 automatic peers), after the network became stable

IV. SIMPLE CONTENT SEARCH

After having created our network in an efficient way, we can now “flood” it !
The principle of this flooding (without any optimization) is simple. If a peer doesn’t have the item asked in the query which he received, the query is forwarded to all its neighbours.
With some optimization, and the introduction of a time to live (TTL) and a message Id, we can prevent our system from forwarding some unnecessary queries.

The following messages are used by the flooding algorithm:

- find: Questioning through the graph of the existence of a peer’s name.
- found: Positive reply from a peer.
- get: ‘Downloading’ of the name of a peer, if the *find* function has previously found it and stored its address in the founditem list of the given peer

The Find packet includes:

- senderpid: the IP Address of the last peer that sent us the the walker
- *sourcepid*: the IP Address of the peer that started the Kfind process
- *TTL*: the Time To Live of the message
- lookingfor: the item sourcepid peer is looking for (just the name of a particular peer, e.g. “P20”).

The Found packet includes:

- *pid*: the IP address of the discovered peer
- *name*: the name of the discovered peer
- *path*: the path that has been taken to discovered the given peer

The Get packet includes:

- *pid*: the IP address of the discovered peer
- *name*: the name of the discovered peer

find algorithm:

```
-----------------------------------------------------------------------
IF I have seen this message THEN
      RETURN
ELSE IF I have the file THEN
      I am sending a found response to the original sender
ELSE
      TTL--
      IF TTL > 0 THEN
            FOR each peer P in nlist
                  IF P has the item THEN
                        forward the find message to P
                        RETURN
            FOR each peer P in nlist
                  IF P is the sender of this message THEN
                        CONTINUE
                  forward the find message to P
-----------------------------------------------------------------------
```

found algorithm:

```
-----------------------------------------------------------------------
Add the address of what I found in my founditem list
-----------------------------------------------------------------------
```

```
get algorithm:
-----------------------------------------------------------------------
IF I have the item in my founditem list THEN
      'download' the file from the peer
ELSE
      Display a warning 'item not yet found'
-----------------------------------------------------------------------
```

V. K-WALKER SEARCH

Kfind

With that command, we try to find items in the network using K-walker algorithm. A peer receiving a kfind query is expected to:

- Check if it has the item
- If not: Look if any of its neighbours have the item
- If not: Send the K-walker to another peer

We have implemented the K-walker algorithm with two improvements:

1. Nodes maintain a state and do not forward the same query to the same neighbour twice in a row (they remember how many times they sent a message to a neighbour for a particular key)
2. If the best candidate to receive the walker is the one we have just received the walker from, then chose the second best candidate (if any).

The kfind packet includes:

- senderpid: the IP Address of the last peer that sent us the the walker
- *sourcepid*: the IP Address of the peer that started the Kfind process
- *TTL*: the Time To Live of the message
- lookingfor: the item sourcepid peer is looking for (just the name of a particular peer, e.g. “P20”).

In our implementation, we chose to utilize a dictionary named *klist*, whose purpose is to keep track of how many times we have sent a walker to a specific peer, looking for a specific item.

The structure of *klist* is:
(neighbour P, key) → value: number of times we sent a walker to neighbour P, seeking for item key

K-walker algorithm:

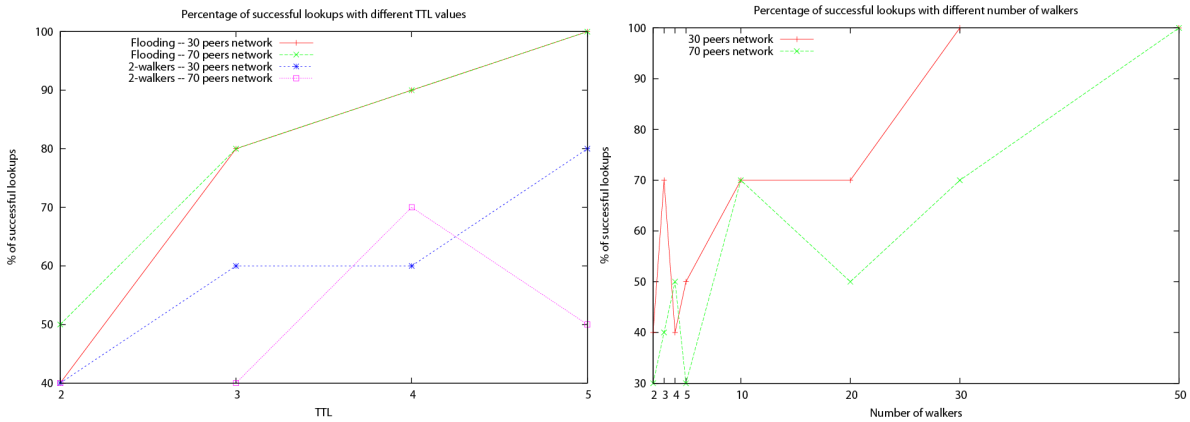
```
-----------------------------------------------------------------------
update logging info of sender peer

IF I have the item THEN
      send found message to source peer
ELSE
      TTL--
      IF TTL > 0 THEN
            IF one of my neighbours has the item THEN
                  send kfind message to it
      ELSE
            FOR each neighbour P
                  IF key (P,lookingfor) is not in my klist THEN
                        ADD key to klist with value 0
            candidates <- all peers on klist who have lookingfor as the key
           min_candidates <- subset of candidates with the minimum value
            IF |min_candidates| > 1 AND senderpid is in min_candidates THEN
```

```
            remove the senderpid from min_candidates
        p <- one peer selected randomly from min_candidates
        increase by 1 the value of the item (p,lookingfor) on klist
        send the walker to p
-----------------------------------------------------------------------
```

Efficiency Comparison

In each of the following experiments, we computed 10 random lookups and averaged their results in order to get a single datum. When we compared flooding and k-walker, we always utilized the same overlay network.

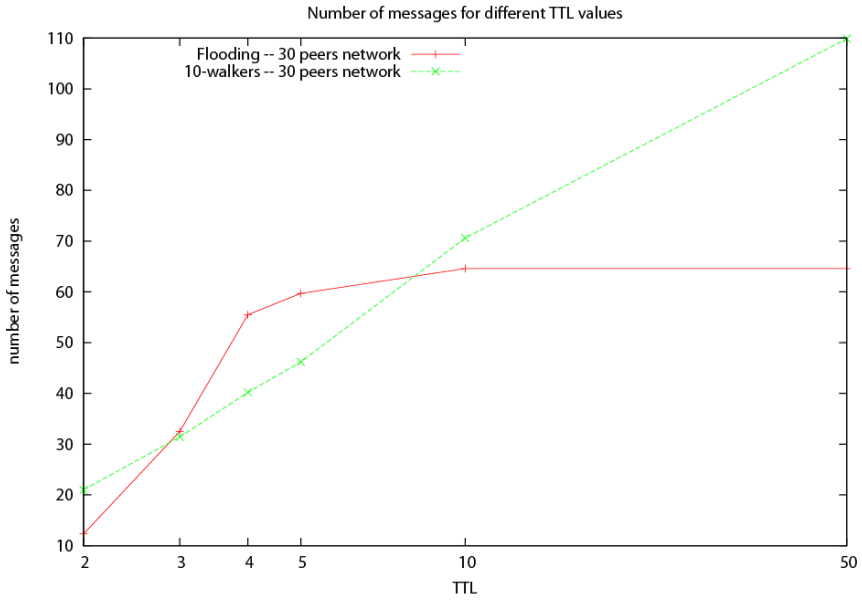
Let us now compare the efficiency of the 2 previous algorithm (“regular” flooding algorithm and K-Walker algorithm), in term of data finding as well as of network overload:



In the first graph, we compare the efficiency of the data finding (i.e. the percentage of successful lookups) of our flooding algorithm, and our 2-walkers algorithm. As TTL grows, the flooding algorithm becomes really efficient, no matter what the size of the network is (for a TTL of 5, we have a 100% of successful lookups with a network of 70 peers).
In comparison, even if the efficiency of the 2-walkers algorithm grows with the growth of TTL, we do not reach such a high rate of successful lookups. This could be expected: the flooding methodically explores every possible edge, whereas every individual walker randomly ‘walks’ and thus can miss some important edges.
The second graph describes the percentage of successful lookups depending on the number of walkers. It shows us that the more numerous the walkers, the more successful the lookups. We see that with a number of 10 walkers, we have a very acceptable successful percentage of 70% for a 30 peers network.

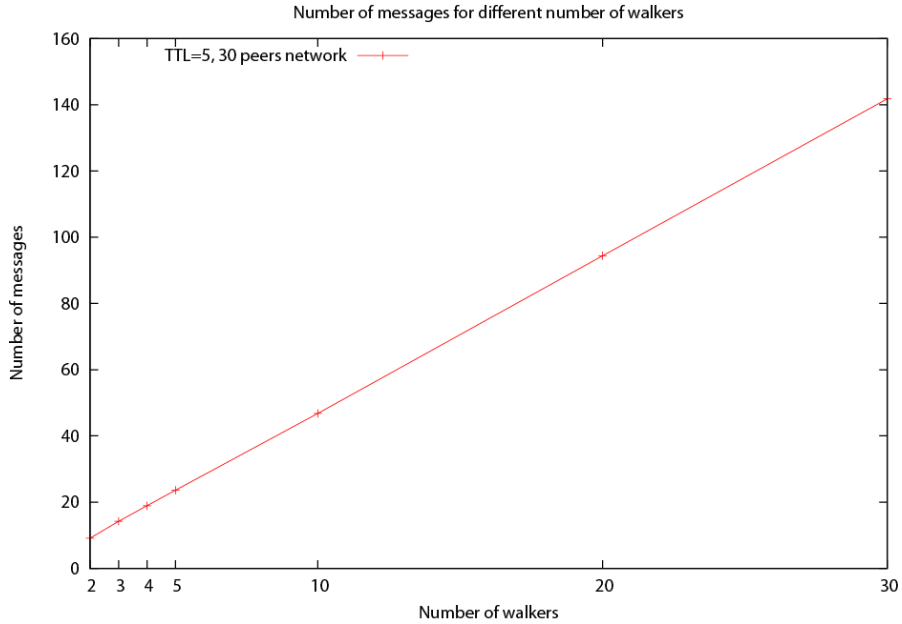
Those 2 first graphs showed us that data retrieving is always more efficient by flooding, but the difference can be compensated if the number of walkers sent in the graph is big enough.

The following graph (With a logarithmic scale for TTL) compares the network overload between our flooding algorithm and our K-Walker (10-walkers here) algorithm. This is measured as total number of messages routed by each participating peer.



- In our logarithmic scale, the 10-walkers curve follows a linear model whereas the flooding curve follows an exponential model. However, it is interesting to notice that the flooding curve has a clear upper bound, given the algorithm prevents message loops. The k-walker algorithm doesn’t have such a behaviour, because the walkers will keep traveling across the network until they reach the goal or TTL expires.
- We can see that for small TTL ($1 \leq TTL < 3$), the overall overload is worse for the 10-walkers algorithm than for the flooding. We can explain this by the fact that for a small TTL, the flooding messages do not have the ‘chance’ to spread exponentially, whereas we already ‘released’ 10 walkers in the network, which will generate approximately *10xTTL* messages.
- For intermediate TTL ($3 \leq TTL < 9$), the efficiency of the 10-walkers algorithm becomes clear. The number of messages generated remains linear, while the flooding algorithm begins to generate a tremendous amount, and literally ‘floods’ the network.
- For big TTL ($TTL \geq 9$), the efficiencies are once more reversed ! Why is that so ? At some point, if the TTL is big enough, the flooding has been through all the possible edges, and can’t explore anything else. On the contrary, our walkers, as long as they didn’t find the item they were looking for, will continue to look for it (They are zombies after all!).
  To efficiently solve that problem, an elegant solution would be to check now and again the status of the search, from the initial peer point of view. Thus, when a walker would find the item, the other walkers wouldn’t continue their pointless search.

To end, the last graph we’ll study is a graph showing the overall overloading, depending on the number of walkers we release:



Without any surprise, we see that, for a fixed TTL, as the number of walkers grows, the number of message generated is linear. In effect, on an average, one walker will make the same number of steps before reaching the Seeked item (We have such a smooth curve because each point is actually already a mean of 10 experiments: the standard deviation is lowered down).

VI. CONCLUSION

Throughout this assignment, we have learnt several pieces of information about the design of an efficient unstructured P2P Network.
The first of these acquired learning is the necessity to have a certain knowledge for the bootstrapping part: every peer should have access to a tracker, or in our case to a “Super peer” to be able to join the network.
Another acquired knowledge is the necessity to build the Overlay Network in a certain way: the creation of a ‘backbone’, that is to say linking together efficient peers (bandwidth-wise) can even allow ‘small’ peers to reach any part of the network, through that ‘backbone’.
To end, the most important things we saw was the necessity to use ‘smart’ algorithm to search throughout the network: using a raw flooding algorithm could be devastating and overload the network, but efficient in term of data finding, whereas using a K-walker algorithm is more gentle in term of network overload, but the amount of walkers should be sufficient, in order to achieve data finding. Moreover, we have learnt that as the number of walker increases, the number of messages also increases, for a fixed TTL, which means that k-walker might be worst than flooding if k is set too high.