



**PEPITO**  
**IST-2001-33234**  
**PEer-to-Peer Implementation and TheOry**

**Deliverable no: D4.8**

**Final report on**

**Distributed Garbage Collection for the DSS**

REPORT VERSION: first    TOTAL NO OF PAGES: 17    CLASSIFICATION: Public

REPORT PREPARATION DATE: 2005.02.22

DELIVERABLE NO: D4.8    DUE DATE: Month 38    DELIVERY DATE: Month 38

ESTIMATED CUMULATIVE NO OF PERSON MONTHS REQUIRED: 1

PROJECT START DATE: 2002.01.01    PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: INRIA

CONTRIBUTING PARTNERS: INRIA, SICS

PROJECT COORDINATOR: Swedish Institute of Computer Science, Kista

PROJECT PARTNERS: EPFL Lausanne, INRIA, KTH Stockholm, SICS Kista, UCAM Cambridge, UCL Louvain



**Project funded by the European Community under the  
'Information Society Technologies' Programme (1998–  
2002)**

Project Number: IST-2001-33234  
Project Acronym: PEPITO  
Title: PEer-to-Peer Implementation and TheOry  
Deliverable No: D4.8  
Final report on  
Distributed Garbage Collection for the DSS  
Due date: project month 38  
Delivery date: 2005-02-22

Responsible Partner: INRIA  
Participating Partners: INRIA, SICS

2nd March 2005

Prepared by Fabrice Le Fessant, with input from Erik Klinskog.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Outline . . . . .	3
<b>2</b>	<b>Related Work</b>	<b>4</b>
2.1	Distributed Garbage Collection . . . . .	4
2.1.1	Partial Garbage Collection . . . . .	4
2.1.2	Complete Garbage Collection . . . . .	5
2.2	Structured Overlays . . . . .	6
<b>3</b>	<b>The Algorithm</b>	<b>6</b>
3.1	The Time-Lease Algorithm . . . . .	6
3.2	The Kademia Overlay . . . . .	7
3.3	The Mixed Algorithm . . . . .	8
<b>4</b>	<b>Experimentation</b>	<b>8</b>
4.1	The Simulator . . . . .	9
4.1.1	The Overlay . . . . .	9
4.1.2	The Workload . . . . .	11
4.1.3	The Garbage Collection Algorithm . . . . .	12
4.2	The Experiments . . . . .	12
4.2.1	Uniform Distribution . . . . .	13
4.2.2	Power-law Distribution . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>15</b>

## 1 Introduction

This report describes a new distributed garbage collection algorithm, especially designed for peer-to-peer systems. Indeed, contrary to other garbage collection algorithms, our algorithm uses the topology of a structured overlay over a peer-to-peer system to decrease the cost of the global garbage collection for every host. As a consequence, it scales better than other garbage collection algorithms with the number of clients, objects and references.

Traditionally, distributed garbage collection algorithms are designed for small-scale systems, containing not more than dozens or hundreds of clients. In such systems, a centralized approach might be possible, even for complete garbage collection. However, in peer-to-peer systems containing maybe millions of clients, such solutions cannot be used. Moreover, reference listing, a scalable and fault-tolerant solution, can perform very badly when the number of references increase in the system. As a consequence, a time-lease mechanism seems to be the only viable approach for such systems. In a time-lease mechanism, a client containing a reference to an object just needs to *book* the object for a given time – the *lease* – corresponding to how long it expects to keep that reference. If necessary, it can ask from time to time for an extension of the lease. This approach has several advantages: first, the information kept by reference and object is very light (a timer for the object, a timer and a pointer for all remote references); second, the system is tolerant to failures, as the failure of one client cannot prevent the eventual collection of an object; finally, the algorithm is complete with regards to cycles, since leases are only extended when local roots are available.

However, time-lease garbage collection algorithms might not scale very well to realistic systems. Indeed, in such systems, following studies on file-sharing peer-to-peer systems, the distribution of objects and references obeys to a power-law ([?, ?], see Figure 1). As a consequence, a client hosting a very popular object might receive thousands of lease requests, either for creation or extension, and might be unable to bear such a load.

### 1.1 Overview

In this report, we present a new time-lease algorithm, that uses routing over a structured overlay to decrease the load of popular clients. Indeed, in a structured overlay, such as the DKS [?], Chord [?], Pastry [?] or Kademlia [?], links between peers are organized so that any peer is reachable from any other peer in a number of hops logarithmic to the total number of clients. Moreover, all the routes to a particular peer form in its neighborhood a tree: with this algorithm, we propose to move the load of a peer to its neighbors in the tree, i.e. the peers which are close to it in most routes towards it.

We implemented a simulated version of this algorithm. We use an overlay close to Kademlia to route messages, and we ran our algorithm on different sets of clients (from 1,000 to 10,000 peers) hosting objects and references, whose distribution is either uniform or in power-law. The evaluation showed that the load balancing is effective.

### 1.2 Outline

The structure of this report is the following: Section 2 presents some related work, Section 3 introduces our algorithm while Section 4 displays the results of our experimentations. Finally, we conclude in Section 5.

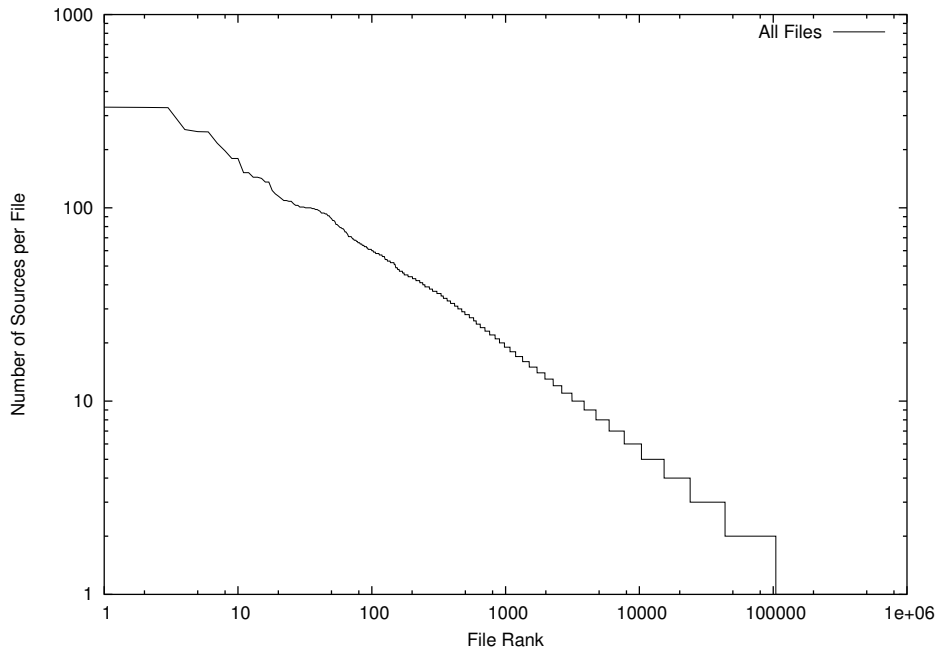


Figure 1: The popularity of files in a trace of the Edonkey network. The trace was collected over 3 days in November 2003, and described in [?]

## 2 Related Work

In this section, we briefly introduce the research done in distributed garbage collection, and independently in structured overlays, which are also often described as distributed hash tables (DHT).

### 2.1 Distributed Garbage Collection

A lot of research has been done in Distributed Garbage Collection over the last twenty years. We classified the proposed algorithms as *Partial* or *Complete* depending on their ability to collect distributed cycles of garbage.

#### 2.1.1 Partial Garbage Collection

Distributed Reference Counting [?] and its Weighted extension are the direct adaptation of reference counting to a distributed environment. For every shared object, a counter indicates how many remote references exist. When this counter drops to zero, and there are no more local references, the object is collected. The Weighted extension allows to increase the counter by more than one for a remote reference, so that the client hosting this reference can create other remote references without sending new messages to increase the object counter. The well-known drawbacks of this approach are the cost in messages (for most reference creations and deletions, messages must be sent to update the object counter) and non-tolerance to failures (if a client fails, the object will never be collected).

Reference Listing [?] solves some of the problems of Reference Counting by keeping the list of all remote references for every shared object. As a consequence, all references from a client that has

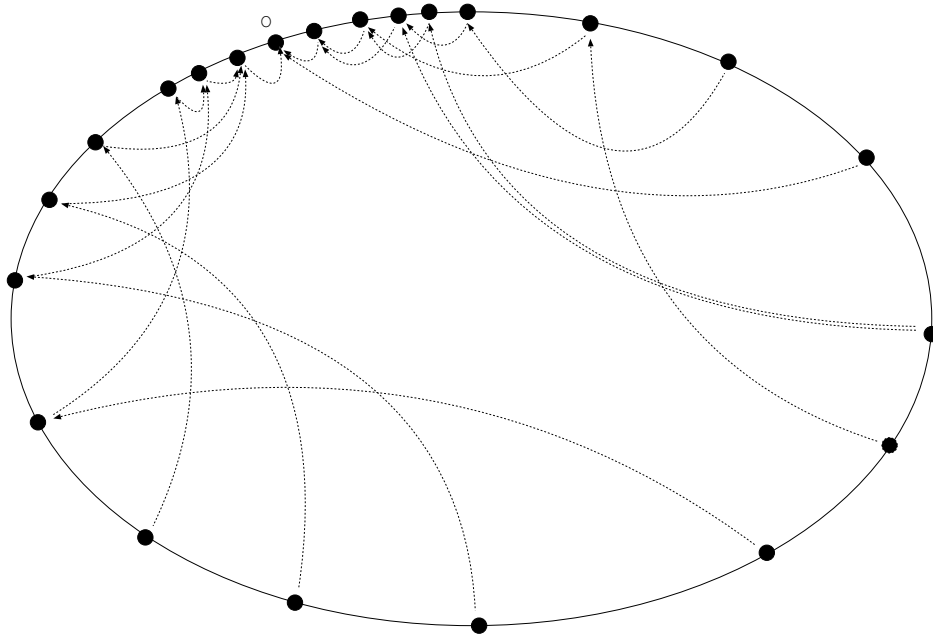


Figure 2: The routes to  $O$ : as messages are sent to  $O$ , they often pass through  $O$ 's neighbors in the logical ring of the Overlay.

crashed can be safely removed, making this mechanism fault-tolerant. Indirect Reference Counting or Listing also improves the previous schemes by adding a reference counter or list for every remote reference, so that creating new remote references does not require to send messages to the targeted object.

### 2.1.2 Complete Garbage Collection

Both these techniques still suffer from the fact that they cannot collect distributed cycles of garbage, which are yet likely to appear in such systems. As a consequence, several approaches have been proposed to extend them with the detection of cycles: [?, ?] divide the set of clients in small groups, depending on some heuristics. These groups are then traced to collect the cycles included in them. [?, ?] migrate objects likely to be included in cycles on a single site, where a trace is then performed. [?, ?, ?, ?] proposed to use back-tracing to detect cycles, i.e. to trace references backwards from suspected objects to their roots, if they exist.

Another approach is to collect all distributed garbage together, making no difference between cyclic and acyclic garbage. The DMOS garbage collector [?] groups the objects in *cars*, forming a *train*, and moves the objects from older cars to newer cars in the train, while old cars containing only garbage are collected. [?] adapts Hughes' algorithm for parallel systems to asynchronous systems, propagating timestamps on remote references from the roots so that garbage can be detected by their obsolete timestamps.

Most of these algorithms are not well-adapted for peer-to-peer systems, since they either require bounds on the number of clients or the number of references. Only one very simple approach seems usable in such environments: time-lease algorithms. Time-lease algorithms rely on the booking of

resources for a given time (the *lease*). We will describe such an algorithm in Section 3, and then extend it with overlay-based load balancing.

## 2.2 Structured Overlays

Basically, peer-to-peer systems can be classified in two categories: unstructured networks, among which famous file-sharing networks such as Gnutella [?], Fasttrack [?], etc. . . ), and structured networks, built upon routing overlays or distributed hash tables. We are here only interested in the second approach: in such systems, each peer is associated with an identifier – often, a semi-randomly generated key of more than hundred bits –, which is used to structure the network. Such systems mainly differ in the way links are created between the peers, and most of them offer routing of messages in a logarithmic number of hops between any two peers of the system, using these links.

CAN [?], for example, associate a square in a multi-dimensional space with each peer, and route messages using the peers on the line between two points in the space. Chord [?] allocate the peers in a ring, and every peer in position  $id$  has a link to the first peer after position  $id + 1/2$ ,  $id + 1/4$ ,  $id + 1/8$  and so on. Pastry [?] builds a routing table such that, if two peer identifiers differ starting at some digit, the routing table of both of them contain some peer whose identifier differ at least one digit later. Finally, Kademia [?], which we use in our experimentation, will be described in Section 3.

## 3 The Algorithm

The distributed garbage collection algorithm that we have designed is divided in two parts: a normal time-lease algorithm, and a structured overlay to route the messages. As an example, we took the Kademia overlay, but any other structured overlay would give similar results. In the next paragraphs, we first present a basic time-lease algorithm, the Kademia overlay, and then our mix of them.

### 3.1 The Time-Lease Algorithm

For the sake of simplicity, we describe the behavior of the algorithm for only one object  $O_0$ , located on client  $C_0$ . A remote reference on  $O_0$  is noted  $R_n$  with  $n > 0$  if it is located on client  $C_n$ .

With the object  $O_0$  is associated a value  $Lease(O_0)$ , containing the time at which all remote references will become obsolete. If  $T_0$ , the local time on client  $C_0$ , becomes greater than  $Lease(O_0)$ , and  $O_0$  is not locally reachable from roots,  $O_0$  is reclaimed by the local garbage collector. Thus, the task of the time-lease algorithm is to keep  $Lease(O_0)$  up-to-date if some remote references are still useful.

For that, with each reference  $R_n$ , we associate a value  $Lease(R_n)$ , which is the approximation of  $Lease(O_0)$  on  $C_n$ . If  $T_{transfer}$  is the maximal delay of transmission for a message,  $C_n$  must send a request to increase the lease to  $C_0$  before  $Lease(R_n) - T_{transfer}$  in local time. The message  $LeaseRequest(O_0, T_n, T_{update})$  contains the local time  $T_n$  on  $C_n$ , and the requested lease extension  $T_{update}$ .

When the  $LeaseRequest(O_0, T_n, T_{update})$  message is received by  $C_0$ ,  $Lease(O_0)$  is set to the maximum of its former value and  $T_0 + T_{update}$ . Finally,  $LeaseReply(O_0, T_n, lease(O_0) - T_0)$  is sent back to  $C_n$ . On receipt of a message  $LeaseReply(O_0, T'_n, T_{updated})$ ,  $C_n$  sets  $Lease(R_n)$  to  $T'_n + T_{updated}$ .

Of course, this algorithm only works if  $T_{update}$  is much greater than  $T_{transfer}$ , and  $T_{transfer}$  a safe upper-bound on transmission delays and the clock drifts, taking also retransmissions into account. It also guaranties that the  $Lease(R_n)$  is a *safe* approximation of  $Lease(O_0)$ , i.e. that if the clocks

of  $C_0$  and  $C_n$  are not synchronized, the object cannot be reclaimed on  $C_0$  before the expiration of  $Lease(R_n)$ .

The main problem of this algorithm in our setting is the fact that  $C_0$  receives `LeaseRequest` messages from all remote references holders every  $T_{update} - T_{transfer}$  time. For a popular object, the load can become unbearable for a normal client. Moreover, this load is not related to the actual use of the object, but to the existence of many references, even if they are not used. Our goal is thus to try to decrease this load on clients hosting popular objects.

### 3.2 The Kademlia Overlay

Kademlia [?] is the only structured overlay currently implemented and used in the *real world*: it is used as a Distributed Hash Table by two file-sharing networks, called Overnet [?] and Kad. In this section, we describe how routing is performed in the Kademlia overlay.

Each client  $C_n$  is associated with a semi-randomly generated identifier or key,  $Key_n$  with  $N$  bits, generally between 128 and 160 bits to avoid collisions. Each client also maintains a *routing table* that is used to route messages in the overlay to any peer identified by its key.

The routing table of a peer is an array, containing at every index a set of other peers, called a *bucket*, of maximal size  $k$ . We note  $Buckets_n$  the routing table of peer  $C_n$  in the following. Every time  $C_n$  learns about the existence in the network of a new peer  $C_p$ , it tries to insert it in its routing table. For that, it compares  $Key_n$  and  $Key_p$  by an exclusive arithmetical or, to obtain the *Xor* distance (or metric) between the two peers.  $Buckets_n[i]$  is built to contain the  $k$  known peers, closest to  $Key_n$  with that metric, and such that they share the  $i$  first bits of  $Key_n$  but not the  $(i + 1)^{th}$  bit. We call  $bucketof_n(Key_p)$  the function that returns the bucket on  $C_n$  that should contain the peer associated with key  $Key_p$ .

To route a message to peer with identifier  $Key_p$ , a peer  $C_n$  maintains a set  $Candidates(Key_p)$ , containing potential peers knowing  $C_p$ . We define a function  $neighbors_n(Key_p)$  as the set of peers, built by adding in order  $Buckets_n[i]$ ,  $Buckets_n[i + 1]$ , ...,  $Buckets_n[N]$ , where  $i$  is the value  $bucketof_n(Key_p)$ , and then  $Buckets_n[i - 1]$ , ...,  $Buckets_n[0]$ , stopping as soon as  $k$  peers are in the set.  $C_n$  first initializes  $Candidates(Key_p)$  with  $neighbors_n(Key_p)$ . Then, it iterates the following process: if a peer  $C_p$  with key  $Key_p$  is contained in  $Candidates(Key_p)$ , send the message to that peer; otherwise, pick the peer  $C_m$  with the closest key  $Key_m$  to  $Key_p$  for the Xor distance, and send it a message `KademliaRequest`( $Key_p$ ). The peer  $C_m$  replies with a message `KademliaReply`( $Key_p, S$ ), where  $S$  is the set returned by  $neighbors_m(Key_p)$ . On receipt,  $C_n$  adds  $S$  to  $Candidates(Key_p)$  and restart the same process with another peer in the set.

The idea behind this process is that all the peers in the bucket  $Buckets_n[bucketof_n(Key_p)]$  have at least one more bit in common with  $Key_p$  than  $Key_n$  has. Then, the peers returned in `KademliaReply` messages by these peers – if the routing tables are correctly built – will also share a longer prefix with  $Key_p$ , until  $C_p$  is finally returned.

This routing algorithm has several advantages: first, as routing is performed by iteration from the same peer, this peer learns about the existence of many other peers during this process, and can update its routing table thanks to this. Second, as every bucket contains many (around  $k$ ) peers, the overlay easily tolerates failures in the network. Finally, Kademlia has been implemented and used on a network of about one million users, which, from an user point of view, seemed to work perfectly.

### 3.3 The Mixed Algorithm

We can now introduce our new algorithm. The basic idea of the algorithm is to use the property displayed in Figure 2: when routing a message to the object  $O_0$ , immediate neighbors of  $C_0$  in the logical ring of the Overlay will often be contacted; thus, we can try to move partially the load on  $C_0$  to these peers.

The algorithm for a Kademlia overlay works as follows: when a  $Lease(R_n)$  needs to be updated on a peer  $C_n$ , for the object  $O_0$  on peer  $C_0$  with identifier  $Key_0$ ,  $C_n$  creates a set  $Candidates(Key_0)$ , initialized with  $neighbors_n(Key_0)$ . Then, it iterates the following step: pick the peer  $C_p$  in the set  $Candidates(Key_0)$  with the closest identifier  $Key_p$  to  $Key_0$ , and send to  $C_p$  a message  $LeaseRequest(Key_0, O_0, T_n)$ .

On receipt of such a message, there are two possible cases:

- ▷  $C_p$  is  $C_0$  and hosts the object  $O_0$ : in this case, as in the basic time-lease algorithm,  $Lease(O_0)$  is updated so that  $Lease(O_0) \geq T_0 + T_{update}$ , and a  $LeaseReply(Key_0, O_0, T_n, Lease(O_0) - T_0)$  message is sent back.
- ▷  $C_p$  hosts a reference  $R_p$  to  $O_0$ : in this case, the result depends on the current lease on  $R_p$ :
  - $Lease(R_p) - T_p \geq T_{update}$ :  $C_p$  knows that the lease on  $O_0$  does not need to be updated, and immediately replies with a message  $LeaseReply(Key_0, O_0, T_n, Lease(R_p) - T_p)$ .
  - $Lease(R_p) - T_p < T_{update}$  or  $C_p$  has no reference to  $O_0$ :  $C_p$  has not enough information on the lease on  $O_0$ , it just replies with routing information in the Overlay, contained in a  $KademliaReply(Key_0, O_0, T_p, S)$ , where  $S$  is the set returned by  $neighbors_p(Key_0)$ , and  $T_p$  the local time on  $C_p$ .

Depending on the reply of  $C_p$ ,  $C_n$  has two possibilities:

- ▷ If it receives a message  $KademliaReply(Key_0, O_0, S)$ , it just adds  $S$  to its  $Candidates(Key_0)$  set, and the pair  $(C_p, T_p)$  to a set  $Updates(Key_0, O_0)$  before going to the next iteration.
- ▷ If it receives a message  $LeaseReply(Key_0, O_0, T'_n, T_{updated})$ , it sets its own value of  $Lease(R_n)$  to  $T'_n + T_{updated}$ , and then, for every pair  $(C_p, T_p)$  of the  $Updates(Key_0, O_0)$  set, it sends a message  $LeaseReply(Key_0, O_0, T_p, T_{updated})$  to  $C_p$ .  $C_p$  uses this message to create a reference  $R_p$  to  $O_0$  if it does not exist, and then set  $Lease(R_p)$  to  $T_p + T_{updated}$ , where  $T_p$  is the time specified in the message (not its local time, but the local time when the  $KademliaReply$  was sent).

The main advantage of this algorithm is that, as soon as one of the peers on the route to  $C_0$  owns a reference to  $O_0$ , its lease will be used to decide whether or not the message should continue or not to  $C_0$ . Moreover, the update of its own lease is performed by the last peer requiring a lease extension, and not by  $C_0$ .

## 4 Experimentation

To evaluate the behavior of our mixed algorithm, we implemented a simulator in Objective-Caml. We are the simulator on a PC with 1 gigabyte of memory, which limited us to a maximum of 10,000 peers in the simulations.



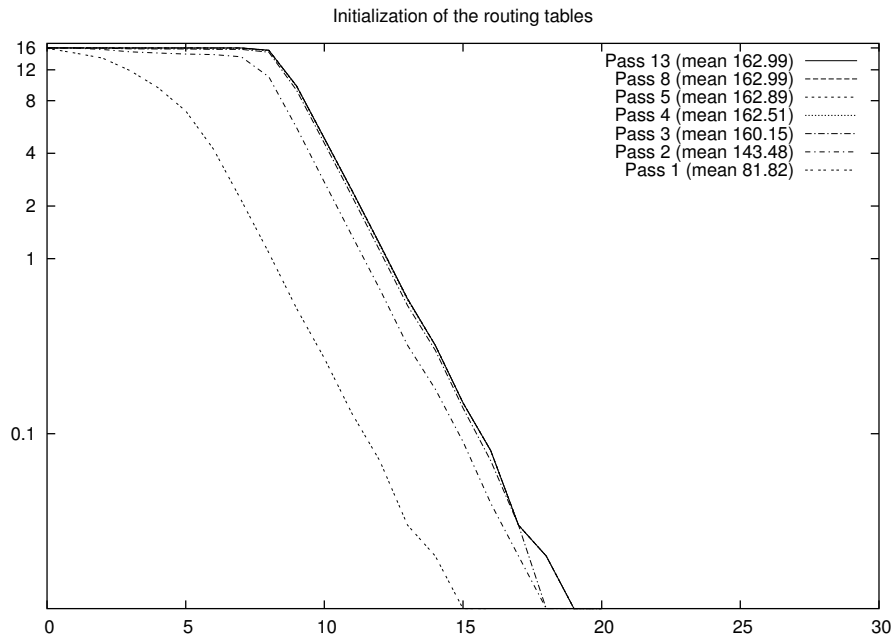


Figure 3: Initialization of the routing tables. With 10,000 peers, the initialization requires 13 passes before stabilization. Each peer has 163 other peers in its routing table.

table, using the key-identifier of the client and its number. The client is always added to the routing table, whatever the number of clients already in the routing table and in each bucket is. The `clean` function is responsible to clean the routing table after many insertions, to have a limited number of clients per bucket (16 in our experiments). Inside a bucket, the clients are stored in 16 sub-buckets, corresponding to the last 4 bits returned by `common_bits`. When cleaning a bucket, we always keep the clients that are in the smallest sub-buckets, i.e. the one with the identifiers closer to the one of the client. Finally, the `find_clients` function returns the clients closer to a given key in a routing table. The function returns in fact a list of sub-buckets, taken first in the bucket of the key to find (with the sub-bucket of the key first), then in the following buckets, and finally in the previous buckets, so that at least 20 non-empty sub-buckets are returned.

```

type reference_type = {
  ref_num : int;
  ref_location : int;
  mutable ref_expire : int;
  mutable ref_lease : int;
}

type client_type = {
  client_num : int;
  client_routing : ROUTING.t;
  mutable client_refs : reference_type Intmap.t;

```

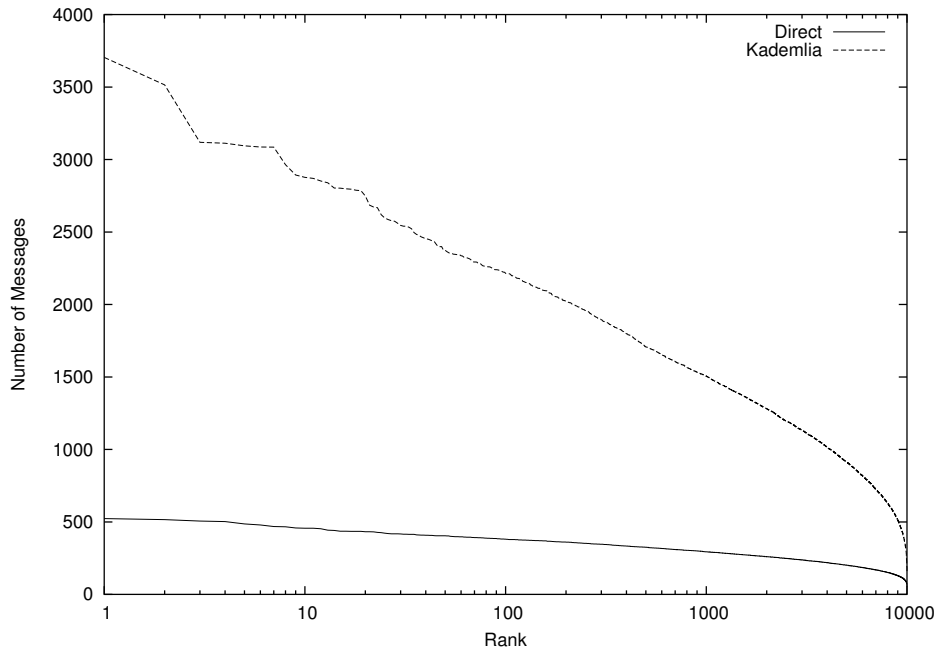


Figure 4: Number of messages with uniform distribution of object popularity

}

```
val clients: client_type array
```

In the simulator, the clients are stored in a table `clients`, and only contain a routing table and a set of references. Before starting a simulation, we need to fill the routing tables. This is a long process, so we do it only once for each set of clients, and save it on disk for other simulations. The process is the iteration of the following step, until the number of clients in the routing tables does not change anymore:

1. For every client:
  - (a) Insert 20 clients found randomly into the routing table.
  - (b) Perform a search of our identifier in the overlay. During this process, query at most 100 peers. Insert all the peers found during the search inside our routing table, and insert us inside their routing table.
2. Clean all the routing tables

The Figure 3 shows the results of the iterations, with the mean of the number of peers per bucket after cleaning, and indicates how many peers a client stores in its routing table after each pass.

#### 4.1.2 The Workload

A workload is a set of events, indicating when a reference is created on some peer and how long it will be used by that peer. When a reference is created for an object which is unknown or already

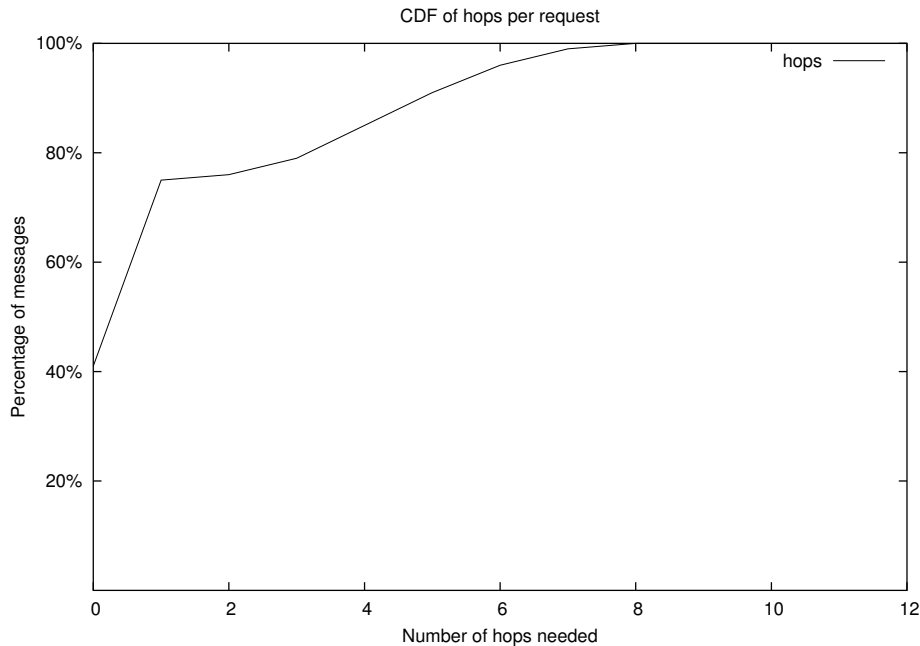


Figure 5: Number of hops with uniform distribution of object popularity

dead, it is interpreted as the creation of that object. Further references on other peers with the same number are interpreted as remote references to that object.

We generate two kinds of workload: uniform workloads and power-law workloads. The first kind means that the distribution of popularities of objects is uniform: every object has the same probability of being referenced as other objects. The second kind means that the distribution of popularities is in power-law, as shown in Figure 1. To create such a distribution, we generate a table such that, if  $n$  is the total number of peers,  $n/2$  peers appear only once in the table,  $n/4$  appear twice in the table,  $n/8$  appear 4 times in the table, etc... For each event, we then randomly pick one peer from the table.

#### 4.1.3 The Garbage Collection Algorithm

The algorithms we implemented in our simulator are the same ones as in Section 3.1 and 3.3. To improve the behavior of our mixed algorithm, we added only a small improvement: when a request is received for a lease of time  $T_{update}$  by the final object host, the lease is not updated by  $T_{update}$  but by  $F_{time-factor} \times T_{update}$ , with  $F_{time-factor}$  in the range [1..4].

## 4.2 The Experiments

Among the many experiments we tried with different workloads, we present the two most interesting ones here: 500,000 references creation in a network of 10,000 peers, either with a uniform or power-law distribution.

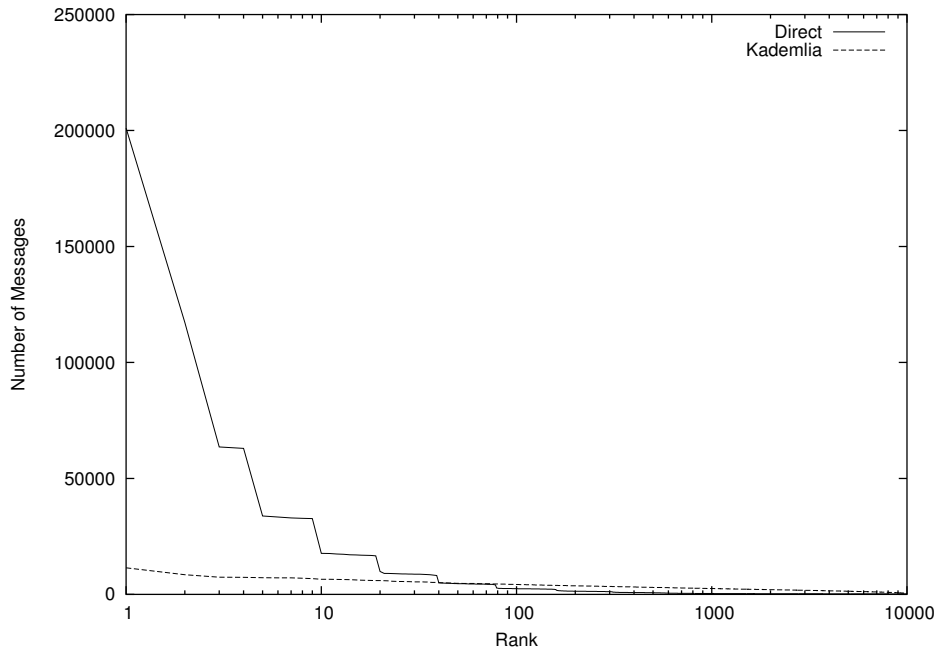


Figure 6: Number of messages with power-law distribution of object popularity

#### 4.2.1 Uniform Distribution

We first compare the two algorithms on a uniform distribution of popularities.

Number of clients	10,000	
Number of objects	30,000	
Period	500 s	
Number of events	500,000	
	Direct routing	Overlay routing
Mean	207	972
Minimum	53	146
Maximum	523	3,704

The Figure 4 shows the number of messages received by each client. As expected, the cost of the mixed algorithm is higher than the basic algorithm. Indeed, for each reference, each message in the overlay must be routed through different peers, while it is sent directly in the basic algorithm. Figure 5 shows the number of hops per message. However, the difference between the two algorithms is quite small, a factor in a range of 3 to 10 in the worst case. In the uniform workload, the difference between the busiest peer and the least loaded peer is less than 3 for the basic algorithm, and less than 4 for the mixed algorithm.

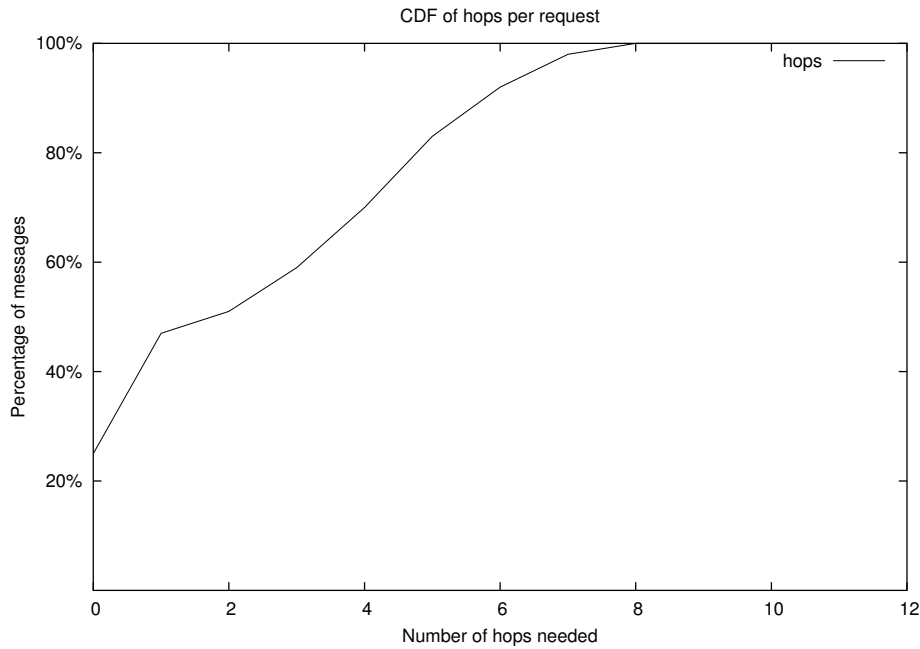


Figure 7: Number of hops with power-law distribution of object popularity

#### 4.2.2 Power-law Distribution

The power-law distribution is expected to be closer to real systems, i.e. to peer-to-peer systems already observed.

Number of clients	10,000	
Number of objects	5,000	
Period	5,000 s	
Number of events	500,000	
	Direct routing	Overlay routing
Mean	400	1,518
Minimum	99	198
Maximum	201,033	11,466

As shown in Figure 6, the basic algorithm performs very badly in term of load balancing. Indeed, one peer receives around 200,000 messages whereas the least loaded one receives only one hundred messages. On the other hand, our mixed algorithm performs pretty well, with a maximal load of fewer than 12,000 messages, i.e. around twenty times smaller than the basic algorithm.

In Figure 8, we display the different curves using different time-factors. In the previous experiment, we used a time-factor of 2. for both algorithms. Here, we can observe that the number of messages received just follows a linear dependency towards the time-factor.

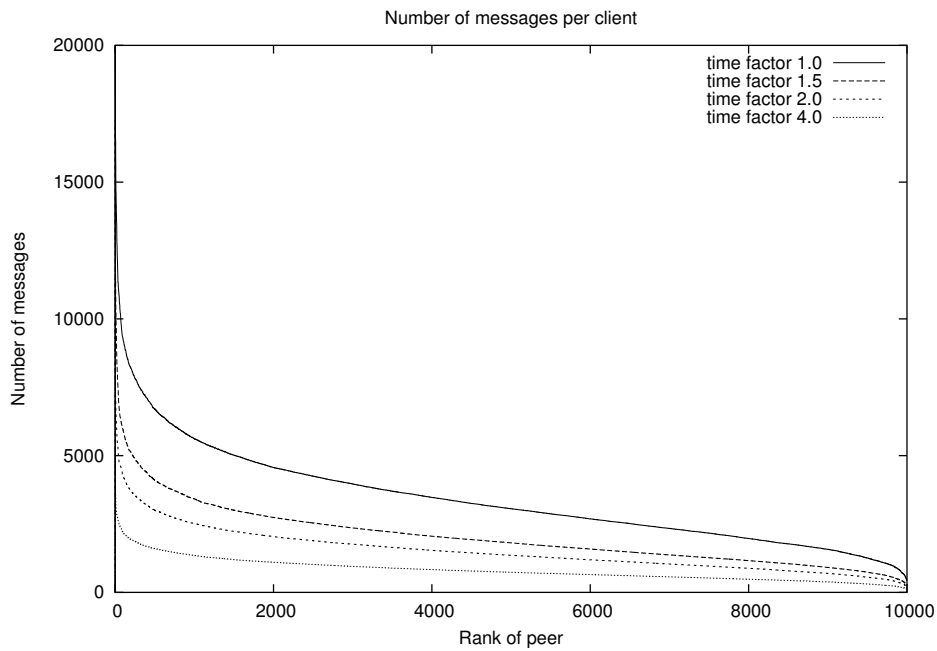


Figure 8: Number of Messages depending on the Time Factor

## 5 Conclusion

In this report, we presented our work on the design of a time-lease distributed garbage collection algorithm for peer-to-peer systems. The main problem to solve was the load balancing of messages in those workloads, in particular because of the power-law distribution of popularities of objects, observed in real systems. We implemented our algorithm together with the basic time-lease algorithm in a simulator, and experimental results show that our new algorithm performs much better than the former algorithm in terms of load balancing, while its average cost is not too high.