
# Simulating the Freenet 0.7 Darknet

## P2P Systems Final Project

Giuseppe Miraglia
University of Pisa

March 9, 2015
A.Y. 2014/2015

## Contents


**1 Introduction** **3**

**2 System Overview** **4**
- 2.1 Freenet Peers . . . . . 4
- 2.2 Routing Algorithm . . . . . 5
- 2.3 Swapping . . . . . 6

**3 Freenet 0.7 Darknet Implementation** **8**
- 3.1 The package *structure* . . . . . 8
  - 3.1.1 *FPeer.java* . . . . . 8
  - 3.1.2 *Message.java* . . . . . 10
  - 3.1.3 *HashMapEntry.java* . . . . . 12
- 3.2 The package *control* . . . . . 13
  - 3.2.1 *LocationKeysManager.java* . . . . . 13
  - 3.2.2 *OverlayInit.java* . . . . . 14
  - 3.2.3 *StatisticsInit.java* . . . . . 16
  - 3.2.4 Facebook Data Set Analysis . . . . . 18
- 3.3 The package *protocol* . . . . . 20
  - 3.3.1 *LinkableProtocol.java* . . . . . 20
  - 3.3.2 *MessagesExchangerProtocol.java* . . . . . 22
- 3.4 The PeerSim *configuration file* . . . . . 32

**4 Freenet 0.7 Darknet Analysis** **33**
- 4.1 Swapping algorithm tuning . . . . . 34
- 4.2 Key replication tuning . . . . . 36
- 4.3 Routing bounding tuning . . . . . 39

**5 Conclusions and Comparison with real parameters** **41**

**References** **42**

# 1 Introduction

The goal of the project is that of implement, simulate and analyze, using *PeerSim* [1], the behavior and the properties of the main operations offered by **Freenet 0.7 Darknet** Peer-To-Peer network, among which content storage, retrieval and swapping. Both the implementation of the real Freenet Client and the implementation described in this document are made in Java, following what described in the system specifics [2] and [3], in the papers [4] and [5], and in the slides of the course of P2P Systems of the University of Pisa.

To make the simulations real and to study the behavior of the system on a large scale network, the overlay network on which the simulations are performed is a representation of the contents of a Data Set extracted directly from the popular **Facebook**, which describes some relationships between entities present in this social network.

## 2 System Overview

As described in [6], **Freenet 0.7** is a P2P network which lets users to anonymously *share files*, *browse and publish freesites* (web sites accessible only through Freenet) and *chat on forums*, without fear of censorship. Furthermore, since Freenet can be used in the "*Darknet mode*", where users directly connect only to their trusted friends (from Peer-To-Peer to *Friend-to-Friend* overlay network), it is, of course, very difficult to detect. In the following we will use the term "Freenet" referring ever the Freenet 0.7 Darknet to which we are interested, with exceptions of where otherwise specified.

### 2.1 Freenet Peers

Communications between Freenet peers are encrypted and are routed to make it extremely difficult to determine who is requesting the information and what the request-content is. Furthermore, each peer contributes to the network by giving **bandwidth** and a portion of their hard drive for **data storage**, in which could be stored (encrypted) also a content of a non-friend peer.

Each Freenet peer owns an unique, immutable, by-friends assigned **identifier** and an unique, mutable, randomly assigned **location key** in $[0, 1)$: the identifiers are used by the peers to identify their trusted friends peers, while the location keys are used in the routing. In the same way, also each content inserted into the overlay network is paired with an unique location key in $[0, 1)$.

Notice that the location keys are treated as **arranged along a circle** like *Chord* and *Symphony* DHTs, so the distance $D$ between location 0.99 and 0.01 is 0.02, rather than 0.9. More formally, the **distance** between two location keys $lock_a$ and $lock_b$ is the smaller of the two angles between them, and can be computed with the following formula:

$$D = \min \{ \, |lock_a - lock_b|, \; 1 - |lock_a - lock_b| \, \}.$$

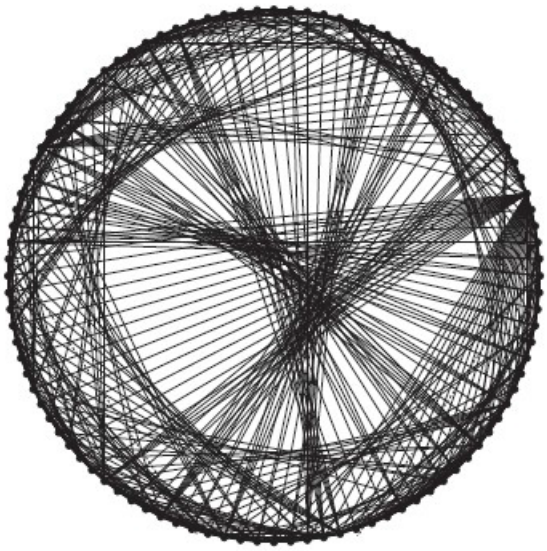
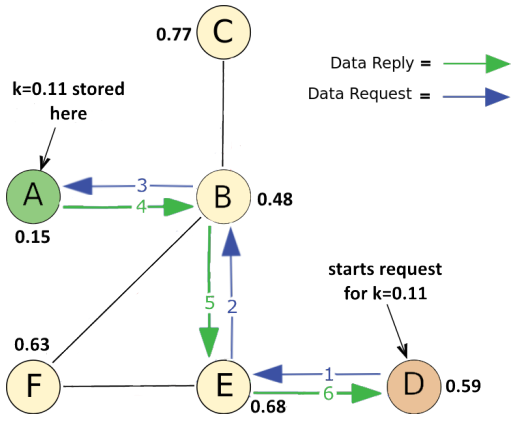


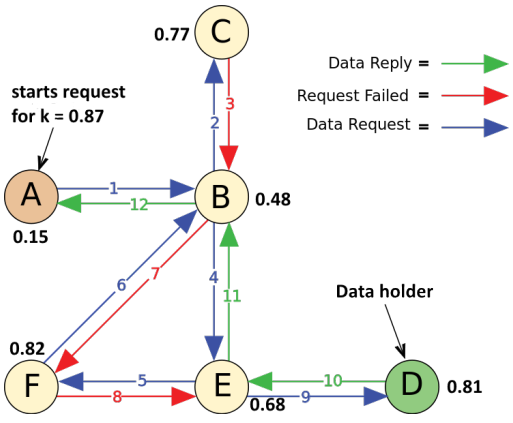
*Figure 1: an example of distribution of location keys around the circle space. The arrows between two points of the circle express the concept of close-because-trust peers, which is different from the closeness because of similarity on location keys exploited by the routing algorithm.*

## 2.2 Routing Algorithm

The **routing algorithm**, both in contents adding (**PUT** requests) and retrieval (**GET** requests), use a Greedy Depth-First strategy, bounded by an **HTL** (Hops-To-Live) counter and driven by the **closeness**, in terms of distance, of the request content location key (paired with the content to search/add) w.r.t. the peers location keys. An example of PUT and GET requests, ignoring HTL management, are shown in *Figure 2*. We will see the implementation and the details about the algorithms in the next sections.




(a) *PUT request that starts from D and ends on A*




(b) *GET request that starts from A and ends on D*

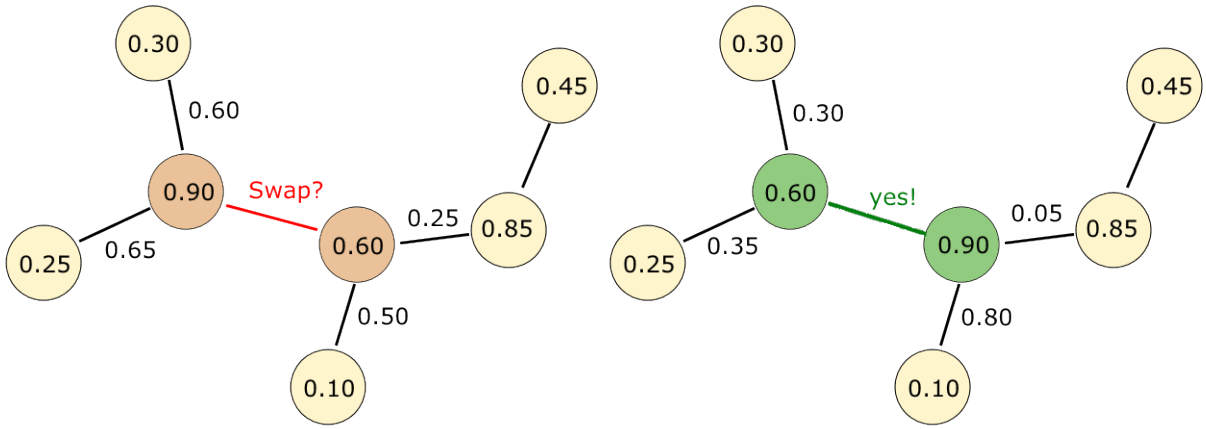
*Figure 2: behavior of the routing algorithm during PUT and GET requests*

However, is important to note that, initially, there is **no theoretical guarantees** that a content paired with a location key will be found, even if it is present into the overlay, or that it will be stored in the storage of the globally best node, because of the randomness of the location keys assignment. As we will see in the next sections, the swapping algorithm transforms this uncertainty into a certainty, with high probability.

## 2.3 Swapping

In order to make the routing algorithm faster, Freenet tends to cluster similar location keys on neighbors peers, with effect similarly to the rewiring technique of the *Kleinberg model*. Since the connections of the overlay network cannot be modified due of the trusted friends communications, each peer periodically tries to **swap** its own location key, together with the contents stored in its local storage, with a random selected peer in its *proximity*, that does not coincide necessarily with an its direct neighbor.

As we can see in *Figure 3*, the swap procedure involves two peers at a time, and performs a quantitative evaluation of the distances between the location keys of the two involved peers and which of their neighbors, first and after the (simulation of the) eventual swap. So, if the simulated swap results in a **better clustering configuration**, the swap is really performed; otherwise, all rest unchanged. A better and detailed explanation of the swapping procedure will be seen in next sections.




(a) *Situation after that the node 0.90 has sent a Swap request to the node 0.60*

(b) *Situation after the approving and the execution of the swapping between nodes 0.90 and 0.60*

*Figure 3: execution of the Swap procedure between two peers of the overlay network*

After that the swapping algorithm **convergence** has been reached (no swap requests can lead to a better clustering configuration), the resulting overlay is characterized by many clusters of peers whose location keys are close w.r.t. each other (many Kleinberg's *short range links*) and some peers whose location keys are farthest (few Kleinberg's *long range links*, due to the probabilistic choices).

As described by the *Sandberg's thesis* [7], the swapping algorithm executed between each pair of connected peers will eventually cause the location keys to converge to a state in which the routing needed for the requests will take only $O(\log N)$ steps, with **high probability**, where $N$ is the total number of nodes of the overlay, with the additional assumption that the overlay forms a **small-world network**.

# 3 Freenet 0.7 Darknet Implementation

As previous described, the implementation of Freenet that we have used in the simulations is made in *Java (JRE 1.8.0_31)*, exploiting the IDE *Eclipse* and *PeerSim* as P2P simulator. All the code relative to the implementation, together with the classes and the PeerSim configuration file, are reported in the other PDF document, while in the next sections of this document we will explain in detail the features, the design choices and the rule of each implemented component within the Freenet System.

## 3.1 The package *structure*

The package *structure* contains all the Java classes that provide a definition of some auxiliary structures and entities used for the implementation of the Freenet System. In the package we will find the files *FPeer.java*, *Message.java* and *HashMapEntry.java*.

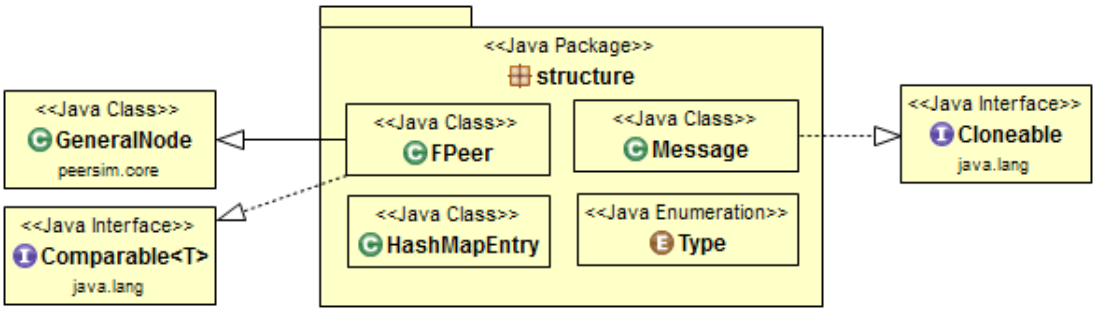



*Figure 4: the content of the structure Java package*

### 3.1.1 *FPeer.java*

The Java class *FPeer* defines the prototype of Freenet peer and, to make it usable for the simulation using PeerSim, the class must be an extension of the class *peersim.core.GeneralNode*. Furthermore, the class implements the interface *java.lang.Comparable<FPeer>* to define a complete order between the peers based on their location keys.

An object instance of the class *FPeer* owns the following fields:

- **identifier** of type *String*: represents the immutable identifier assigned to the peer when it joins the overlay network;

- **locationKey** of type *double*: represents the swappable-location key, in $[0, 1)$, which is pseudo-randomly assigned to the peer when it joins the overlay network;
- **storedContentLocationKeys** of type *HashSet*: represents the local storage of the peer. Even if it is initially empty, it will be filled with the content location keys paired to the contents that the peers will insert into the overlay network, according to the PUT routing algorithm;
- **isInvolvedInSwap** of type *boolean*: specifies if the peer is actually involved in a Swap operation with another peer and, therefore, if it cannot accept Swap requests.

Furthermore, an object of the class *FPeer* offers, among the most important, the following methods:

- **getDistanceFromLocationKey**: given a location key, computes the distance (circular, see Section 2.1) between the location key of the caller peer and the given location key;
- **compareTo**: given a peer, compares its location key with the location key of the caller peer, exploiting the natural ordering of the double-precision numbers. In doing this, the method defines a total order between the peers;
- **addContentLocationKey**: given a content location key, adds it into the local storage of the caller peer;
- **swapWith**: given a peer, performs the swap operation between it and the caller peer. This method is invoked only **after** that two peers have decided to swap, and performs the update of their data structures.

An instance of the *FPeer* class is created for each different identifier found in the Facebook Data Set, during its parsing in the phase of overlay setup.

### 3.1.2 *Message.java*

The Java class *Message* defines the prototype of message that will be exchanged between the peers of the overlay during the simulations. The class implements the interface *java.lang.Cloneable* to allow the duplication of an its instance.

The class owns the following static fields:

- **Type** of type *enum*: represents all the typologies of messages that can be sent by the peers during the simulations, among which:
  - **GET**, **GET_FOUND** and **GET_NOTFOUND**, used to identify a request of content retrieval, a positive answer and a negative answer w.r.t. to the same, respectively;
  - **PUT**, **PUT_OK** and **PUT_COLLISION**, used to identify a request of content insertion, a positive answer and a negative answer w.r.t. to the same, respectively;
  - **PUT_REPLICATION** and **PUT_REPL_COLLISION**, used to identify a request of content replication and a negative answer w.r.t. to the same, respectively;
  - **SWAP**, **SWAP_OK** and **SWAP_REFUSED**, used to identify a request of swapping, a positive answer and a negative answer w.r.t. to the same, respectively.
- **nextMessageID** of type *long-integer*: it is used to incremental assign unique identifiers to the new created messages.

An object instance of the class *Message* owns the following fields:

- **messageID** of type *long integer*: represents an unique identifier for the message, assigned to it using the previously seen class static field;
- **messageType** of type *enum value*: represents the type of the message;
- **messageLocationKey** of type *double*: represents the content location key, in $[0, 1)$, paired with the content to search or insert into the overlay network using the message;

- **lastHopFPeer** of type *reference* (in the simulations, but e.g. unique identifier in real networks): represents the peer of the overlay that have sent the message the last time;
- **HTL** of type *integer*: represents the Hops-To-Live counter of the message. When the message is created, the field is set to the maximum HTL value, as specified in the system configuration. Then the counter is decreased by 1 each time that the message is forwarded. Notice that, under certain conditions that we will see in the next sections, the HTL counter of the message could be **reset** to the maximum value;
- **pathClosestLocKey** of type *double*: represents the location key, in $[0, 1)$, of that peer which has been met during the routing of the message and that has location key closest w.r.t. the message content location key;
- **THC** of type *integer*: represents the True-Hops-Counter of the message, so the real number of steps that the message has traveled during the routing (only increment, no resets). Notice that this field is not used to implement the protocol itself, but only for statistical analysis.

Furthermore, an object of the *Message* class offers, among the most important, the following methods:

- **changeMessageType**: given a type, changes the type of the caller message with the given type;
- **changeLastHopFPeer**: given a peer reference, changes the reference of the peer that have sent the last time the caller message with the given reference;
- **changePathClosestLocKey**: given a location key, changes the most closest, w.r.t. the caller message content location key, peer location key of the caller message with the given location key;
- **decreaseHTL**: decreases by 1 the value of the HTL field of the caller message;
- **resetHTLTo**: given a value, resets the value of the HTL field of the caller message to the given value;

An instance of the class *Message* is created for each different sent request, e.g. GET, PUT, PUT_REPLICATION or SWAP. Otherwise, for the answers to these requests, e.g. GET_FOUND, GET_NOTFOUND, PUT_OK, etc., is exploited the instance of the request message itself, obviously changing the appropriate fields according to the context.

### 3.1.3 *HashMapEntry.java*

The Java class *HashMapEntry* defines the prototype of the value-side of an HashMap key-value entry, used by the peers of the overlay to keep trace, during the simulations, of the sent and received requests.

An object instance of the class *HashMapEntry* owns the following fields:

- **receivedFrom** of type *reference*: represents the peer of the overlay from which the peer owner of the HashMap has received the message identified by the key-side of the correspondent HashMap entry;
- **sentTo** of type *HashSet*: represents the list of peers of the overlay toward which the peer owner of the HashMap has sent the request identified by the key-side of the correspondent HashMap entry;
- **lastUseTimetamp** of type *long integer*: represents the time instant, in milliseconds, in which the entry was used the last time, where "used" means both queried or updated. This field is used by each peer to performs **cleanup operations** on the own HashMap, in order to remove old and useless entries. The **cleanup period** and the **useless factor** are parameters specified in the PeerSim configuration file.

Furthermore, an object of the class *HashMapEntry* offers, among the most important, the following methods:

- **setTimestampToNow**: sets the last used time of the caller entry to the current (invocation) time. Notice that this method is private, but that it is invoked by each one of the next described methods;
- **addSent**: given a peer reference, adds it to the list of the peers toward which the HashMap's owner peer has sent the correspondent request. The method updates the last used time of the entry;

- **alreadySentTo**: given a peer reference, checks if it identifies a peer toward which the HashMap's owner peer has sent a request. The method updates the last used time of the entry.

An instance of the class *HashMapEntry* is created for each *different* GET, PUT or PUT_REPLICATION received request, and it is used as value-side of the HashMap entry having as key-side the message identifier of the correspondent request.
This is a very helpful class, because an entry having this structure permits to drive efficiently the Greedy Depth-First routing of a request, **avoiding duplicated** requests, **cycles** and simplifying the **backward routing**.

## 3.2 The package *control*

The package *control* contains all the Java classes that provide a definition of the entities used for the location keys management and for the overlay setup, initialization and statistics. In the package we will find the files *LocationKeysManager.java*, *OverlayInit.java* and *StatisticsInit.java*.

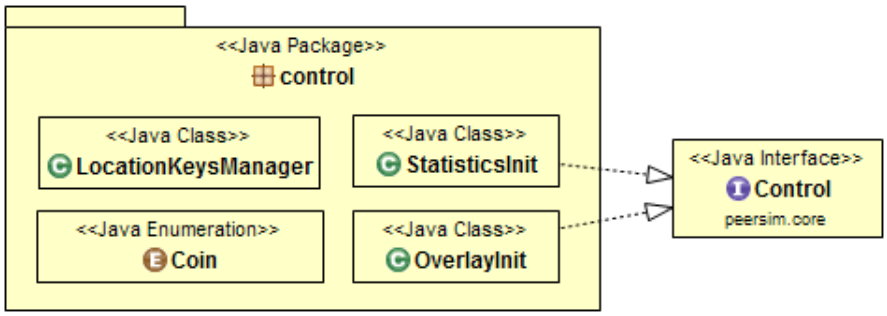



*Figure 5: the content of the control Java package*

### 3.2.1 *LocationKeysManager.java*

The Java class *LocationKeysManager* defines the entity that manages the generation of double-precision, unique location keys in $[0, 1)$ to assign to contents and peers during the simulations.

The class owns the following static fields:

- **Coin** of type *enum*: represents the two faces of a standard coin, **HEAD** and **TAIL**;

- **MAX_ITERATIONS** of type *integer*: represents the maximum number of iterations used to generate an unique location key;
- **hsFpeers** of type *HashSet*: represents the list of all the already generated location keys that were assigned as peer location keys;
- **hsContents** of type *HashSet*: represents the list of all the already generated location keys that were assigned as content location keys.

Furthermore, the class owns the following static methods:

- **tossCoin**: given a biasing factor, implements the operation of tossing of a biased coin. To make the result unbalanced, the probability is computed using a pseudo-random generator, every time initialized with a different seed, and the returned face of the coin depends of the biasing factor;
- **generateUniform**: generates a pseudo-random, unique, double-precision location key to assign to a peer or to pair with a content. The distribution used for the generation is the **uniform distribution**;
- **getAvailableContentLocationKey**: selects a content location key from the set of all already generated. Notice that the choice is made using a pseudo-random generator, every time initialized with a different seed.

Since the methods of this class are static, they are used during the simulations by the other classes to generate unique location keys to assign to the peers or to pair to the contents to PUT in the overlay, or also to search using a GET request.

### 3.2.2 *OverlayInit.java*

The Java class *OverlayInit* defines the entity that initialize and setup the overlay network for the simulations, parsing the given Facebook Data Set seen in the previously sections. The class behave as **Initializer**, so it implements the PeerSim interface *peersim.core.Control* and it runs once before the start of each simulation.

The described Initializer owns the following fields:

- **FPeer_prefix** of type *String*: represents the prefix, specified in the PeerSim configuration file, with which create a new peer as instance of the previously seen *FPeer* class;
- **linkablePID** of type *integer*: represents the identifier assigned by PeerSim to the used Linkable protocol, which we will see when we will describe the package *protocol*;
- **datasetPath** of type *String*: represents the File System relative path, specified in the PeerSim configuration file, of the Data Set to parse in order to create the overlay network.

The Initializer is invoked once by the simulator engine before the begin of each simulation, and performs the following actions defined by the method **execute**:

1. It tries to open in *read mode* the Data Set file at the specified File System path: if the open fails, then it stops the execution of the simulation setup. Otherwise continue.
2. It reads the opened file line-by-line: since, for our purpose, each line represents a trusted relation between two identifiers associated to two different entities of the Data Set, it allocates a new peer (using the support method **createAndAddFPeer**) for each not already processed identifier and an undirected edge between each pair of met identifiers: the method assigns not only the met **identifier** to the created peer, but also a random **location key** generated using the methods of the previously seen class *LocationKeysManager*. The neighborhood view of each peer is managed using the class *LinkableProtocol*, which we will see in the description of the next package.
3. If during the step 2 any error occurs, e.g. read, memory allocation etc., the Initializer stops the execution of the simulation setup. Otherwise continue.
4. At this point, the Data Set has been parsed successfully and the overlay network, together with its peers and the trusted connections between them, is online: the simulation environment is ready to start the real Dark Freenet simulation.

### 3.2.3 *StatisticsInit.java*

The Java class *StatisticsInit* defines the entity that performs some "*offline*" statistics on the parsed Facebook Data Set correspondent to the already created overlay network. "Offline" means that such statistics are not directly related to the simulations of the Freenet System, but mainly on the static overlay of its trusted connections, in order to extract some networks properties from it.
The class behave as **Initializer**, so it implements the PeerSim interface *peersim.core.Control* and it runs once before the start of each simulation.

The described Initializer owns the following fields:

- **MAX_DISTANCE** of type *integer*: represents the maximum distance used in Floyd-Warshall shortest paths computation;
- **linkablePID** of type *integer*: represents the identifier assigned by PeerSim to the used Linkable protocol, which we will see when we will describe the package *protocol*;
- **dataSetName** of type *String*: represents the File System relative path, specified in the PeerSim configuration file, of the Data Set on which performs the statistics;
- **selectedOverlayStatistics** of type *String*: specifies, through the configuration file, which kind of statistics must be performed on the overlay network created from the relative Data Set. The allowed values are:
  - **null**, if no statistics must be computed.
  - **degree**, if must be computed the degree of each peer of the overlay network;
  - **cc**, if must be computed the **local clustering coefficient** of each peer of the overlay network;
  - **diameter**, if must be computed the **diameter** of the overlay network;
  - **avgpl**, if must be computed the **average shortest path length** of the overlay network;

The Initializer is invoked once by the simulator before the begin of each simulation and performs, depending on the value of its above discussed field, the following actions defined by the method **execute**:

- if the value is "**degree**", is invoked the method *analizesFPeersDegree*, which computes and writes on a statistics file the number of neighbors of each peer of the overlay network;
- if the value is "**cc**", is invoked the method *analizesFPeersLocalCC*, which computes and writes on a statistics file the **local clustering coefficient** $C_i$ of each peer $i$ of the overlay network, where $N(i)$ is the set of its neighbors, using the following formula :

$$C_i = \begin{cases} \dfrac{|\{e_{jk} \mid j,k \in N(i),\ e_{jk} \in E\}|}{|N(i)|\,(|N(i)|-1)} & \text{, if } |N(i)| > 1 \\ undefined & \text{, if } |N(i)| \leq 1. \end{cases}$$

  After, defining $V^* = \{n \in V \mid |N(n)| > 1\}$ as the set of all the overlay's peers for which the local cluster coefficient is defined, these results are collected and summed to obtain the network average clustering coefficient:

$$\vec{C} = \frac{1}{|V^*|} \sum_{i \in V^*} C_i$$

- if the value is "**diameter**", is invoked the method *findDiameter*, which:
  1. computes the shortest paths between each pair of peers of the overlay in $O(|V|^3)$ using the **Floyd-Warshall algorithm** [8];
  2. then finds the most length shortest path: its length corresponds to the diameter of the overlay network.
- if the value is "**avgpl**", is invoked the method *findAvgShortestPathLength*, which:
  1. computes the shortest paths between each pair of peers of the overlay as in the previous case;

2. then computes the average of all their lengths, applying the following formula in order to find the average length of the shortest paths of the overlay network:

$$l_G = \frac{1}{|V|\,(|V|-1)} \sum_{\substack{i,j \in V \\ i \neq j}} |shpath(i,j)|$$

### 3.2.4 Facebook Data Set Analysis

Using the methods offered by the class *StatisticsInit*, the created overlay network can be analyzed to extract its main properties. The Data Set consists in 44.138 undirected relations between 7.190 different nodes, and has the following properties:

1. the **degree distribution** of its nodes follow the *Power Law* distribution: many nodes with a small degree, few nodes with a big degree (*Figure 6*);
2. the **average shortest path length** between each pairs of its nodes is $l_G \simeq 3.56$;
3. its **diameter** is $d = 4$;
4. its **network average clustering coefficient** is $\vec{C} \simeq 0.24$.

As expected by a social network, it represents a Small-World network having a low average shortest path length and some nodes with an high local clustering coefficient. This is a fundamental assumption which allows Freenet routing, with the help provided by the swapping algorithm, to achieves expected $O(\log N)$ steps, with high probability.

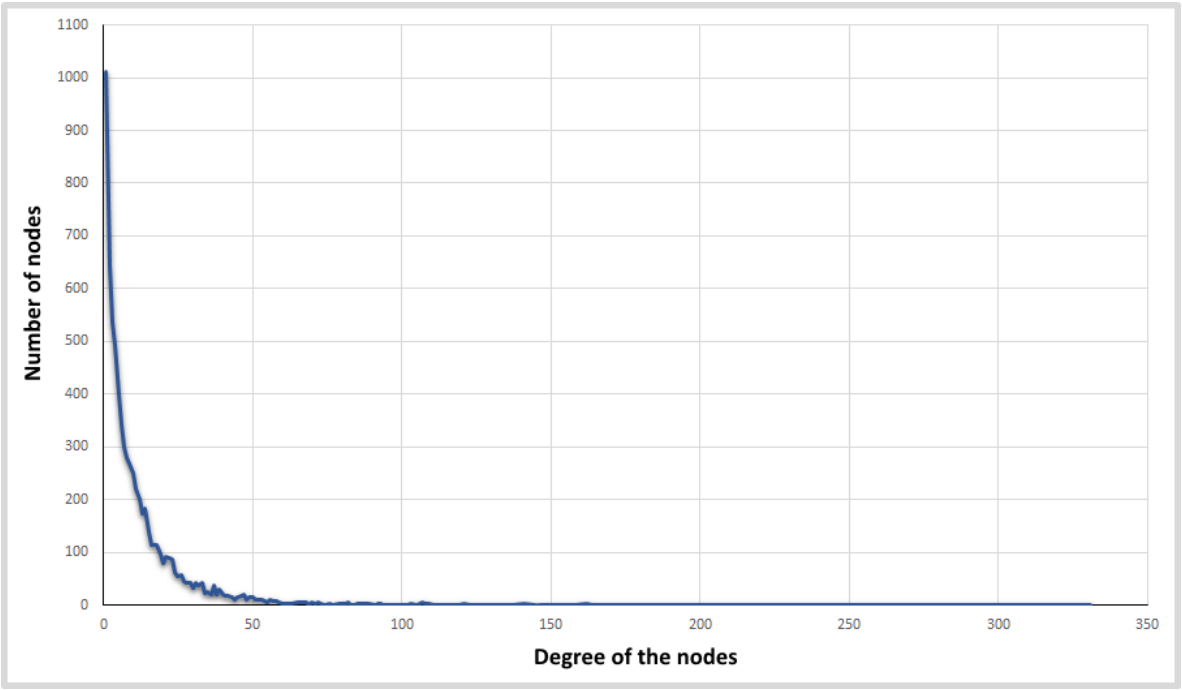



*Figure 6(a): the degree distribution of the nodes of the overlay network from the Facebook Data Set in a standard plot*

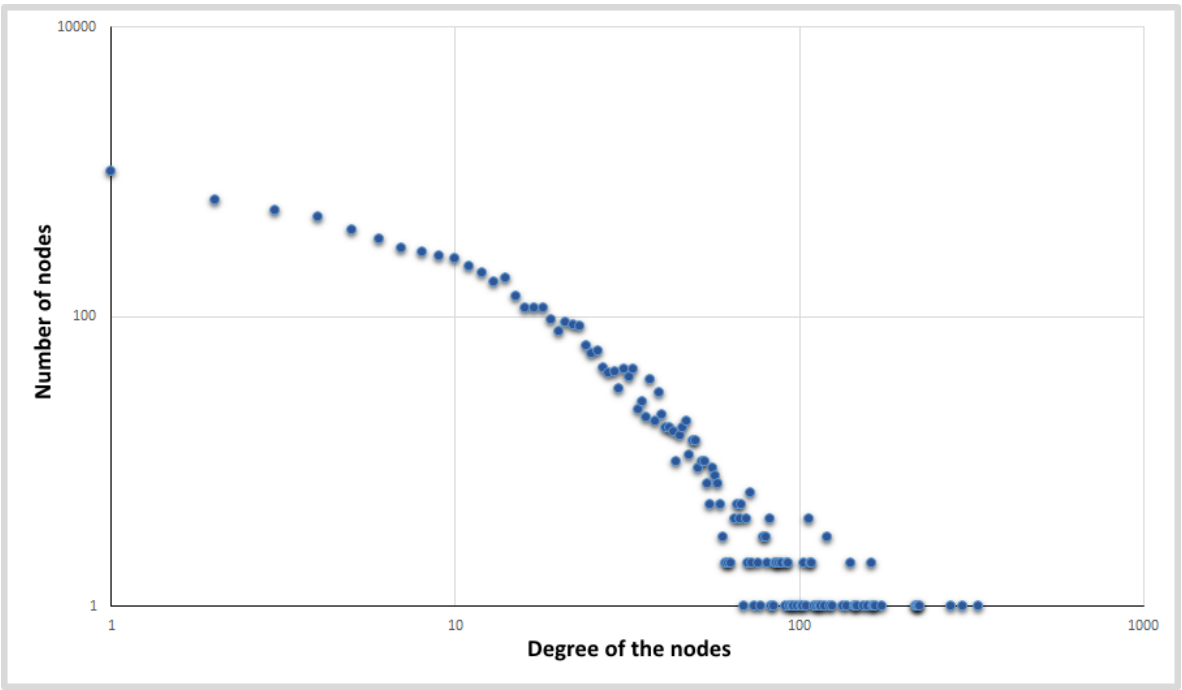



*Figure 6(b): the degree distribution of the nodes of the overlay network from the Facebook Data Set in a Log-Log plot*

## 3.3 The package *protocol*

The package *protocol* contains all the Java classes that provide a definition of the protocols executed by the peers of the overlay network during the simulations. These protocols allow the exchange of messages between the peers of the overlay, the management of the neighborhood view and of the trusted connections of each peer, and the periodic execution of various actions by each peer. In the package we will find the files *LinkableProtocol.java* and *MessagesExchangerProtocol.java*.

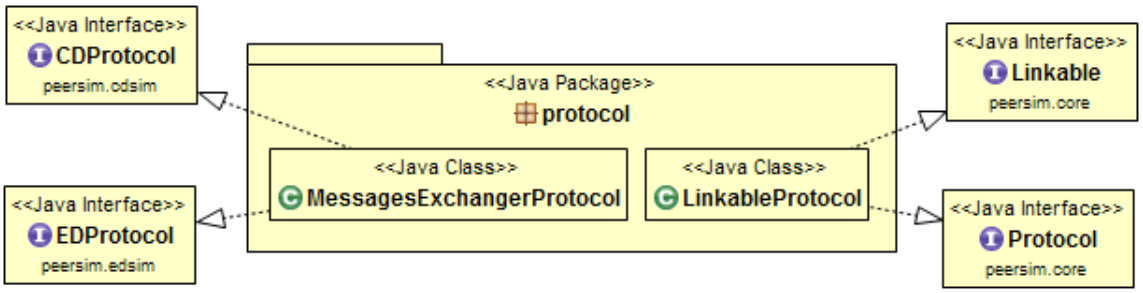



*Figure 7: the content of the protocol Java package*

### 3.3.1 *LinkableProtocol.java*

The Java class *LinkableProtocol* defines the entity that manages the neighborhood view and the trusted connections concept of each peer w.r.t. the other peers of the overlay network. In order to furnish an abstraction of the neighborhood, the class must implements the PeerSim interface *peersim.core.Linkable*, and since its behave also as **Protocol**, it must implements also the PeerSim interface *peersim.core.Protocol*.

The class abstracts the neighborhood view of each peer using its **neighborsRBTree** field, which corresponds to a **Red-Black Tree** [9] (*java.util.TreeSet*), that allows to organize the neighbors peers with an ordering criteria defined by the total order that the peers self-defines through the *java.lang.Comparable* interface. With such representation, each peer can perform all the basic data structure operations (e.g. add, contains, remove) in $O(\log S)$, where $S$ is the number of peers in its neighborhood. However, is important to notice that the most costly (w.r.t. others data structures like LinkedList or ArrayList) *add* operation is heavily performed only dur-

ing the overlay setup, so before the begin of each simulation, and so its cost has no impact on the simulations itself. An example of Red-Black Tree is shown in *Figure 8*.

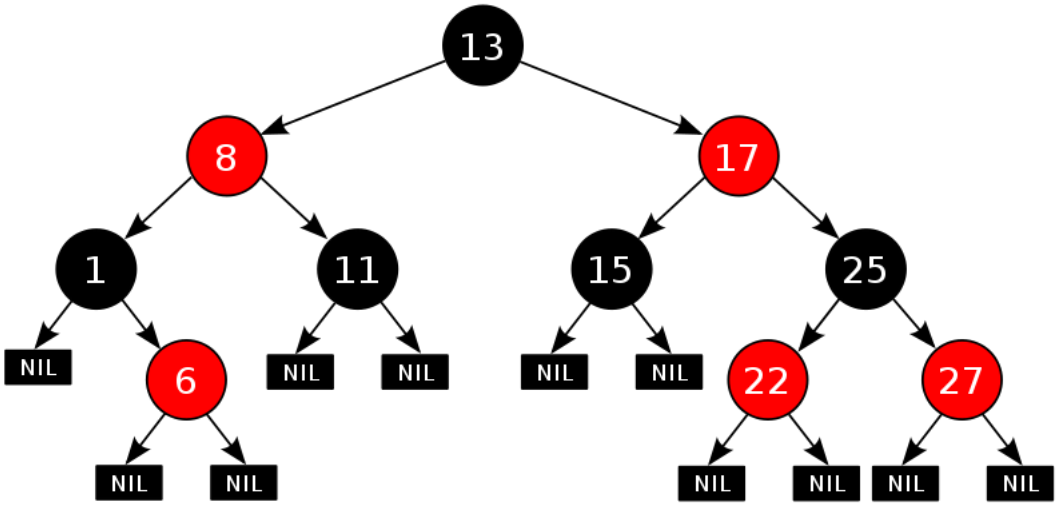



*Figure 8: an example of Red-Black Tree data structure. Obviously, in the case in which the stored items are peers rather than integers, we can imagine that there are location keys in the tree nodes.*

The *LinkableProtocol* class offers, among the most important, the following methods:

- **addNeighbor**: given a peer, adds it into the neighborhood of the peer associated to the caller protocol;
- **contains**: given a peer, checks if it is a neighbor of the peer associated to the caller protocol;
- **degree**: returns the number of neighbors of the peer associated to the caller protocol;
- **getNeighborByLocationKey**: given a location key, retrieves the peer owner of that location key from the neighborhood of the peer associated to the caller protocol;
- **retrieveTopKNeighbors**: given a location key and an integer $k$, retrieves, from the neighborhood of the peer associated to the caller protocol, the $k$ peers that have location key closest w.r.t. the given location key. Notice that $k$ is an upperbound to the number of neighbors to retrieve: if the running peer has less than $k$ neighbors, the method

retrieve less than the request number of peers. In the other cases, the method return exactly the $k$ most closest neighbors.
Notice that the operation implemented by this method is a part of the core of the Freenet routing algorithm, as we will see in the next section, and it is invoked many, many times during the simulations, and often more than once for each processed request. Of course, also in this case, the representation of the neighborhood as Red-Black Tree provide a good performance improvements, due to the availability of the **headSet** and **tailSet** methods (see the code for a more clear explanation).

- **updateNeighborhood**: performs the update of the tree-representation of each one of the neighbors of the peer associated to the caller protocol. This action is necessary when, after a swap operation, a peer changes its location key: in this case, the tree representations of the neighbors of that peer must be updated in order to maintains the peers sorted based on their location keys.

### 3.3.2 *MessagesExchangerProtocol.java*

The Java class *MessagesExchangerProtocol* defines the entity that manages the behavior of the peers of the overlay network during the simulations and that defines the Freenet routing protocol. Among the actions that each peer could perform during a simulation, we have:

- the sending of a request (e.g. GET, PUT, PUT_REPLICATION or SWAP) toward an its neighbor;
- the processing of both a received request (e.g. GET, PUT, PUT_REPLICATION or SWAP) or a received answer (e.g. GET_FOUND, GET_NOT-FOUND, PUT_OK, PUT_COLLISION, PUT_REPL_COLLISION, SWAP_OK or SWAP_REFUSED) from an its neighbor;
- the removing of the useless entries from its own HashMap;
- the writing of some computed statistics on a statistic file.

In order to be able to performs some periodical actions during the simulations, the class must implements the PeerSim interface *peersim.cdsim.CDProtocol*. While, in order to be able to send and receive messages, the class

must implements the PeerSim interface *peersim.edsim.EDProtocol* which, in turn, needs of a transport protocol. The used tranport protocol is defined by the PeerSim class *peersim.transport.UniformRandomTransport* [10], which implements a transport layer that reliably delivers messages to the peers of the overlay.

This apparently means that the *MessagesExchangerProtocol* class behaves as an **Hybrid Protocol**, both Cycle-Driven and Event-Driven, but it is not the truth: the simulation control is on the messages as in the Event-Driven one, but using the class *peersim.edsim.CDScheduler* [11], PeerSim permits to incorporates cycles in the events processing, as shown in *Figure 9*.

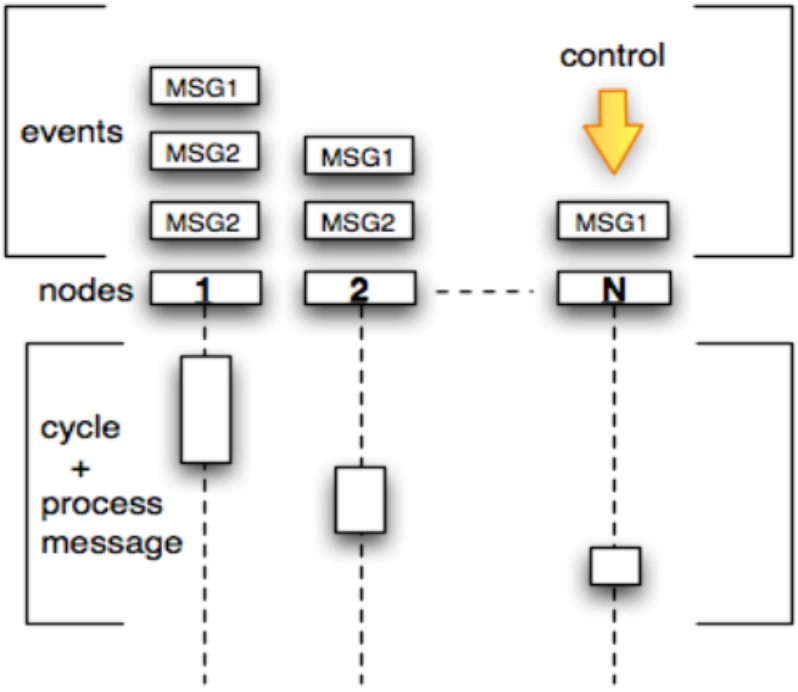



*Figure 9: an representative schema of PeerSim Hybrid Simulation.*

The described *MessagesExchangerProtocol* class owns the following static fields:

- **statisticsFileExtension** of type *String*: represents the extension to append to the statistics file used for the simulation. Since the field is ever initialized with the current system time, is ensured a different extension for each statistic file;

- **simStatToWrite** of type *boolean*: identifies the first write on the statistics file used in the simulation. This is necessary to write once, on the head of the file, a summary of the values of the various parameter used for the current simulation.

In the same way, a protocol instance of the *MessagesExchangerProtocol* class owns the following fields:

- **itselfPID** of type *integer*: represents the identifier assigned by PeerSim to the used message exchanger protocol itself;
- **transportPID** of type *integer*: represents the identifier assigned by PeerSim to the used transport protocol;
- **linkablePID** of type *integer*: represents the identifier assigned by PeerSim to the used Linkable protocol;
- **maxHTL** of type *integer*: represents the maximum value with which set the Hops-To-Live field of a message, when a GET, PUT or PUT_COLLISION request is created;
- **maxHTLswap** of type *integer*: represents the maximum value with which set the Hops-To-Live field of a message, when a SWAP request is created;
- **replicationFactor** of type *integer*: represents the maximum number of neighbors toward which a content location key must be replicated, during the content replication algorithm, after a successfully PUT request;
- **uselessFactor** of type *integer*: represents the time, in minutes, based on which a peer decides that an its HashMap's entry is to mark as "useless" and so becomes a candidate for removal;
- **cleanupFrequency** of type *integer*: represents the frequency, in units of time, based on which the protocol removes from its own HashMap the "useless" entries;
- **swapFrequency** of type *integer*: represents the frequency, in units of time, with which a peer performs a SWAP request in order to try to swap its location key with another peer;

- **biasFactor** of type *double*: represents a factor used to model the probability to send a GET request rather than a PUT request, during the execution of the simulations and, in particular, of a single peer. To better understand how this probability is managed, see in details the method *tossCoin* of the already described class *LocationKeysManager*;
- **printsAllowed** of type *boolean*: specifies if the protocol must performs some prints on the console during its execution. Ideally, it is only a debug field but, surely, deactivating the prints rather than cut off its from the code, it becomes more readable and easy to understand;
- **SRmessages** of type *HashMap*: represents the HashMap used by each peer to keep trace of the sent and received requests during the simulations. Each HashMap's entry can be seen as a pair **(ID, ref)**, where *ID* is the identifier of the message that has caused the allocation of the entry, while *ref* is a reference to an object instance of the previously seen class *HashMapEntry*, which contains information on the peer from which the message has been received and toward which it has been sent.

Furthermore, a protocol instance of the *MessagesExchangerProtocol* class owns, among the most important, the following methods:

- **cleanHashMap**: performs a removing of all the useless entries from the HashMap of the peer correspondent to the caller protocol. This method is invoked periodically depending on the parameter *cleanupFrequency* (see above);
- **writeStatisticsOnFile**: performs some writes on the statistics file reserved for the simulation. Typically, each write corresponds to a string as "messageType messageTHC", or "currentTime messageLocKey messageTHC" etc;
- **isLessWrtContent**: given two peer locations keys and a content location key, checks if the first peer location key has distance, w.r.t. the content location key, less than the second peer location key;
- **findBestIndex**: given a list of neighbors, tries to find a neighbor of the peer associated to the caller protocol to which it has not already sent the request identified by the given HashMap entry, and

that is different from the given peer to avoid. In checking duplicated requests, the method exploits the features offered by the already seen class *HashMapEntry*;

- **sendMessage**: given a sender peer, a receiver peer and a message, invoke the transport protocol of the sender peer to send the message to the receiver peer. This method represent the send-part of the core of the implemented message exchanger protocol;
- **replicatesTowardNeighbors**: replicates the given content location key, already successfully inserted by a PUT request in the local storage of the given peer, sending a PUT_REPLICATION request toward a specified number of its closest w.r.t. the content location key neighbors. This number is specified by the field *replicationFactor*, and it is an upperbound for the number of neighbors toward which replicates the content location key, since a peer could have a less number of neighbors. Notice that each sent PUT_REPLICATION request, relatively to a single PUT request, has the same message identifier and has the HTL field set to the maximum value;
- **handleBackwardMessage**: given a peer, handles its receiving of the given answer message (e.g. PUT_OK, PUT_COLLISION, GET_FOUND), performing the following operations:

  1. it checks the HashMap of the given peer in order to find the peer from which it has received the relative request message (e.g. PUT, GET);
  2. if that peer is itself, this means that it is the owner of the request message, so do nothing and ends the routing; otherwise, it propagates the received answer message toward that neighbor.

- **handleReceivedGET**: this method implements the core of the routing algorithm for content retrieval (GET requests), following the pseudocode described in [4] and [5]. The algorithm use a Greedy Depth-First Search strategy, bounded by an HTL value. A peer that receive a GET request, performs the following actions:

  1. If the request is identical to an its recently processed requests (check performed exploiting information present in its own HashMap, ac-

cessible through the message identifier), the peer notifies the sender about duplication status sending it a GET_NOTFOUND answer. Otherwise continue.

2. If the requested data (in our case, content location key) is present in its own local data storage, the peer sends an GET_FOUND answer to the sender. Otherwise continue.
3. If its location key is closer, w.r.t. the content location key to search, than the location keys of any previously visited peer during the routing of the processed message (exploiting the message's *pathClosestLocKey* field, we needs just a single check), the peer resets the message Hops-To-Live field to the maximum value, specified by the parameter *maxHTL*, and continue. Otherwise, simply continue.
4. If, at this point, the Hops-To-Live counter of the request is zero, the peer answers to the sender with a GET_NOTFOUND message. Otherwise continue.
5. If the peer has reached this point, means that it must forward again the GET request. So it tries to find (using the *findBestIndex* method) an its neighbor which have location key closest w.r.t. the content location key to search, obviously excluding those already visited and the peer from which it has received the current processed request. If a such neighbor exists, the peer decrements the HTL counter of the message and forwards the GET request to this neighbor peer, which repeats the algorithm from step 1. If such neighbor does not exists, the peer answers the sender of the request with a GET_NOTFOUND message.

- **handleReceivedPUT**: this method implements the core of the routing algorithm for content insertion (PUT requests). The algorithm for PUT request of a content retraces exactly the first depth-path of an unsuccessfully GET request for the same location key: only the first because it does not permits backtracks in the Depth-First Search. A peer that receive a PUT request, performs the following actions:

  1. If the peer already contains the data (in our case, the content location key) in its own local storage, it answers to the sender with a PUT_COLLISION message. Otherwise continue.

2. Then, the peer tries to find an its neighbor which has location key closest than its location key, w.r.t. the content location key to insert: if such neighbor exists, the peer forward the PUT request to it. In the other case:
   (a) it stores the content location key in its own local storage;
   (b) it sends a PUT_OK answer toward the peer from which it has received the request;
   (c) it propagates a PUT_REPLICATION request, for the same content location key to insert and having maximum HTL value, to some of its neighbors.

   The maximum number of neighbors toward which the peer replicates the request is specified by the parameter *replicationFactor*. Furthermore, in order to optimize the routing of the replication requests, the neighbors are selected again evaluating its proximity w.r.t. the content location key.

   Notice that due to the fact that the routing is driven by the closeness w.r.t. the content location key, the triangular inequality property holds, and so the check to exclude the already visited neighbors is not necessary.

- **handleReceivedPUT_REPLICATION**: this method implements the core of the routing algorithm for content replication (PUT_REPLICATION requests), invoked by a peer after that a successfully PUT request has been completed in its own local storage. The routing strategy used by the PUT_REPLICATION requests is exactly the same implemented for the GET requests, but with the additional stop conditions induced by the routing algorithm of the PUT requests:

  1. If the request is identical to an its recently processed requests (check performed exploiting information present in its own HashMap, accessible through the message identifier), the peer notifies the sender about duplication status sending it a PUT_REPL_COLLISION answer. Otherwise continue.
  2. If its location key is closer, w.r.t. the content location key to search, than the location keys of any previously visited peer during the

routing of the processed message (exploiting the message's *pathClosestLocKey* field, we needs just a single check), the peer resets the message Hops-To-Live field to the maximum value, specified by the parameter *maxHTL*, and continue. Otherwise, simply continue.

3. If, at this point, the Hops-To-Live counter of the request is zero, the peer stores the content location key in its own storage and ends the routing for the message. Otherwise continue.
4. If the peer has reached this point, means that it must forward again the PUT_REPLICATION request. So it tries to find (using the *findBestIndex* method) an its neighbor which have location key closest w.r.t. the content location key to replicates, obviously excluding those already visited and the peer from which it has received the current processed request. If a such neighbor exists, the peer decrements the HTL counter of the message and forwards the PUT_REPLICATION request to this neighbor peer, which repeats the algorithm from step 1. If such neighbor does not exists, the peer stores the content location key in its own storage and ends the routing.

- **selectNeighborForSwap**: given a peer, tries to select an its neighbor to involve into a swapping process with itself. The choice is made taking care that a peer is not already involved in a swapping process with another peer, and that the selecting process ends also if no peers are available, after some attempts;

- **handleReceivedSWAP**: this method implements the core of the location keys swapping algorithm (SWAP requests), which is derived from the *Metropolis-Hastings algorithm* [12]. Seeing the overlay network as an undirected graph $G = (V, E)$, a peer $b \in V$ that receives a SWAP request from another peer $a \in V$, performs the following actions:

  1. First of all, $b$ checks the Hops-To-Live value of the SWAP message: if this value is greater than zero, then $b$ selects a random neighbor which is different from $a$ (from which it has received the actual request) using the *selectNeighborForSwap* method seen above, and

forwards to it the SWAP request decreasing the HTL value. Otherwise continue.

2. At this point, the message's HTL value is equal to zero, so $b$ tries to swap its location key and its stored content location keys with $a$, performing the following computation:

$$D_1(a,b) = \prod_{\substack{(a,i)\in E \\ i\neq b}} |lock_a - lock_i| \prod_{\substack{(b,j)\in E \\ j\neq a}} |lock_b - lock_j|$$

$$D_2(a,b) = \prod_{\substack{(b,i)\in E \\ i\neq a}} |lock_a - lock_i| \prod_{\substack{(a,j)\in E \\ j\neq b}} |lock_b - lock_j|$$

where $D_1(a,b)$ represents the product between the distance before the swap, while $D_2(a,b)$ represent the product of the distances after the eventual (simulated) swap.

3. Then if $D_1(a,b) > D_2(a,b)$, it means that the simulated swap results in a better clustering configuration, so the peer $b$ accept the SWAP request sending to $a$ a SWAP_OK answer message. Otherwise, if $D_1(a,b) \leq D_2(a,b)$, there is also a possibility for $b$ to accept the $a$'s SWAP request: $b$ accept the request with probability equal to the ratio $\dfrac{D_1(a,b)}{D_2(a,b)}$, otherwise it refuse the request. For $b$, refuse a request means send a SWAP_REFUSED message toward $a$.

   Notice that the algorithm tends to cluster close location keys on neighbors peers, in order to optimize, with high probability, the asymptotic number of step required to find a content. However, the possibility to accept the swap request also in the case in which it does not results in a better clustering configuration is used to escape from local minima;

- **handleReceivedSWAPanswer**: given a peer, handles its receiving of the given answer relative to a SWAP request (e.g. SWAP_OK, SWAP_REFUSED) that it has sent in the past, performing the following actions:

  1. if the received answer is of type SWAP_REFUSED, it does nothing;

2. otherwise, if the received message is of type SWAP_OK, the peer swaps its location key and its stored content location keys with the location key and the stored content location keys of the peer from which it has received the positive answer;

- **processEvent**: given a peer, handles its receiving of a message invoking the right method depending on the type of the given message. This method represent the receiving-part of the core of the implemented message exchanger protocol and must be implemented due to the fact that the described class implements the interface *peersim.edsim.EDProtocol* and use a transport layer for message exchanging;

- **performRequest**: this method is invoked by a peer when it wants to send a request to another peer. Depending on the given type (see the *Type* enum in the already described class *Message*) and the given location key, the method creates a request of that type for that location key. With regard to request HTL value, obviously if it is a GET request is used the field *maxHTL*, while if it is a SWAP request is used the field *maxHTLswap*.

  Furthermore, in all the cases in which the peer that wants to perform the request:

  – is unable to find a neighbor toward which propagates the SWAP request, or
  – already contains, in its own storage, the content location key to insert using the PUT request, or
  – already contains, in its own storage, the content location key to search using the GET request

  the request is **aborted**;

- **nextCycle**: this method allows each peer to performs periodically actions during the simulations. The method is invoked by the PeerSim Cycle-Driven engine with the periodicity specified in the configuration file, and must be implemented due to the fact that the described class implements the interface *peersim.cdsim.CDProtocol*.

Each peer, during the execution of this method, performs the following actions:

1. performs the cleanup of the useless entries of its own HashMap, if the cleanup period decades as specified in the PeerSim configuration file;
2. performs a SWAP request, if the swap period decades as specified in the PeerSim configuration file;
3. performs a GET or a PUT request, based on if the results of the *tossCoin* method is HEAD or TAIL, and depending on the *biasFactor* field used to unbalance the frequency of the two requests (see the class *LocationKeysManager*) .

Obviously, the tossing coin operation to decide if a peer must performs a GET or a PUT request is just a simple example on how requests can be randomly performed by the peer of the overlay and through which statistics can be computed, but many others heuristics can be used for the same attempt.

## 3.4 The PeerSim *configuration file*

In order to exploit PeerSim for the simulation, a configuration file must be defined. This file, in our case, is represented by the file *conf.cfg* and contains the main directives to setup the simulator environment, together with a list of parameters with the respective values to assign, during the simulations setup, as values of some fields of the implemented classes. The description of each parameter of the configuration file is explained directly in the configuration file itself, that can be found in the other PDF document together with the rest of the code.

# 4 Freenet 0.7 Darknet Analysis

In addition to the effective implementation, some empirical statistics have been computed on Freenet, in order to analyze its properties and its performance.
As seen in the previously sections, the protocol behaves differently based on some configuration parameters, among which:

1. the maximum number of steps that a request message, relative to a content, could perform during its Greedy Depth-First routing (parameter **maxHTL**);
2. the number of peers toward which a content must be replicated after its successfully insertion in the local storage of a peer (parameter **replicationFactor**);
3. the number of steps that a swap request can performs to get away from the neighborhood, in order to involves in the swapping process also peers far from it (parameter **maxHTLswap**).

The **tuning** of these parameters is made performing some analysis through an high number of simulations, which leads to the results that we will see in the following sections.
Mainly, we will focus on following typologies of analysis, performed w.r.t. the varying of the above described parameters:

- first of all, a most important analysis is based to the study of the **average number of routing steps** that a GET request requires to receive the relative answer, either positive or negative. Of course, the same can be made for a PUT request, but it does not make much sense because, as previously seen, the routing of a PUT request is not Depth-First, but just stops when find a disadvantageous path, also if it is not the best;
- another type of helpful analysis may be the study of the **percentage of positive and negative answers** relative to a fixed number of GET request, performed for some existing content location key, w.r.t. the varying of the above described parameters. Remember that there are no theoretical guarantees that, also for a GET request, a content

paired with a location key will be found using the Greedy Depth-First routing algorithm;

- finally, we can study the **impact of the swapping algorithm** on the previously described analysis, which theoretically should provide great improvements.

## 4.1 Swapping algorithm tuning

Observing the trend of the average number of routing steps that a GET request requires to receive a positive or negative answer, in function of the parameter **maxHTLswap**, we can study the **impact of the swapping algorithm on the number of steps required by a GET request**, so on the **routing performance**.

In doing this analysis, the other two parameters can be fixed, for instance, to **maxHTL = 300** and **replicationFactor = 10**. While **10** is a good value for the maximum number of peers toward which replicates the content location key (analysis treated in the next section), **300** is a *huge*, *unacceptable* value to bound the Depth-Search routing: exactly for this reason, if it is true that the swapping procedure brings benefits on the routing algorithm, *after some time* this value should not affect the number of required steps.

The analysis is based on performing a GET request per unit of time, by 1.500 pseudo-randomly chosen peers of the overlay network, for a period of *256 units of time*. The overall situation sees 1.500 random requests performed per unit of time, so a total of *384.000 requests* during each simulation. Each GET request searches for a location key that is randomly chosen from a set of 1.500 already successfully inserted and replicated location keys.

In each unit of time, all the peers performs a SWAP request, with no exception for those that does not performs a GET request, in order to obtain a better clustering configuration and optimize the routing of the next requests, until the **convergence** of the swapping algorithm has not been reached.

The results of this analysis for values 0 (*no swap*), 2, 6 and 10 of the parameter **maxHTLswap** are shown in *Figure 10*.

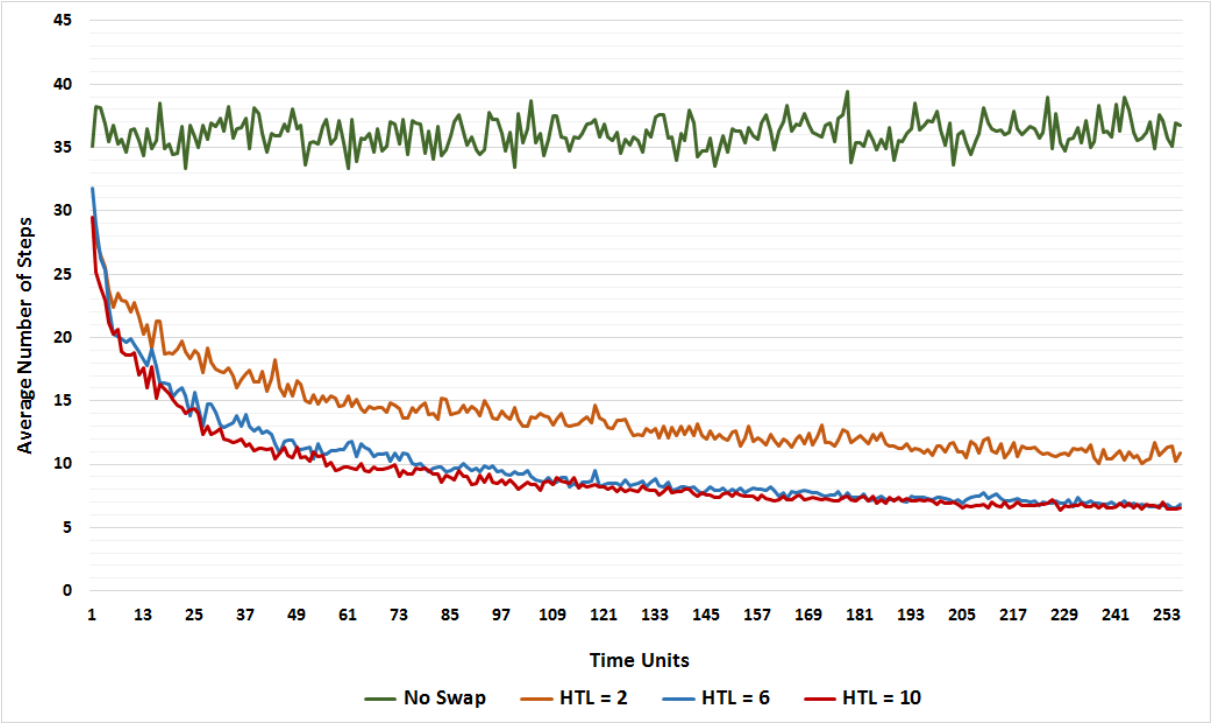



*Figure 10: the trend of the average number of steps required to answer to GET requests, for different values of the parameter maxHTLswap.*

As expected, we can see that by effect of the swap algorithm, the average number of steps required to answer a request **highly decreases** in the time, of course unlike the case in which the swapping algorithm does not runs. Furthermore, we can notice that for increasing values of the parameter *maxHTLswap*, the swap algorithm reaches the **convergence** more rapidly, but for values greater or equal than 6, the improvement becomes minimal and the time of propagation of the request could slow down the convergence. Once that the convergence of swap algorithm has been reached, we can notice a stabilization of the number of required steps between 5 and 10: since the Data Set have $N = 7190$ nodes, the theoretical result of expected $O(\log N) \simeq 13$ steps is reached.

However, since for hypothesis all the keys are present in the overlay network and **maxHTL = 300**, we get **100%** of the GET_FOUND answers in all cases, even when the swapping algorithm is not executed.

## 4.2 Key replication tuning

Similarly to the previous section, in this section we start observing the trend of the average number of routing steps that a GET request requires to receive a positive or negative answer, and the impact of the swapping algorithm on it, but now observing the trend in function of the **replicationFactor** parameter.

In the previous section, we have set *replicationFactor = 10*, supposing that 10 was a good value for the relative parameter. So, in this section, we justify why this is a good value, analyzing the **impact of the maximum number of times that a content location key is replicated on the routing performance**, after an its successfully insertion in the overlay.
In doing that, we fix the other two parameters to **maxHTLswap = 6** and **maxHTL = 18**: we say, from the previously discussed analysis, that **6** is a good value for the swapping algorithm, while we suppose that **18** is also a good value for the Depth-First Search bound (analysis treated in the next section).

The current analysis is based, again, on performing a GET request per unit of time, by 1.500 pseudo-randomly chosen peers of the overlay network, but now for a period of *110 units of time*, because we are not interested on see the trend for a long period of time. The overall situation sees 1.500 random requests performed per unit of time, so a total of *165.000 requests* during each simulation. Also in this case, each GET request searches for a location key that is randomly chosen from a set of 1.500 already successfully inserted and replicated location keys, but this time the number of times that each content location key is replicated is the variable of the analysis.

Furthermore, also in this case, in each unit of time all the peers performs a SWAP request, in such a way to see the impact of the swapping algorithm also in function of the replication factor.

The results of the analysis for values 4, 6, 8, 10, 12 and 14 of the parameter **replicationFactor** are shown in *Figure 11.*

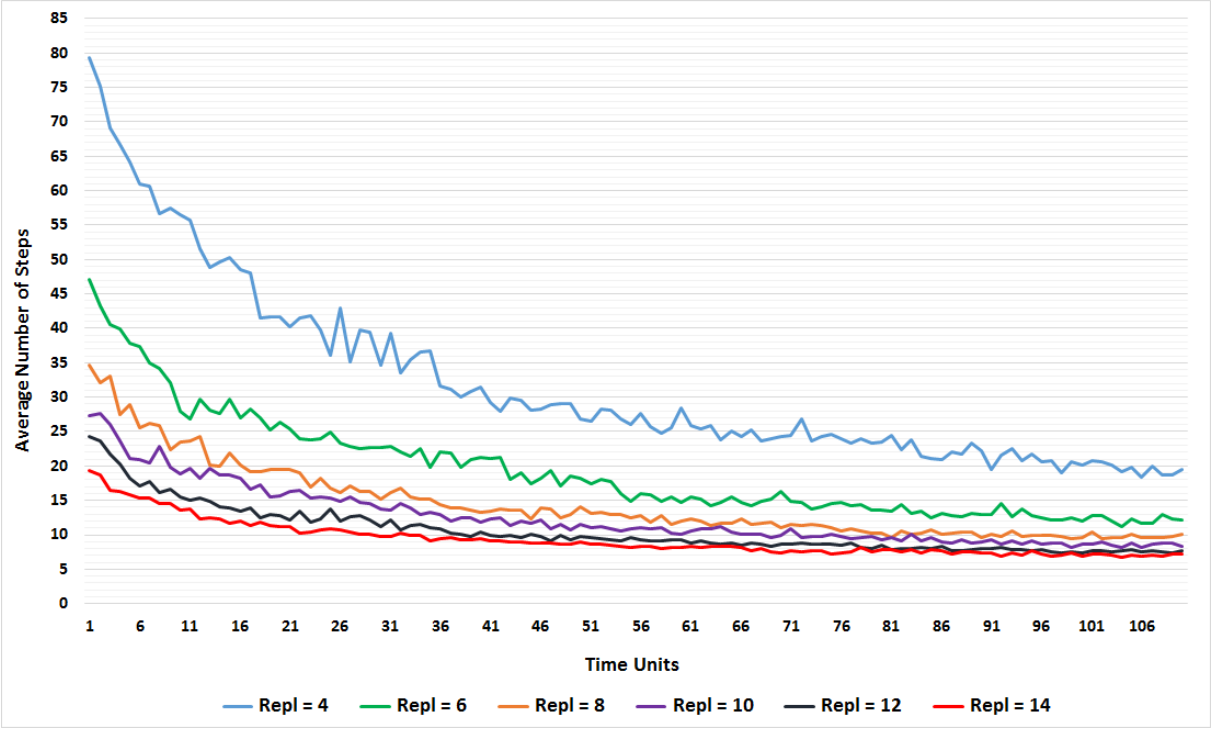



*Figure 11: the trend of the average number of steps required to answer to GET requests, for different values of the parameter replicationFactor.*

As expected, we can notice that in the first part of the simulation, so in the first units of time, a **low replication factor** could lead to *destructive results for the routing performance*, with an average number of required steps also in the range $[35, 80]$, even if, later and much more slowly, the swapping algorithm leads evenly to evident improvements.
On the other hand, an **high replication factor** could lead to an *infective replication* of the content location key on too many nodes, maybe neighbors, occupying inefficiently their local storage. Furthermore, it can lead also to a *zero-impact of the swapping algorithm*, since the action of cluster peers with similar location have no sense if each content location key is stored in almost every node of the overlay.

Instead, changing type of analysis and remembering that a content location key cannot be found even if it is present in the overlay, for exact the same simulation we can study also the **percentage of positive and negative answers** relative to the performed GET requests, w.r.t. the total number of performed requests. The detailed results of this analysis are shown in *Figure 12*.

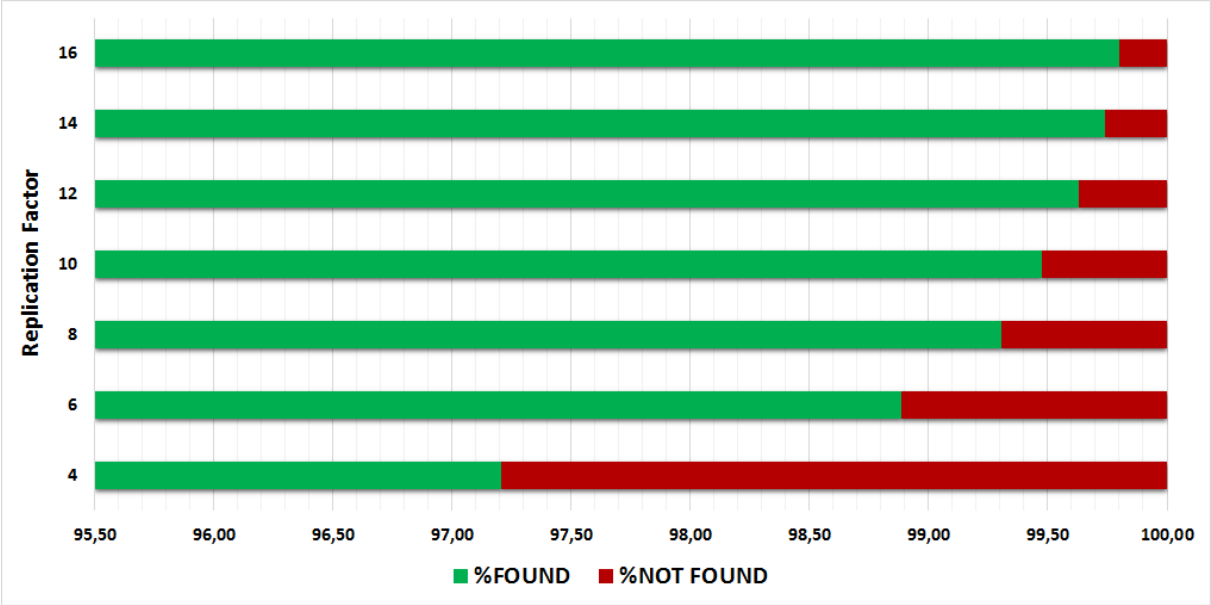



*Figure 12: overall percentage of positive and negative answers relative to the performed GET requests.*

As is clear from the above figure, the use of optimal values for the swapping algorithm and the routing bound leads to an high overall percentage of GET_FOUND, not less than about **97%**, independently from the value used for the replication factor.
However, we can note some **considerable improvements** increasing the replication factor from 4 to 6, 8 or 10, but **negligible improvements** increasing it from 10 to 12 or 14, considering also that bigger is the value of the replication factor, more peers occupy its local storage to store the same content location key.

In conclusion, we can deduce that a solution that offers a good **trade-off** between storage space management, an helpful impact of the swapping algorithm and the speedup of the routing algorithm, is that of choice **replicationFactor** $\in [8, 10]$ as, indeed, we have already done in the previous section.

## 4.3 Routing bounding tuning

As third, last analysis, we proceed as in previous sections, but exploiting the last possible combination between the treated parameters: we fix **maxHTLswap = 6** and **replicationFactor = 10**, which we have seen to be good settings for the respective parameters, and observe the trend in function of the parameter **maxHTL**.

The analysis consists in performing 1.000 GET requests per unit of time, each of which is performed by a randomly chosen peer of the overlay and searches for a random chosen already successfully inserted and replicated location key, for a duration of *105 units of time* and a total of 105.000 requests.

We start analyzing the overall **percentage of positive and negative answers** induced by the performed GET requests, in function of the parameter **maxHTL**, in the cases in which the swapping algorithm is not executed (*Figure 13*) and in which, instead, it runs using the fixed parameter described above (*Figure 14*).

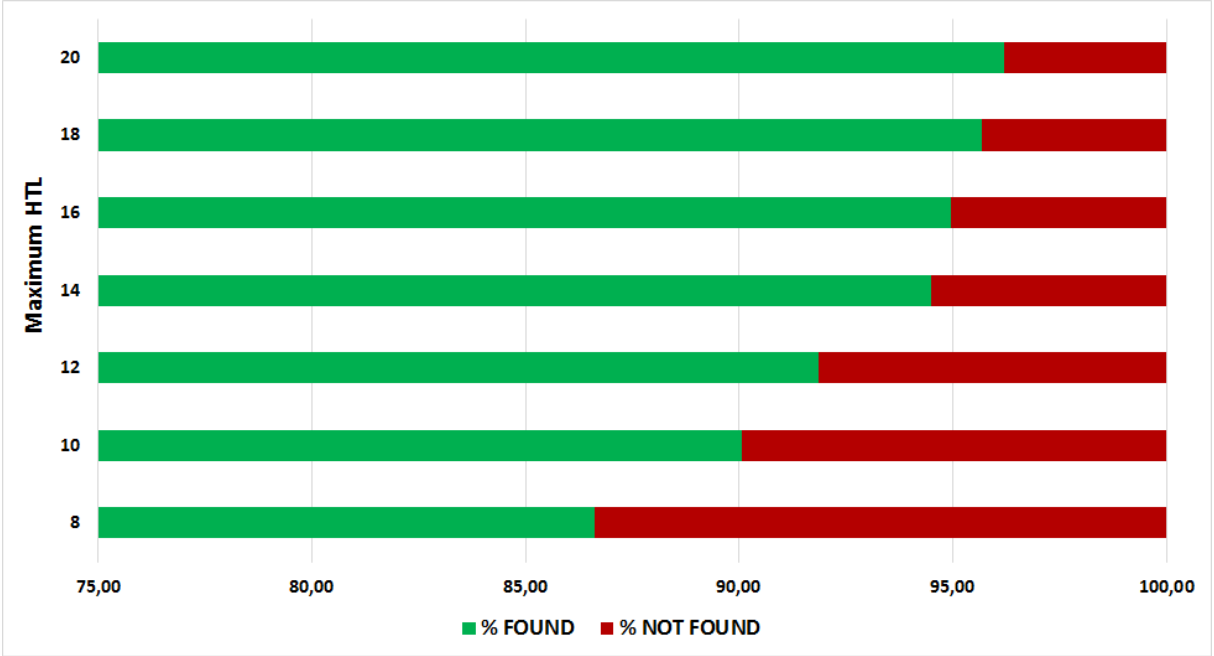



*Figure 13: overall percentage of positive and negative answers relative to the performed GET requests, in the case in which the swapping algorithm does not runs.*


*Figure 14: overall percentage of positive and negative answers relative to the performed GET requests, in the case in which the swapping algorithm runs.*

Finally, we conclude analyzing the trend of the **average number of routing steps** that a GET request requires to receive a positive or negative answer, in function of the parameter **maxHTL**, in the cases in which the swapping algorithm is executed (*Figure 15*).

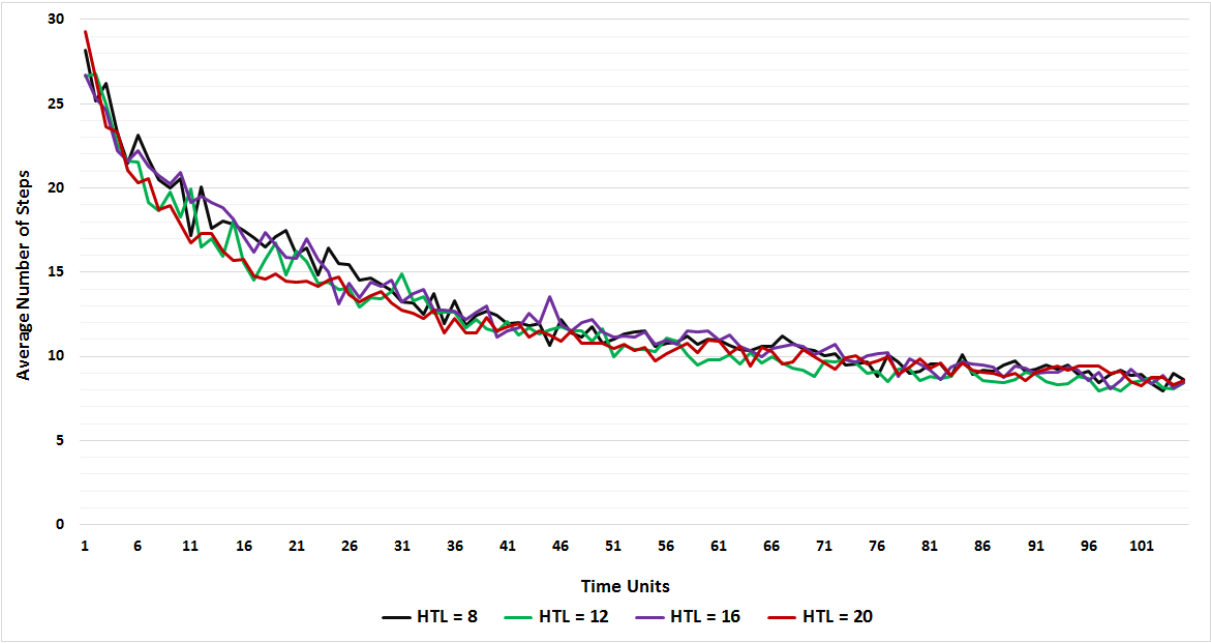



*Figure 15: the trend of the average number of steps required to answer to GET requests, for different values of the parameter maxHTL.*

As expected, is evident due of the strong overlapping of the functions in the figure above that the variations of the parameter **maxHTL** per se **does not influence the average number of steps used by the routing algorithm**, which by hypothesis is executed in an environment that presents good values of the other two parameters and so reaches the convergence quickly. On the other hand, however, we can notice that instead the variations of the same parameter **affects the percentage of positive and negative responses**: in the case in which no swaps are performed during the simulations, for the examined values the percentage of GET_FOUND answers may decrease up to over about **85%**, while, in the other case, the percentage is ever greater than about **98%**.

## 5 Conclusions and Comparison with real parameters

At the end, we can note that the configuration **maxHTL = 18**, **replicationFactor = 10** and **maxHTLswap = 6**, allows us to obtain a good trade-off that balance the routing performance, so speed on convergence of the swapping algorithm, the number of positive answers and the memory usage, and so that can be used as "**default setting**" for the use of the implemented system on large scale small-world networks.

Wanting to make a comparison with the parameters used by the **real implementation** of Freenet, we can discover that:

- the community have used, in the time, values for **maxHTLswap** between 6 and 10 and, currently, the value used is precisely 10 (search for "*SWAP_MAX_HTL*" in [13]);
- the used value for **maxHTL** is exactly 18 (search for "*DEFAULT_MAX_HTL*" in [14]);
- the used value for **replicationFactor** is "all the neighbors", as described in the protocol specifics [4].

At the end of the work, it's nice to note that the results of the various statistics carried out on their own implementation of a minimal part of the Freenet System, lead to results very similar to those expected on the real implementation.

## References


[1] *PeerSim: a Peer-to-Peer Simulator*
http://peersim.sourceforge.net/

[2] *The Freenet Project Wiki*
htts://wiki.freenetproject.org/Main_Page

[3] *Freenet REference Daemon*
https://github.com/freenet/fred

[4] Nathan S. Evans, Chris GauthierDickey, Christian Grothoff. *Routing in the Dark: Pitch Black*. Colorado Research Institute for Security and Privacy and Department of Computer Science University of Denver, USA.
http://grothoff.org/christian/pitchblack.pdf

[5] Ian Clarke, Chris Mellish. *A Distributed Decentralised Information Storage and Retrieval System*. Division of Informatics, University of Edinburgh, 1999.
http://freenetproject.org/papers/ddisrs.pdf

[6] *What is Freenet?*
http://freenetproject.org/whatis.html

[7] O. Sandberg. *Searching in a Small World*. PhD thesis, University of Gothenburg and Chalmers Technical University, 2005.

[8] From Wikipedia, the free encyclopedia. *Floyd-Warshall Algorithm*
http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Algorithm

[9] From Wikipedia, the free encyclopedia. *Red-Black Tree Data Structure*
http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

[10] *Peersim: Uniform Random Transport Class*
http://peersim.sourceforge.net/doc/peersim/transport/UniformRandomTransport.html

[11] *Peersim: CDScheduler Class*
http://peersim.sourceforge.net/doc/peersim/edsim/CDScheduler.html

[12] From Wikipedia, the free encyclopedia. *Metropolis-Hastings algorithm* http://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm

[13] *Freenet GitHub Repository: LocationManager Class* https://github.com/freenet/fred/blob/f3ddb7a9357d878a321ecfda07c7414458ca44da/src/freenet/node/LocationManager.java

[14] *Freenet GitHub Repository: Node Class* https://github.com/freenet/fred/blob/f3ddb7a9357d878a321ecfda07c7414458ca44da/src/freenet/node/Node.java