Freenet 0.7 Darknet simulation with PeerSim

Luca Lovagnini

June 22, 2017

1 Introduction

In this project is presented an implementation of Freenet 0.7 Darknet [EGG07], an algorithm to implement small-worlds networks where nodes direct communication between nodes is limited, e.g. mobile ad-hoc networks or friend-to-friend overlay networks, where the user want to maintain his privacy.

This project has been implemented in Java with the support of the PeerSim [1] to simulate the behavior of the implemented algorithm on large networks. In particular, two different networks of different size created will be used to evaluate the proposed implementation. Both of these networks are created from Facebook to represent a friend-to-friend overlay network as much realistic as possible.

As we will see in the evaluation section, unfortunately the presented implementation does not achieve the logarithmic complexity for look-up operations expected by small-world networks. However, the results are similar to what we expect, and we achieve the desired complexity on smaller networks. Finally, we will try to give explanations about these results.

2 System Description

In this section we are going to describe Freenet 0.7 in details, in particular the routing protocol and the swapping operation.

Freenet 0.7 main feature is to store contents in a DHT fashion, where user nodes and stored objects are organized along a circle, similarly to Figure 2, which represents the DHT used in Chord. Many popular systems such as Symphony [MBR+], Dynamo [DHJ+07] or Chord [SMK+01] itself have shown the efficiency of Distributed Hash Tables to store and route contents between many nodes. However, in all these works each peer had a "global" vision (i.e. he could directly contact any other peer in the system), or at least each peer could contact any other peer in the network. Instead, in Freenet 0.7 we want to implement a friend-to-friend network (or darknet), where the priority is to obscure the participation of a node in the network. As consequence, each node can see (and can be seen) only by a small set of neighbors (or "friends") and each node peer can contact untrusted peer only by friend-to-friend links. This requirement introduces a great challenge to create a small-world network.

Each node or content in Freenet 0.7 is mapped to the range $[0, 1)$ and despite many other systems, each *key* (referred also as *location key*), which is used to uniquely identify a peer, can change over time (as we will see soon).

As it usually happens in DHT systems, the distance D between two keys k_1 and k_2 is measured as:

$$D(k_1, k_2) = \min(|k_1 - k_2|, 1 - |k_1 - k_2|$$

This is true both for nodes and stored content keys.

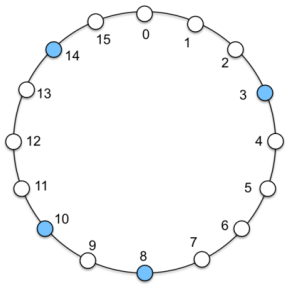

Figure 1: Local ring in Chord

2.1 Routing Algorithm

Following are summarized the Freenet 0.7 requirements and features that we described in the previous section:

[1] https://peersim.sourceforge.net/

- Network based on a *cyclic address space*.
- Each peer has a constant set of connections which *will never change* in any circumstance.
- Data is assumed to be stored at peer with the *closest identifier*.
- Routing algorithm is based on key-proximity in a depth-first greedy fashion.

Given the undirected graph $G = (N, E)$ used to represent the overlay network, we can now define the three main operations that can be performed in Freenet 0.7:

1. *Get(i,k)*: the node $i \in N$ forwards a request for a content with key k to the system. A depth-first algorithm with a limited number of Hops-To-Live (HTL) will be described in section 2.1.1 in details. Notice that because of the nature of the algorithm and the presence of a finite HTL, it is possible that k is not found by the request. In that case a "content not found" message is routed back to i, otherwise an "ok" message will be received by i with the desired content.
2. *Put(i,k)*: the node $i \in N$ inject the content with key k into the network. The content will be routed in a greedy fashion to a node $j \in N$. According to the original paper, *"a* `put` *operation is routed in the same fashion and reaches exactly the same peers as an unsuccessful* `get` *operation"*. Notice that this does not guarantee that j is the closest node in the *whole* network w.r.t. k, i.e. there could a node $z \in N$ s.t. $D(z, k) \leq D(j, k)$. A replication algorithm is introduced in order to increase the probability of a successful *get* operation and to increase its efficiency. Finally, no backward message is sent to i when k is stored in j.
3. *Swap(i,j)*: given two nodes $i \in N$ and $j \in N$, we try to swap their *location keys, without changing any of their neighbors*, as it will be described in section 2.1.3. Notice that this means that we swap their stored keys along with their location keys too. Why do we need this operation? As explained in the Kleinberg model, each peer should have many short range link, while a few long range ones. Since we can not change the trusted peers/friends of a given node, the swapping algorithm tries to achieve this feature by swapping location keys between neighbors (if some conditions are satisfied). We want to stress that this operation does not alter each node neighborhood, but only the content location of the two swapped peers. According to the Sandberg theorem, the swapping action will make converge the location keys so the number hops needed by a `get` or `put` operation will take only $\mathcal{O}(\log(n)$ steps to be performed (with high probability).

Notice that in our implementation we do not actually stores any kind of content, since the main purpose of this project is to simulate the routing process proposed by Evans et al in the original work. Following is described each operation in details. Describing the three operations previously reported is not a simple task, especially because only the `get` algorithm is described in details. In particular, the replication performed after a put operation is not described at all, except for Figure 5 of the original paper. In addition, many important details present in the open source [2] are omitted in the original paper. For example, `SWAP_MAX_HTL` explains that each swap not necessarily performed between two neighbors node, but it could happen also between two peers not directly linked, which could greatly affect system performance. However, this project tries to implement the original routing protocol, without introducing any new element.

In the following sections (where operations are described in details), we use the same annotation used up to now. To understand them better, as we will explain in details in section 3, apart from the `put` messages received during the replication process, no message is received twice by the same peer.

2.1.1 Get Operation Algorithm

Luckily, this is the only algorithm presented in details in [EGG07]. Supposing that node j receives the get request looking for the content key k:

1. j looks for k in his set of stored keys. If k is found, send a backward message with content found, otherwise continue.

[2] https://github.com/freenet/fred

2. If j is closer to any than any previously visited peer, then reset HTL to the original value.

3. If all j's neighbors have already received this message or HTL is zero, then send back a content not found message. Otherwise, supposing that z is the closest neighbors of j w.r.t. k (and z has not received this message yet), then add j to the set of visited peers and forward the message to z.

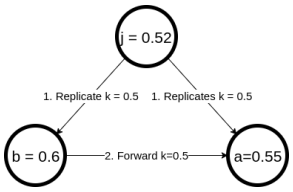

Figure 2: Case where `put` with key $k = 0.5$ is received twice by `a`

Notice that the steps reported above are a little bit different from the ones described in the original work, but that's because in this implementation the same message can not be received twice from the same peer, as it will be shown in section. In section 3 we will describe what happens if a node receives a content not found message.

2.1.2 Put Operation Algorithm

These are the steps performed in by a node j which receives a put operation for the key k:

1. If j has stored k, then simply ignores this message.

2. If j is closer to k w.r.t. *any* of his neighbors, then stores k.

3. j looks for a peer z which didn't received the message already. If z doesn't exists, than the `put` requests dies (this cannot happen if the message has never been stored yet). Otherwise continue.

4. If j is closer to any than any previously visited peer, then reset HTL to the original value.

5. If this is the first time that the message has been stored somewhere, start the replication process by forwarding the message to every neighbor (even the ones which already received the message already).

Notice the first step: according to our implementation assumption that each node can't receive the same message twice, the first step should never happens. However, when we consider replication, this scenario can happen as shown in figure 2.1.1. Finally, it's worth to spend a few words on the third case: an obvious case where this can happen is during replication, where all j's neighbors already received the message. But a more complicate case is shown in Figure 2.1.2: node a forwards his `put` message to b, but before he receives the message they swap their location key. As conclusion, b receives a put message which shouldn't have received. As consequence, the message is not forwarded to a (since he already received it), but it can't be stored in b neither. It is not explained what to do in this case in the original work, but since the `put` requests has to be treated similarly to `get` requests, and since it is specifically reported written that any previously message has to be discarded, as final result we have that the `put` message "dies" without being stored somewhere.

As already said, the replication process is not well explained in the original paper. For example: supposing that j stores k, which is replicated to z as result of the replication process, then z should repeat the replication process? If so, how many times should we repeat it? The process can not be repeated an unlimited number of times (e.g. the key is stored in every node of the network), but it could make the `put` operation prohibitively expensive and the key could be replicated on inappropriate nodes. Notice that in the original paper is not explained what to do when a *put* message dies during the replication process: should we replicate the key on the last node or not? In the first case, the key is replicated a number of times equals to the j's degree, incrementing the probability of a successful get message. On the other hand, this could lead to a key pollution, where keys are stored in nodes which location key is not so close to the content key. This could raise another question: should we replicate the content to all j's neighbors, or only to the top-K (where K is a tuning parameter) closest neighbors w.r.t. k or only to the neighbors which distance to k is lower than a given threshold? Finding an answer to this question is beyond the scope of this project, but it's clear how all this lack of details can bring to bad performance.

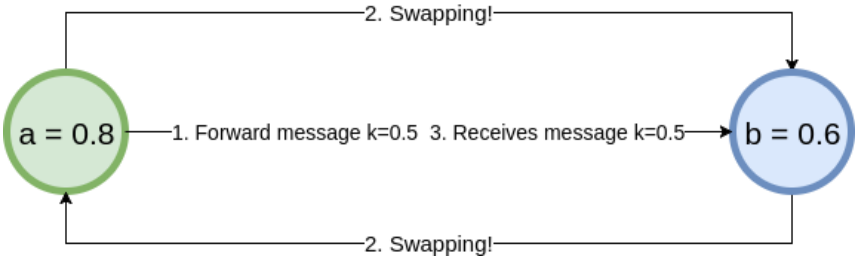

Figure 3: Case where `put` fails because of swapping.

2.1.3 Swap Operation Algorithm

Supposing that node i start the swap operation:

1. i randomly selects one of his neighbors j.
2. Then it computes $D_1(i,j)$ and $D_2(i,j)$, which are reported in Equations 1 and 2, respectively (where function $L(n)$ gets the location key of the given node n).
3. If $D_1(i,j) \leq D_2(i,j)$ they swap their location keys (and their stored objects), otherwise swap with probability $D_1(i,j)/D_2(i,j)$. The deterministic swap always decreases the average distance of nodes with their neighbors. The probabilistic swap is used to escape local minima.

$$D_1(i,j) = \prod_{(i,n)\in E} |L(i) - L(n)| \times \prod_{(j,n)\in E} |L(j) - L(n)| \tag{1}$$

$$D_2(i,j) = \prod_{(i,n)\in E} |L(j) - L(n)| \times \prod_{(j,n)\in E} |L(i) - L(n)| \tag{2}$$

Unfortunately, as we will seen in section 4 for this implementation the formula above will not help to decrease the average number of hops for `get` requests.

3 Implementation

In this section details about the implementation of this project. There are three Java packages in the project:

3.1 `control`

3.1.1 `KeysGenerator`

As we explained in section 2, each node or stored object is represented by an unique identifier. This class implements static methods to generates the keys used in this project.

In particular, at the begin of the simulation, a number of keys (specified by `maxKeys`) are generated sequentially, then are randomly shuffled. Every time that a node is created (in `Setup`) or an object is going to be injected in the system, the next key is returned. This process guarantees that the generated keys are unique. In addition, the class stores the keys generated by a `put` request, so when we perform a `get` operation we randomly pick up one of the generated keys that were already store in the system in order to guarantee that `get` operations are performed on already stored keys.

3.1.2 Setup

This class reads the dataset files (described in section 4.1) to initialize the overlay network before the simulation begins.

3.1.3 Statistics

This class is executed by *PeerSim* at the beginning of the simulation. For the given overlay networks, several statistics are computed:

1. The graph diameter.
2. The average shortest between each pair of nodes in the overlay network.
3. The degree of each node in the system.
4. The cluster coefficient of each peer.

Notice that in order to compute the first and second statistics, the shortest path (implemented in `shortestPath`) between each pair of peers has to be computed.

3.2 `dataStructure`

This package is the most important one of the entire project, where active actors in our system (such as messages and peers) are implemented and described in the following sections. Notice that all the kind of messages are implemented in the sub-package `message`.

3.2.1 `DarkPeer`

This class implements each peer used in Freenet 0.7. In particular, the keys of the (fake) stored objects are stored in `storedKeys`, as well the (possibly mutable because of swapping) `locationKey`. As we will explain in the `LinkableProtocol` class, `DarkPeer` objects are compared based on their `locationKey`, and that's why explain the implementation of `compareTo`. In addition to methods to mange stored content keys, in this class is implemented the distance function D defined in section 2.

3.3 `Message`

This abstract class implements one of the most crucial aspects of this project. The abstract method `doMessageAction` will have to be implemented by every class which extends `Message` and it defines the behavior of the concrete message. By exploiting polymorphism, this makes extremely easy (to implement `processEvent` and `nextCycle`, which represent the main methods for the event-based and cycle-based engines in our simulation (as explained in the section about `MessageProtocol`), respectively.

The main field in this class is `allPeersVisited`, which keeps track of each all the peers visited during the routing process. In particular, for `ForwardMessage` it will prevent to send the same message twice to the same peer (except during replication, as explained in section 2.1.2, while for `BackwardMessage` it will be passed again to an eventually generated `ForwardMessage` to continue the routing process. We will explain this in more details in `GetNotFoundMessage`.

3.3.1 `ForwardMessage`

This abstract extends `Message`, without implementing `doAction`. It represents the messages relative to an user request, i.e. `put` and `get` operations implemented in `PutMessage` and `GetMessage`, respectively.

Differently from `BackWardMessage`, `ForwardMessage` have a `HTL` field, which decides either a given message can be forwarded or not. Finally, the `bestDistance` field is the smallest distance between the content key and any other location key of the peers visited. As explained in section `putAlgorithm`, this is used to reset the HTL when the key of the peer which received this message is closer than another peer previously visited.

3.3.2 `BackwardMessage`

Also this abstract class extends `Message`, which is used to represent all the messages which are created as answer to another message, namely `GetNotFoundMessage` and `GetOkMessage`.

The `routingPath` field is a stack which is used to know what will be the next peer to visit during the rolling back process, until the node who created this message is reached.

3.3.3 PutMessage

This class implements in `doMessageAction` the `put` operation that we described in section 2.1.2.

Notice the `replicationFactor` parameter, which is defined in the `config.cfg` file (used to initialize the simulation): as we explained, we deduced that the replication process (where a peer forward who stored a key forwards its `put` requests to *all* his neighbors), in that case `replicationFactor=1`. But in case we want to replicate the injected object more aggressively, we can set it to a value bigger than `1` and repeat the replication process several times. Every time that a node stores a new key, `replicationFactor` is decreased.

3.3.4 GetMessage

The `doMessageAction` of this class implements the `get` operation described in section 2.1.1.

First of all, a constructor is provided when we create a new `GetMessage` after the node has received a `GetNotFoundMessage`. This happens when a peer who forwarded a `GetMessage` referring to the same `messageLocationKey` (so the same user request) forwards a `GetMessage`, but then it receives a negative answer from the receiver peer. Then, following the depth-first algorithm, he forwards again a new `GetMessage` (referring to the same user request) to another peer which didn't receive the message yet.

Finally, every time that we forward this message to a neighbor, the peer who is processing this message is added to `routingPath`, which will be used in future by `GetNotFoundMessage` or `GetOkMessage`

3.3.5 GetOkMessage

The action executed by this message in `doAction` is very simple: pop the next peer contained in `routingPath` and forward this message to the previous peer in the routing process.

3.3.6 GetNotFoundMessage

This message is created whenever a peer does not store the searched key and `HTL=0` or all the neighbors already received the correspondent `GetMessage`.

The first check performed in the `doMessageAction` of this message checks is if $HTL \geq 0$ (obtained from the correspondent `GetMessage`, here represented as `getMessageHTL`): if this is not the case, then we have the only choice that we have is to keep going on the routing back process. Otherwise, we try to find the closest neighbor which has never seen this `get` request before: if such a neighbors exists, then we forward a net `GetMessage`, passing to it all the necessary information, such as:

- `allPeerVisited` to avoid already visited peers.
- `getBestDistance` which is the correspondent `bestDistance` of the correspondent `bestDistance`.
- `getMessageHTL`.
- `originalMessageId` used to uniquely identify each message request.
- `originalHTL` in case we will reset `HTL` in the future to its initial value.

If this neighbor does not exist, we keep forwarding back this message by using the `routingPath`.

3.4 protocol

3.4.1 LinkableProtocol

In this class are implemented all the aspects about the neighborhood management of each peer.

The only field `darkNeighbors` stores a reference to all the peers linked to the considered node. A set of methods to implement the swapping mechanism (described later) or to initialize the network are provided (i.e. `addNeighbor`, `removeNeighbor`) or to get information about the neighbors are used: for example, `isClosestThanNeighbors` is used by `PutMessage` to understand if this peer is the closest one to the `put` content key than *any* of his peers, while `getClosestNeighbors` returns a reference to the closest neighbor w.r.t. the given key who did not receive this message already (or returns `null` if a suitable neighbor does not exist).

Name	Size	Diameter	Average Shortest Path	Average Degree	Average Cluster Coefficient
Small	198	2	1.94	9.6	0.57
Large	7190	4	3.55	12.3	0.24

Table 1: Information about the datasets used for evaluation.

3.4.2 `MessageProtocol`

This class, in addition to implement the cycle and event engines methods, implements also the swapping operation described in section 2.1.3.

Depending on the considering git branch, the `nextCycle` implementation is different:

1. In case we are on the `master` branch, then each peer randomly performs either a `put` or `get` request. This approach tries to simulate a realistic scenario where both operations are randomly scheduled.

2. In case we are on the `2000PutGetTest`, then the first peer at the first cycle generates 2000 `put` requests using all the nodes around the overlay network. Otherwise, 2000 `get` operations are performed at each cycle. This simulation approach will be used in section 4 to evaluate this implementation.

In both cases, every peer at the end of his cycle performs the `swap` operation by calling `tryToSwap`. Notice that to implement this operation, the `TreeSet` used in `LinkableProtocol` makes everything a little bit more complicated, since in order to maintain the data structure consistent and sorted, we need to remove the peer before updating his location key (otherwise the structure would be broken since the order between a pair of peers depends on their location keys).

This class implements also a static method `writeStatistics` so we can write on file useful information about a request result (i.e. if the obtained result is either a `GetOkMessage` or a `GetNotFound` message, `HTL` or the total number of hops used to complete this operation.

Finally, it's worth to notice the simplicity of `processEvent` implementation (used for the event-based engine): it simply call `doMessageAction` for any kind of received message, nothing more. This mechanism is very flexible also in the case we want to introduce a new kind of message, making the code maintainable.

4 Evaluation

In this section we are going to evaluate the performance of the Freenet 0.7 implementation provided by this project.

4.1 Datasets

We will use two datasets obtained from real friends relationship on Facebook, namely:

1. `FacebookDataset_Small.csv` containing 198 peers (*Small* in short).

2. `FacebookDataset_Large.csv` containing 7190 peers (*Large* in short).

In Table 1 there are reported many information about the used datasets. From this table we can understand that we are in a small-world scenario, where thousands of nodes are separated by less than five nodes (in the optimal case), this is also justified by the high cluster coefficient of a few nodes. But even more important to justify the presence of a small-world scenario are the plots shown in Figure 4.1 and its log-log scale version 4. In the plot we can see that both datasets are not proper power laws (as by the distance from the trending lines which represent per power laws), since the distribution does not form a straight line, but we can still say that the node degree for both datasets "almost" follow a power law distribution.

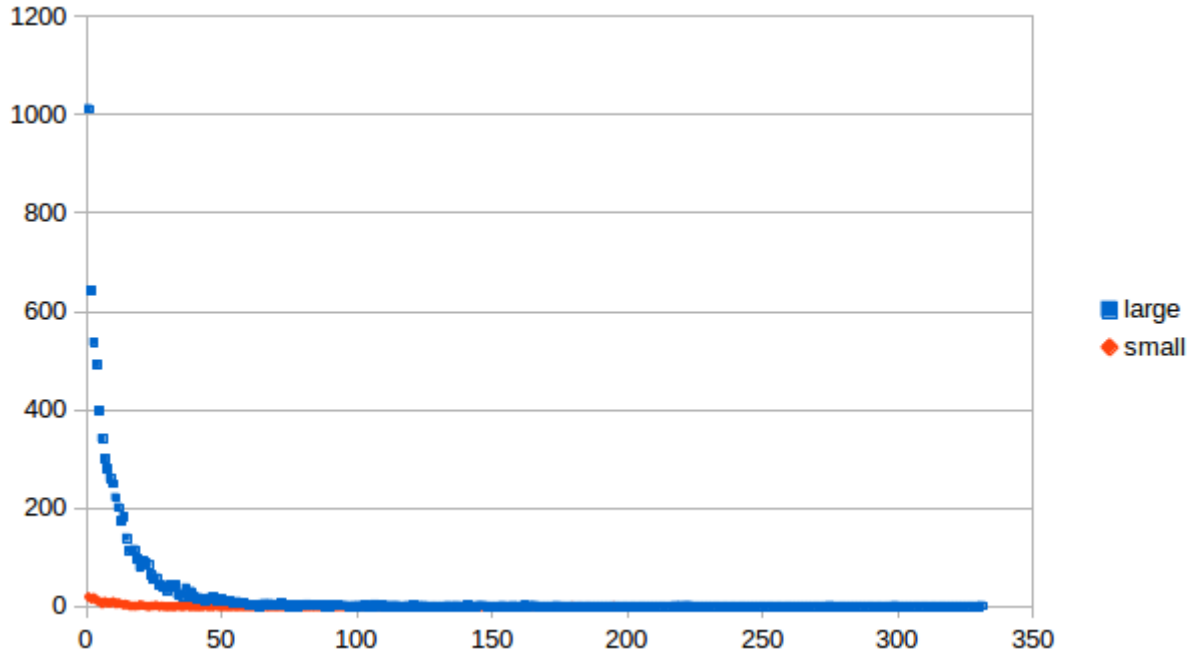

Figure 4: Node degree distribution of evaluated datasets.

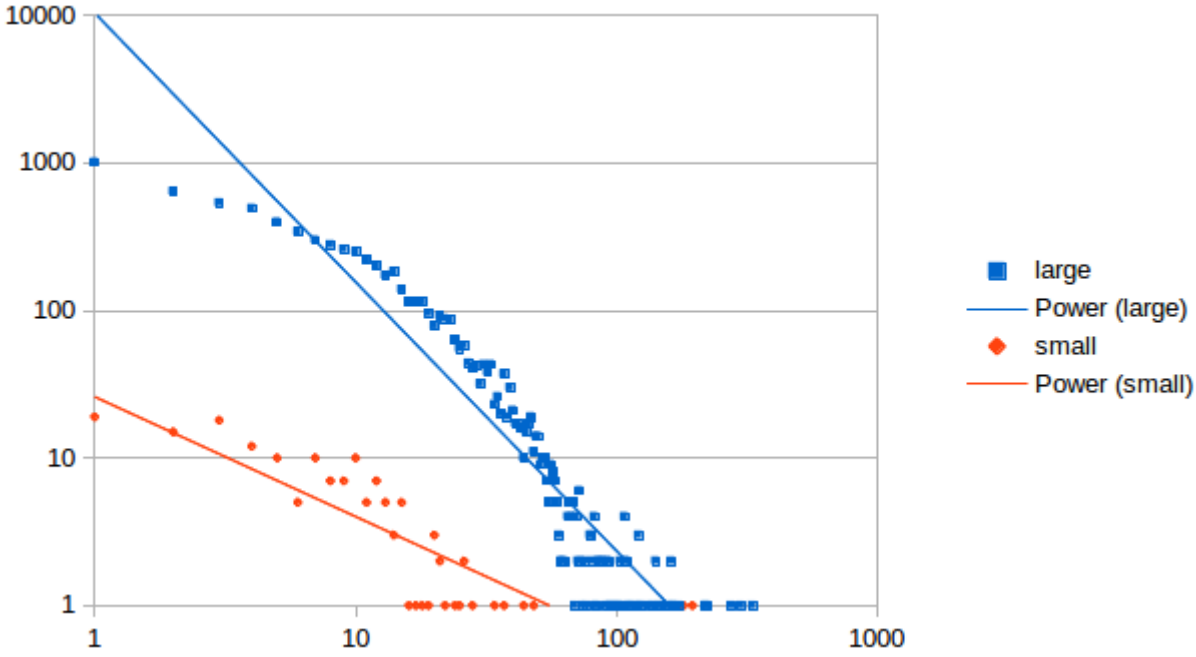

Figure 5: Figure 4.1 in a log-log scale, including trending lines.

4.2 Experiments

In this section we are going to describe different experiments on both datasets in order to evaluate their routing performance:

- The Found/Not Found ratio depending on both parameters `HTL` and `replicationFactor`.
- The number of hops for `get` requests depending on `HTL` and `replicationFactor`.
- How the `swap` mechanism affects routing performance.
- How keys are distributed between nodes depending on `replicationFactor`.

All the experiments were conducted by initializing the network with 2000 `put` operations. Then, 2000 random `get` operations are executed with same parameters (depending on the considered analysis).

4.2.1 The Found/Not Found ratio

As we can see Figure 4.2.1, `HTl` influences the success ratio of this implementation, especially for the large scale experiment: for the smaller dataset we find more than 99% of messages, but for the larger dataset even by using `HTL=20` almost 20% of messages are not found.

In Figure 6 we can see that the average number of hops necessary for `GetOkMessage` is always lower the logarithmic threshold w.r.t. the network size, but still, we have to consider that almost of 20% of the requests are not satisfied. As we can see, best results are obtained by using `HTL=18`. Finally, notice that as expected the number of hops is larger always larger than the correspondent `HTL`, since the number of remaining hops is reset during the process.

4.2.2 `replicationFactor` tuning

The results of this experiment are reported in Figure 4.2.2, where we used `HTL=18`. Notice that since we already obtained excellent results with smaller network (over 99%) we only reported results for the bigger network.

As we can see, when we decide to repeat the replication factor more than once, the results improve significantly: with `replicationFactor=3` more than 99% of successful `get` operations. However, looking at Figure 8 the number of hops is greatly smaller than the 12, which is around the logarithm of the network size. This means that probably each key is greatly replicated all over the network, and so each node is close to most of the keys.

4.2.3 Swap effect

In this experiment we test the swapping effect defined in the original paper. We used `HTL` and we look how the system behavior by using `replicationFactor = 1,3,5`. Unfortunately, as it can be seen from Figure 4.2.3 the number of hops per request doesn't change over time, but it's almost constant, i.e. the expected swapping effect does not happen. We already talked in the previous sections why this could happen, however the author didn't find a concrete reason for this.

5 Conclusions

In this project report we have seen a Java implementation using *PeerSim* to simulate the Freenet 0.7 routing algorithm. Unfortunately, the implemented swapping algorithm presented in the original work did not converge to the expected logarithmic complexity for `get` requests on large scale experiments, while in small scale graph we obtained the desired results. However, by implementing an aggressive replication policy, we obtained the expected result (even better than logarithmic complexity for `get` requests). The author did not find a an answer to this problem (and he didn't sleep one whole night trying to solve it) without success, but one of the possible reasons is the lack of implementation details on the original work, which can greatly influence final performance.

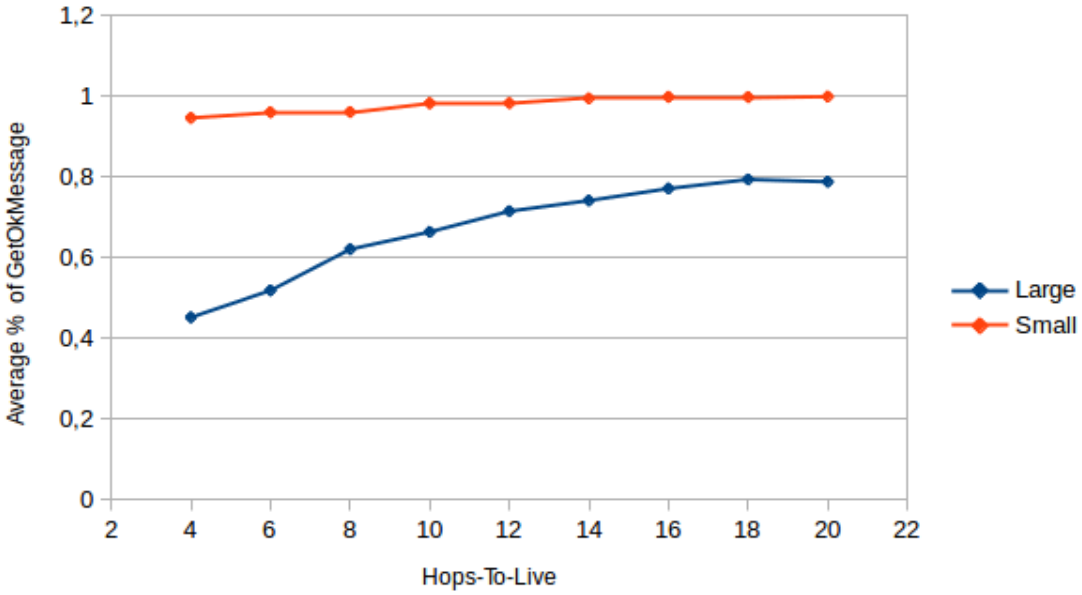

Figure 6: Get success ratio in relation of `HTL`.

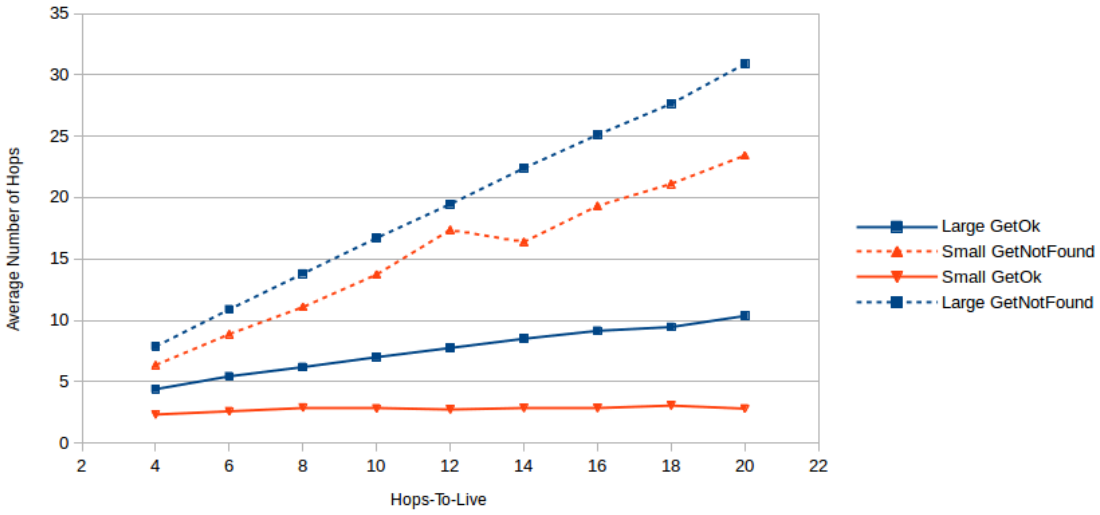

Figure 7: Average number of hops depending on `HTL`.

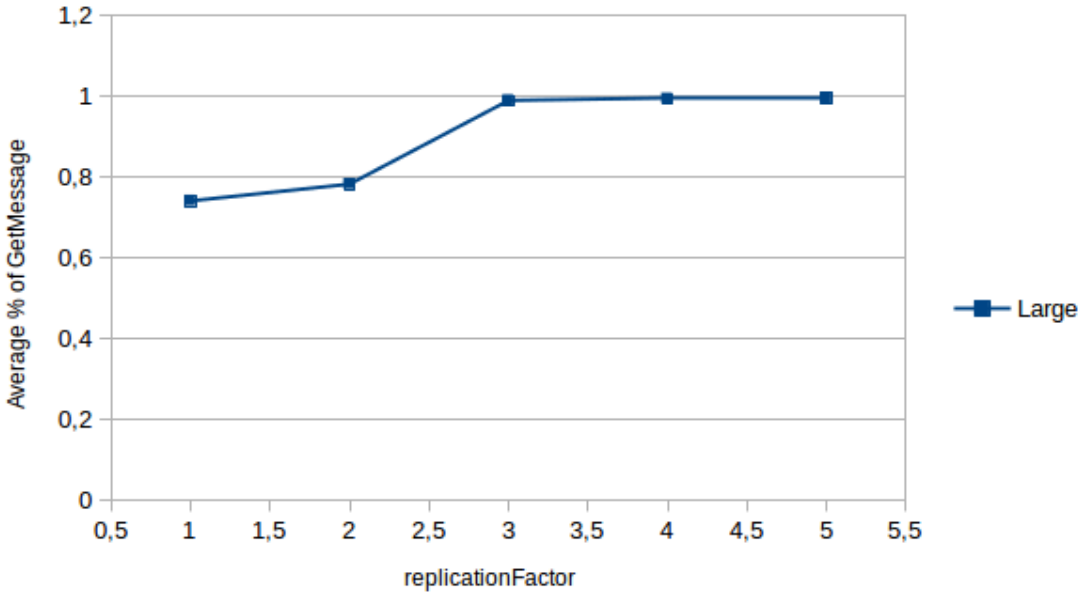

Figure 8: Get success ratio in relation of `replicationFactor`.

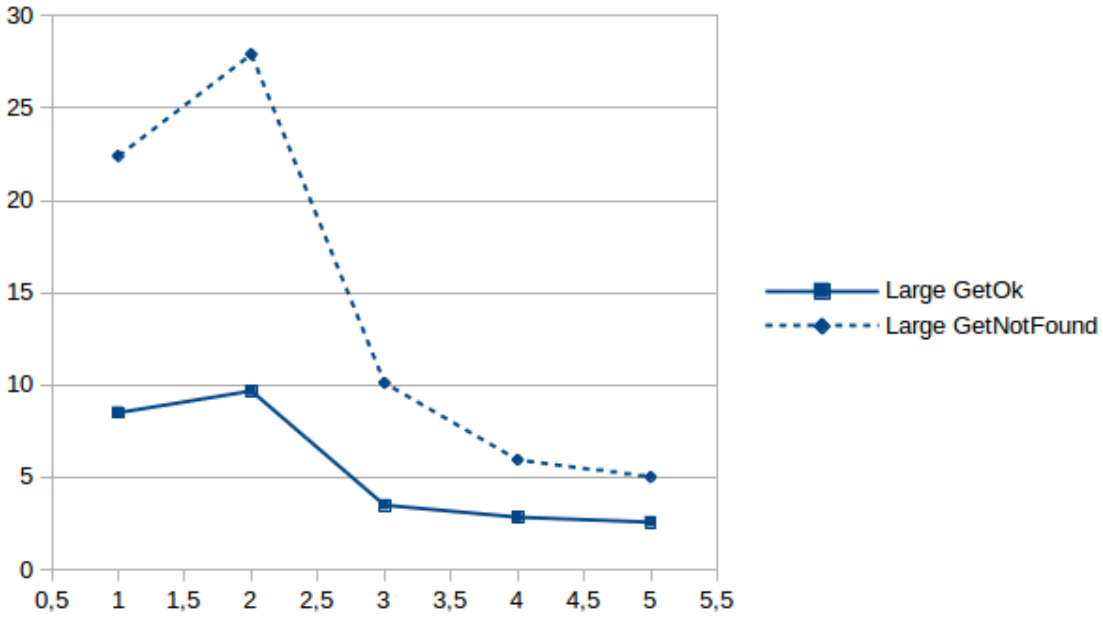

Figure 9: Average number of hops depending on `replicationFactor`.

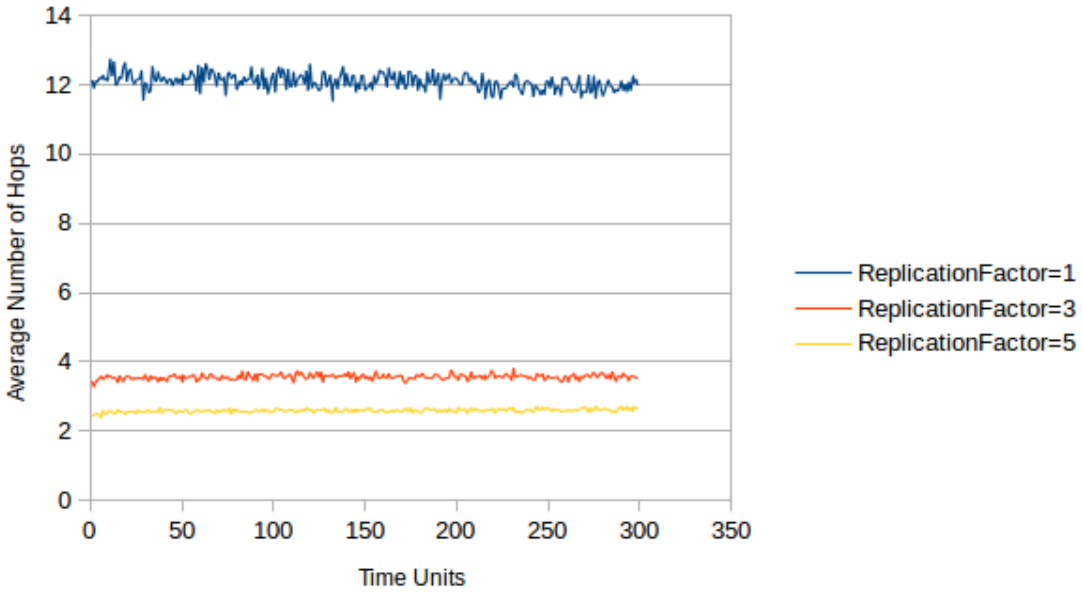

Figure 10: Swapping effect over time for different values of `replicationFactor` .

References

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[EGG07] Nathan S Evans, Chris GauthierDickey, and Christian Grothoff. Routing in the dark: Pitch black. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 305–314. IEEE, 2007.

[MBR+] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, et al. Symphony: Distributed hashing in a small world.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.