Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

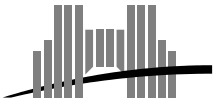
Efficiency of Tree-Structured Peer-to-peer Service Discovery Systems

Eddy Caron ,
Frédéric Desprez ,
Cédric Tedeschi

June 2008

Research Report N° 2008-18

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



Efficiency of Tree-Structured Peer-to-peer Service Discovery Systems

Eddy Caron , Frédéric Desprez , Cédric Tedeschi

June 2008

Abstract

The efficiency of service discovery is a crucial point in the development of fully decentralized middlewares intended to manage large scale computational grids. The work conducted on this issue led to the design of many peer-to-peer fashioned approaches. More specifically, the need for flexibility and complexity in the service discovery has seen the emergence of a new kind of overlays, based on ***tries***, also known as ***lexicographic*** trees.

Although these overlays are efficient and well designed, they require a costly maintenance and do not accurately take into account the heterogeneity of nodes and the changing popularity of the services requested by users.

In this paper, we focus on reducing the cost of the maintenance of a particular architecture, based on a dynamic prefix tree, while enhancing it with some load balancing techniques that dynamically adapt the load of the nodes in order to maximize the throughput of the system. The algorithms developed couple a self-organizing prefix tree overlay with load balancing techniques inspired by similar previous works undertaken for distributed hash tables.

After some simulation results showing how our load balancing heuristics perform in such an overlay and compare to other heuristics, we provide a fair comparison of this architecture and similar overlays recently proposed.

Keywords: Service discovery, computational grids, peer-to-peer, prefix trees, mapping, load balancing

Résumé

L'efficacité de la découverte de services est un point crucial du développement d'intergiciels de grille totalement décentraliés. Les travaux ayant pour but la résolution de ce problème ont généré un certain nombre d'approches pair-à-pair. le besoin de flexibilité et d'expressivité a donné lieu au développement d'architecture s'appuyant sur des arbres de préfixes (ou arbres lexicographiques).
Ces overlays souffrent d'une maintenance couteuse et ne prennent pas en compte la nature hététogène de la plate-forme physique sous-jacente et la popularité différente et changeante de chaque ressource enregistrée. Dans ce rapport, nous nous focalisons sur la rduction du cout de maintenance d'une telle architecture, basée sur un arbre de préfixes dynamique, tout en lui donnant la possibilité de s'adapter à l'hétérogénéité précitée par l'enrichissant de mécanismes de répartition de la charge qui adaptent dynamiquement la charge des noeuds dans le but de maximiser le débit su service. Notre approche couple des travaux de répartition de la charge dans les DHTs avec un overlay en arbre de préfixes auto-organisant.
Après des résultats de simulation mettant en évidence l'éfficacité de notre heuristique, nous comparons notre approche avec les travaux s'appuyant sur des structures distribuées similaires.

Mots-clés: Découverte de services, grilles de calcul, pair-à-pair, arbres de préfixes, répartition de la charge, plongement

1 Introduction

Grids connecting geographically distributed computing resources have become a low cost alternative to supercomputers. For a few years now, the convergence of the grid computing and peer-to-peer communities has produced numerous papers attempting to design new approaches to make grid middleware able to work over fully decentralized platforms [9]. One crucial point in the design of such systems is the efficiency of the service discovery. More specifically, the need for flexibility and complexity in the service discovery process led to the emergence of a new kind of overlays, based on ***tries*** *a.k.a.,* lexicographic trees. These architectures usually support range queries, automatic completion of partial search strings and are easy to extend to multi-attribute queries.

Although these systems provide flexible meanings of search and are designed to be efficient over dynamic large scale platforms, they require a costly maintenance and suffer from a lack of adaptability as heterogeneity increases in the platform and hot spots appear and disappear more and more frequently.

In this paper, we focus on the DLPT (Distributed Lexicographic Placement Table), a more particular architecture recently developed [5]. This approach is a two layer architecture. The upper layer is a prefix tree maintaining the information about services declared. This tree is mapped onto the lower layer *i.e.,* the physical networks by the intermediary of a distributed hash table. The first contribution of this paper is the avoidance of the DHT: we developed a self-contained tree overlay able to both maintain the tree and map it onto the network. The load balancing issue has been mostly ignore by previous work on this architecture [5, 6]. The second contribution is the design of a new load balancing heuristic based on the local maximization of the throughput. This heuristic is inspired by existing approach for the load balancing within distributed hash tables. We then adapt it to our case and compared its performance to the closest existing heuristic we are aware of. Finally, the paper ends by an attempt to give the means for a fair comparison of our approach and similar works in terms of functionalities and performance. These works include both the load balancing techniques used in distributed hash tables and other close trie-based architecture.

2 Preliminaries

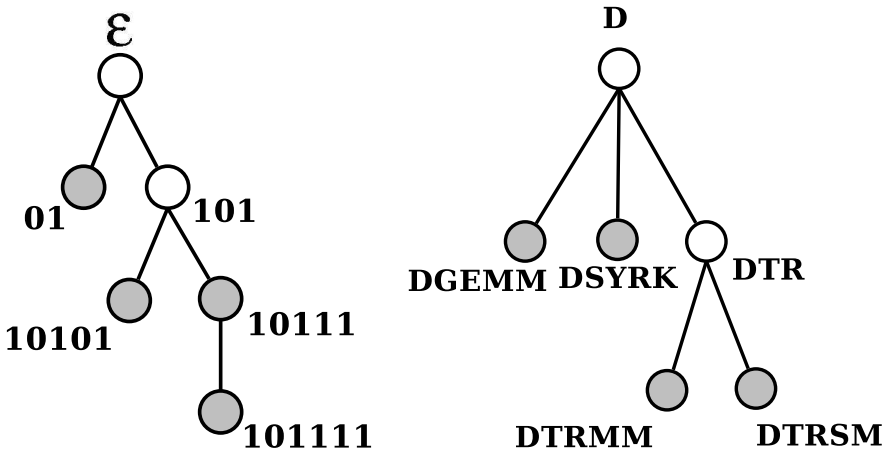
System Model. A P2P network consists of a set of asynchronous ***physical*** nodes with distinct IDs. In the following, we use the term ***peer*** to refer to this kind of nodes. The peers communicate by exchanging messages. Any peer P_1 can communicate with another peer P_2 provided P_1 knows the ID of P_2. Each peer maintains one or more ***logical*** nodes of the distributed ***logical*** tree. In the following, we use the term ***node*** to refer to the nodes of the tree.

Greatest Common Prefix Tree. Let A be a finite set of digits *e.g.,* $A = \{0, 1\}$. A non empty identifier w over A is a finite sequence of digits $a_1, \ldots, a_i, \ldots, a_l$, $l > 0$. The ***concatenation*** of two ids u and v, denoted as uv, is the id $a_1, \ldots, a_i, \ldots, a_k, b_1, \ldots, b_j, \ldots, b_l$ such that $u = a_1, \ldots, a_i, \ldots, a_k$ and $v = b_1, \ldots, b_j, \ldots, b_l$. Let ϵ be the ***empty id*** such that for every id w, $w\epsilon = \epsilon w = w$. The ***length*** of an id w, denoted as $|w|$, is the number of digits of w—$|\epsilon| = 0$. An id u is a ***prefix*** (respectively, ***proper prefix***) of a word v if there exists an id w such that $v = uw$ (resp., $v = uw$ and $u \neq v$). The ***Greatest Common Prefix***

(resp., ***Proper Greatest Common Prefix***) of a collection of ids $w_1, w_2, \ldots, w_i, \ldots$ ($i \geq 2$), denoted as $GCP(w_1, w_2, \ldots, w_i, \ldots)$ (resp. $PGCP(w_1, w_2, \ldots, w_i, \ldots)$), is the longest prefix u shared by all of them (resp., such that $\forall i \geq 1, u \neq w_i$).

Definition 1 (PGCP Tree)**.** *A **Proper Greatest Common Prefix Tree** is a labeled rooted tree such that the label of each node of the tree is the Proper Greatest Common Prefix of the labels of every pair of its children.*

Architecture. The DLPT architecture developed in [5] maintains a PGCP tree over a DHT. Figure 1(a) gives a sample of such a tree constructed with binary identifiers where 01, 10101, 10111 and 101111 are keys of resources made available by some servers. Note that the non-filled nodes 101 and ϵ have been created to maintain a tree satisfying Definition 1. More generally, it can be built with any kind of strings, for instance, with routines of the BLAS, as shown on Figure 1(b) (Note that in this last case, no hashing is required). When a discovery request sent by a client enters the tree, on a random node, the request moves upward until reaching a node whose subtree contains the requested node and then moves upward to this node. The DLPT system supports range queries and automatic completion of partial search strings.



a. Binary identifiers b. BLAS routines

Figure 1: Examples of PGCP tree

In its original design [5], the tree is mapped onto the network using a distributed hash table, thus requiring to maintain both layers. Such a mapping is illustrated by Figure 2, using the Chord mapping technique [18] *i.e.,* mapping a key on the peer with the lowest identifier higher than the key.

This leads to the first contribution of this paper: Avoiding the need for a DHT to both maintain the physical network and map the data keys onto it in such an architecture. In the following, we present a scheme that maintains a prefix tree over a P2P network without requiring a DHT.

Load balancing. The routing scheme proposed in [5] and the heterogeneity of both the capacity of peers and popularity of keys could lead to an unbalanced distribution of the load and thus create bottlenecks on different peers.

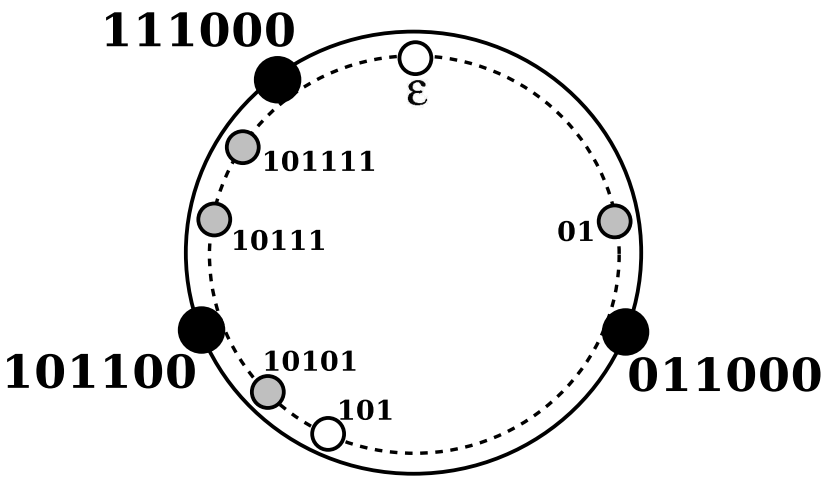


Figure 2: Ring

The second issue we face in this paper is how to inject some load balancing in the architecture proposed in [5]. In Section 3, we also give a novel load balancing heuristic taking these heterogeneities into account, based on a local maximization of the throughput.

3 Protocol

Consider a set of digits A and a circular identifier space $\mathcal{I}$ of all distinct ids i such that i is a finite sequence of digits of A. The protocol is made of two distinct parts.

One part maintains the physical network *i.e.,* builds a bidirectional ring over the peers as they join the network. Denote $\mathcal{P} \subseteq \mathcal{I}$ the set of peer identifiers currently in the ring. Peers are ordered in a bidirectional ring. Each peer $P \in \mathcal{P}$ has the knowledge of its immediate predecessor $pred_P$ and immediate successor $succ_P$ *i.e.,* peers whose identifier is the highest lower than P and the lowest higher than P, respectively. Let $P_{max} \in \mathcal{P}$ and $P_{min} \in \mathcal{P}$ be the two peers whose ids are the highest and lowest in the ring, respectively. Recall that this part of the protocol is mainly achieved by the nodes and does not require extra connections between peers.

The other part maintains a Greatest Common Prefix Tree over data keys as data are made available by some servers. Denote $\mathcal{N} \subseteq \mathcal{I}$ the set of node identifiers currently in the tree. The registration of a data k leads to the creation of some node if $\neg(\exists n \in \mathcal{N} \ : \ n = k)$. The protocol maps the tree onto the peers as it is growing. The mapping scheme ensures that the peer P chosen to run a given node n always satisfies the condition that P is the lowest peer id higher than n. Recall that if $\forall n \in \mathcal{N}$ such that $n > P_{max}$, the peer running n is P_{min}. Each node n maintains a father f_n, a set of children C_n and the set of all data δ_n associated with the key $k = n$. For the sake of readability, we use the key of a data to refer to both the key and the value associated with.

We assume two basic functions.

- PREFIXES(k) returns the set of ids properly prefixing k. For instance, PREFIXES(10101) returns $\{\epsilon, 1, 10, 101, 1010\}$

- GCP(k_1, k_2) returns the longest common prefix shared by k_1 and k_2. For instance, GCP$(101, 100) = 10$.

3.1 Peer insertion

When a peer P joins the system, the routing of the join request sent by P is handled by the nodes, until the routing process reaches a node run on a peer close to the final destination of P. Then the effective insertion is performed. This protocol is detailed by algorithms 1 and 2. The sought peer is the one with the highest identifier lower than P. To reach this peer, we first route the request to the node with the highest id lower than P.

In details, the path of a PeerJoin request is made of three steps. During a first step (lines 1.03-1.10), the request is marked 0, moves upward and eventually reaches a node that is either a prefix of P or the root, what changes the state of the request to 1. During a second step, the the request is marked 1 (lines 1.11 to 1.14) and moves downward until reaching the node t whose id is the highest lower than P. t then sends the request to the peer T on which it runs, and the request is delegated to the peer layer (Line 1.16). The final step consists in deciding whether P shall be a predecessor of T, or a predecessor of $succ_T$ (tested Line 2.03). Only these two cases are possible since $pred_T < P \leq succ_T$. $pred_T < P$ comes from the facts $pred_T \leq t$ and $t \leq P$. Now, by contradiction, let's assume $P > succ_T$. Since ν_T, the set of nodes run on P is not empty, $\exists n \in \nu_T$ such that $n > t$, which means that the first part of the algorithm did not give the proper t. A contradiction.

Once decided whether $succ_P$ is T or $succ_T$, it remains to effectively insert P and dispatch ν_{succ_P} among P and $succ_P$, according to their value, as detailed lines 2.05-2.10. The YourInformation message contains the information required for P to run *i.e.,* $(pred, succ, data)$. The UpdateSuccessor message informs $pred_Q$ that its successor has changed (from Q) to P. We do not detail the reception of these two messages due to their triviality.

```
1.01 Variables:      f_p, identifier of the father of p
                     C_p, finite set of children of p
1.02 upon receipt of <PeerJoin, P, s> do
1.03     if s = 0 then
1.04         if P ∉ Prefixes (p) then
1.05             if (f_p = ⊥) then
1.06                 send(<PeerJoin, P, 1>, p)
1.07             else
1.08                 send(<PeerJoin, P, 0>, f_p)
1.09         else
1.10             send(<PeerJoin, P, 1>, p)
1.11     else
1.12         q =Max ({q ∈ C_p : q ≤ P})
1.13         if (q ≠ ⊥) then
1.14             send(<PeerJoin, P, 1>, q)
1.15         else
1.16             send_to_host(<NewPredecessor, P>)
```

Algorithm 1: Peer insertion, on node p

3.2 Data Insertion

To declare the availability of a resource identified by k, a peer (or server) sends a DataInsertion request to a random node of the tree. The protocol routes the request to the node

```
2.01  Variables:    succ_Q, successor of Q
                    pred_Q, predecessor of Q
                    ν_Q, set of nodes running on Q
2.02  upon receipt of <NewPredecessor, P> do
2.03      if Q < P then
2.04          send(<NewPredecessor, P>, succ_Q)
2.05      else
2.06          ν_P = {n ∈ ν_p : n ≤ P}
2.07          ν_Q = {n ∈ ν_p : n > P}
2.08          send(<YourInformation, (Q_pred, Q, ν_P)>, P)
2.09          send(<UpdateSuccessor, P>, pred_Q)
2.10          pred_Q := P
```

Algorithm 2: Peer insertion, on peer Q

with the identifier closest to k. If $\nexists n \in \mathcal{N}$ with $n = k$, such a node is created, inserted in the tree and run on a peer. In any case, k is eventually added to the set of data δ_n of the node $n = k$. This process is detailed in Algorithm 3. On receipt of the request, the node p proceeds according one of the four following cases:

- If $p = k$ (Line 3.03), p is the proper node. Data associated with k is added to δ_p.
- If $p \in$ Prefixes (k) (lines 3.04 to 3.09), the sought node is in the subtree of p. If $\exists q \in C_p$ that shares a longer prefix with k than p, the sought node is in the subtree rooted at q, the request is forwarded to q. Otherwise, the sought node does not exist and is created as a child of p. To find a host for the new node, its whole information $(key, father, set\_of\_children, data)$ is sent to p itself using the SearchingHost message. This part of this protocol is detailed later.
- If $k \in$ Prefixes (p) (lines 3.10 to 3.20), the sought node is upward. If k is also a prefix of f_p, then the request is forwarded to f_p. Otherwise, the sought node does not exist and is created between p and f_p (the new node become the root of the tree if $f_p = \perp$). The UpdateChild(old, new) message notifies its recipient that its child a *old* must be replaced by a child *new* in its set of children.
- Finally, if none of the previous cases were satisfied, the algorithm behaves similarly than for the previous case. If f_p shares the same prefix with k as p, the request is again forwarded to f_p. Otherwise, the sought node does not exist. p and the node labeled k are siblings, but their common parent also does not exist. Two nodes are created, one to store k and one to preserve the prefix patterns inside the tree, common parent of p and k, labeled by GCP(p, k).

Once created, a new node n must find the peer on which it will run. As mentioned earlier, the SearchingHost$(key, parent, set_of_children, data)$ message initiates the search for a peer to host n. This part is detailed by lines 3.32 to 3.37. Because the first recipient of such a message always prefixes n, it remains to move the request downward until reaching the highest node lower than n. With Line 3.37, the proper peer receives the information required to host n.

```
3.01  Variables:      δ_p, set of data stored on p
                      f_p, identifier of the father of p
                      C_p, finite set of children of p
3.02  upon receipt of <DataInsertion, k> do
3.03      if k = p then δ_p := δ_p ∪ {k}
3.04      elseif p ∈ Prefixes (k) then
3.05          if ∃q ∈ C_p : | GCP(k, q)| > |GCP(k,p)| then
3.06              send(<DataInsertion, k>, q)
3.07          else
3.08              send(<SearchingHost, (k, p, ∅, {data_k})>, p)
3.09              C_p := C_p ∪ {k}
3.10      elseif k ∈ Prefixes (p) then
3.11          if (f_p = ⊥) then
3.12              send(<SearchingHost, (k, ⊥, {p}, {k})>, p)
3.13              f_p := k
3.14          else
3.15              if (|GCP(k, f_p)| = |p|) then
3.16                  send(<DataInsertion, k>, f_p)
3.17              else
3.18                  send(<SearchingHost, (k, f_p, {p}, {k})>, f_p)
3.19                  send(<UpdateChild, (p, k)>, f_p)
3.20                  f_p := k
3.21      else
3.22          if (f_p ≠ ⊥) ∧ (|GCP(k, p)| = |GCP(k, f_p)|) then
3.23              send(<DataInsertion, k>, f_p t)
3.24          else
3.25              if (f_p = ⊥) then
3.26                  send(<SearchingHost, (GCP(p, k), f_p,
                          {p, k}, ∅)>, p)
3.27              else
3.28                  send(<SearchingHost, (GCP(p, k), f_p,
                          {p, k}, ∅)>, f_p)
3.29                  send(<UpdateChild, (p, GCP(p, k))>, f_p)
3.30              send(<SearchingHost, (k, p, ∅, {k})>, f_p)
3.31              f_p := GCP(p, k)

3.32  upon receipt of <SearchingHost, (l, f, C, δ)> do
3.33      q =Max{f ∈ C_p : f ≤ l}
3.34      if (q ≠ ⊥) then
3.35          send(<SearchingHost, (l, f, C, δ)>, q)
3.36      else
3.37          send_to_host(<Host, (l, f, C, δ)>)
```

Algorithm 3: Data insertion, on node p

3.3 Load balancing

Each peer runs a set of nodes. As detailed before, the routing follows a top-down traversal. Therefore, the upper a node is, the more times it will be visited by a request. Moreover, due to the sudden popularity of some data, the nodes storing the corresponding keys, independently from their depth in the tree, may become overloaded. The heuristic we present now deals with this issue by maximizing the aggregated throughput of two consecutive peers *i.e.,* the number of requests these two heterogeneous peers will be able to process. This is achieved by periodically redistributing the nodes on the peers, based on recent history.

For the sake of clarity, we consider a discrete time and choose one particular peer S. The load balancing process is triggered on S at the end of each unit of time. Let $P = pred_S$ be the predecessor of S. Refer to Figure 3(a). C_S and C_P refer to their respective capacities *i.e.,* the

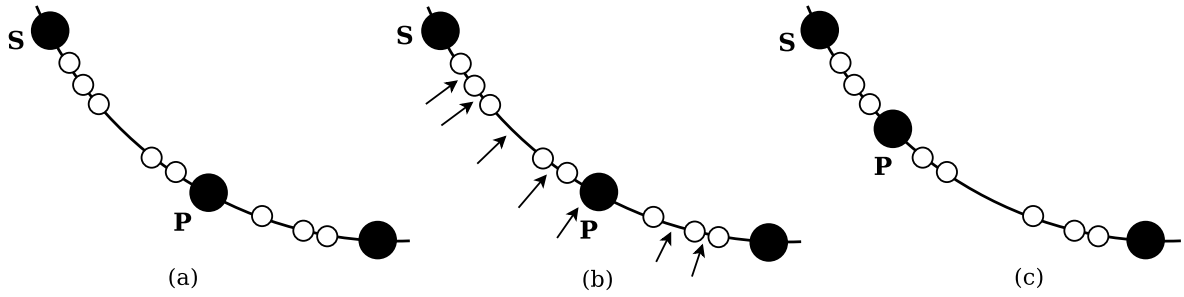


Figure 3: One local load balancing step

number of requests they are respectively able to process during one unit of time. Note that the peers capacity does not change over time. At the end of a time unit, each peer sends the number of requests received during this time unit, for each node it runs, to its predecessor. Assume that, during last time unit τ, the set of nodes runs by S and P were respectively ν_S^τ and ν_P^τ and that each $n \in \nu_S^\tau \cup \nu_P^\tau$ has received a number of request l_n. Then, the load of S during the period τ is the sum of the loads of the nodes it runs *i.e.,*

$$L_S^\tau = \sum_{n \in \nu_P^\tau} l_n.$$

We easily see that the number of satisfied requests during a time unit τ *i.e.,* that were effectively processed is:

$$T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P).$$

Starting from this knowledge *i.e.,* the load of every nodes $n \in \nu_S^\tau \cup \nu_P^\tau$, we want to maximize the throughput of the next unit of time $\tau + 1$. To do so, we must find the new distribution $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes the throughput *i.e.,* such that:

$$T_{S,P}^{\tau+1} = min(\sum_{n \in \nu_S^{\tau+1}} l_n, C_S) + min(\sum_{m \in \nu_P^{\tau+1}} l_m, C_P)$$

is maximum. The number of possible distributions of nodes on peers is bounded by the fact that nodes identifiers can not be changed, in order to ensure the routing consistency. Then, as illustrated on Figure 3, finding the best distribution is equivalent to find the ***best position*** of P moving along the ring, as illustrated by arrows on Figure 3(b). The number of candidate positions for P is $|\nu_S \cup \nu_P| - 1$. Thus, the time and extra space complexity of the redistribution algorithm is clearly in $O(|\nu_S^\tau \cup \nu_P^\tau|)$. An example of the result of this process is given by Figure 3(c). This heuristic is henceforth referred to as ***MLT*** (***Max Local Throughput***).

4 Simulation

To validate our approach, we developed a simulator of this architecture, into which we integrated two load balancing heuristics: ***MLT*** and an adaptation of a recent load balancing algorithm initially designed for DHTs known as the ***k-choices*** algorithm [11] and, to our

knowledge, the most related existing heuristic. We denote ***KC*** this adaptation. More precisely, when used, ***KC*** is run each time a peer joins the system. Because some regions of the ring are more densely populated than others, ***KC*** finds, among k potential locations for the new peer, the one that leads to the best local load balance. Please refer to [11] for more details.

We used a discrete time in the simulations. One simulation was made of a fixed number of time units. Each simulation were repeated 30, 50 or 100 times, to have some relevant results. Recall that the ***capacity*** of a peer refers to the maximum number of requests processed by it during one time unit. All requests received on a peer after it reached this number are ignored. The ratio between the most and the least powerful peers is 4. A request is said to be ***satisfied*** if it reaches its final destination. The number of peers is approximately 100, and the number of nodes around 1000. We set ***KC*** with $k = 4$. The prefix trees are built with identifiers commonly encountered in a grid computing context such as names of linear algebra routines.

We first estimated the global throughput of the system when using ***MLT***, ***KC*** or no explicit load balancing at all. Each time unit is composed of several steps. (1) If ***MLT*** is enabled, a fixed fraction of the peers executes the ***MLT*** load balancing. (2) A fixed fraction of peers join the system (applying the ***KC*** algorithm if enabled, or just the protocol detailed in Section 3, otherwise. (3) A fixed fraction of peers leaves the system. (4) A fixed fraction of new services are added in the tree (possibly resulting in the creation of new nodes). (5) Discovery requests are sent to the tree (and results on the number of satisfied discovery requests are collected).

During first experiments, services requested were randomly picked among the set of available services. Figure 4 gives the percentage of satisfied requests using ***MLT***, ***KC*** or no load balancing, for 50 time units. The first 10 units correspond to the period where the prefix tree is growing. After, it remains the same. The obtained curves show that using heuristics, and more particularly ***MLT***, leads to a non negligible gain. Figure 5 shows the results of the same experiment, but with a very high number of requests, in order to stress the system. We observe similar results, even if the satisfaction percentage is obviously globally lower.

Until now, experiments were conducted in a relatively stable network. It means that the number of peers joining and leaving the system were intentionally low. Moreover, the efficiency of the ***KC*** algorithm relies on the dynamic nature of the system since load balancing is done each time a peer joins the system. Now, 10% of the nodes are replaced at each time unit. This is why we repeated these experiments with an increased level of peers joining and leaving the network. Figures 6 and 7 give the results of the same experiments than before but conducted over a dynamic platform. We see that ***KC*** performs a bit better than previously, and gives results similar to ***MLT***.

We conducted these experiments for different loads. The results are summarized in Table 1. The percentages in the left column express the ratio between the number of requests and the aggregated capacity of all peers in the system. The table gives the gain on the number of satisfied requests of each heuristic compared to the architecture with no load balancing. The gain can be really important.

Our last simulation, whose result is illustrated by Figure 8, consisted in creating hot spots in the tree, by temporarily launching many discovery requests on some keys stored in the same region of the tree *i.e.,* lexicographically closed, in bursts. The experiment is divided in time as follows. During the first 40 time units, services are again randomly picked. Then, between 40 and 80, a hot spot is created on the particular S3L library. Most of S3L routines are named by a string beginning by "S3L". We thus overloaded the subtree containing the keys prefixed

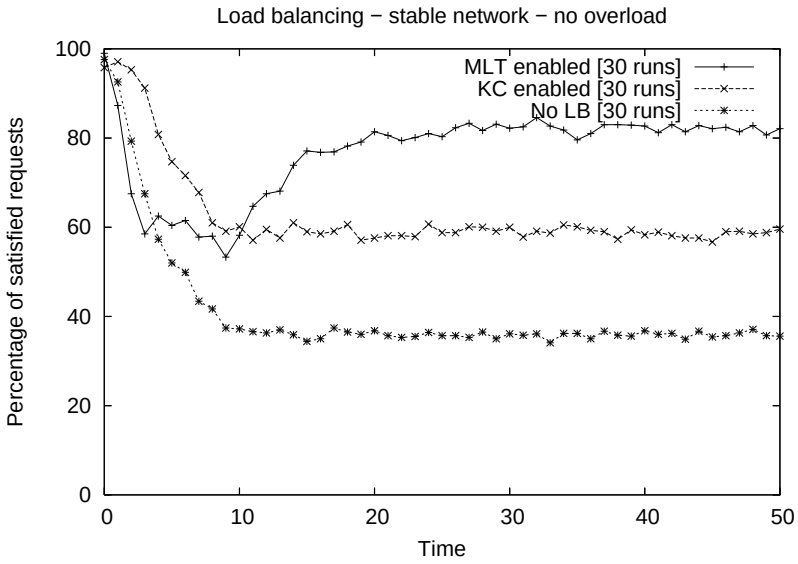


Figure 4: Stable network, low load

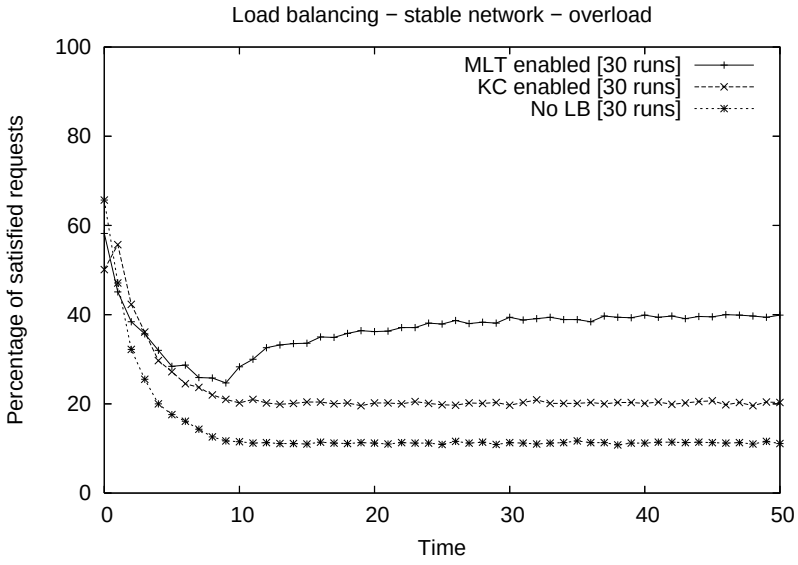


Figure 5: Stable network, high load

by "S3L". The network was previously balanced for random requests. The number of satisfied requests suddenly falls. However, the ***MLT***-enabled architecture adapts to the situation and increases the satisfaction ratio to a reasonable point. Unfortunately, a second change arises at time 80, when simulating the arrival of many requests on the ScaLapack library whose functions begin with "P". The system stabilizes again. The random way to pick services is chosen for the 40 last time units, leading to a behavior similar to the one of the beginning. Finally, as previously said, the mapping scheme is better in several ways than a random DHT-

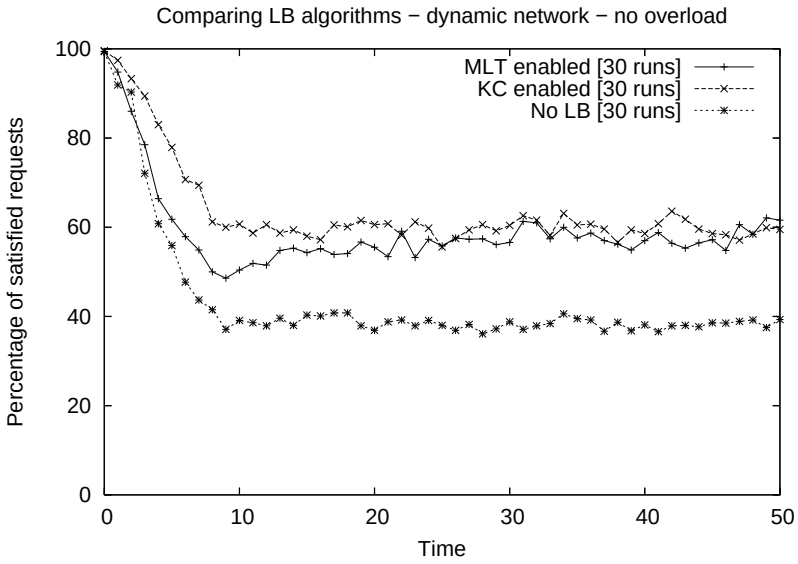


Figure 6: Dynamic network, low load

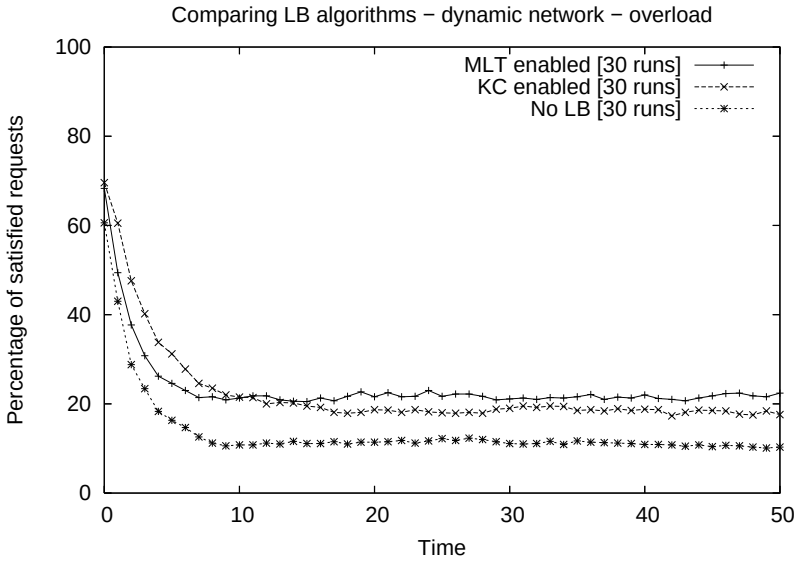


Figure 7: Dynamic network, high load

based mapping, since a random mapping results in breaking the locality. Connected nodes in the tree are randomly dispatched in random locations of the physical network. With our mapping scheme, the set of nodes stored on one peer are highly connected. This fact brings about a reduction of the communications between peers, since a high amount of routing steps in the tree involves two nodes run on the same peer. Figure 9 gives, for each time unit, the average number of hops in the tree required to reach their final destination. We see that our self-contained mapping featured with ***MLT*** significantly reduces the amount of

Load	Stable network		Dynamic network	
	MLT	KC	MLT	KC
5%	39,62%	38,58%	18.25%	32,47%
10%	103,41%	58,95%	46,16%	51,00%
16%	147,07%	64,97%	65,90%	59,11%
24%	165,25%	59,27%	71,26%	60,01%
40%	206,90%	68,16%	97,71%	67,18%
80%	230,51%	76,99%	90,59%	71,93%

Table 1: Summary of gains of ***KC*** and ***MLT*** heuristics

communications within the physical network.

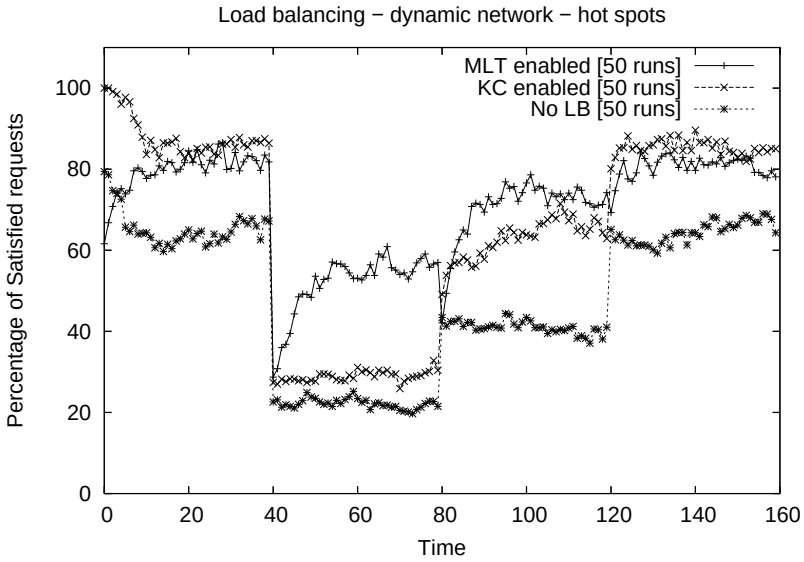


Figure 8: Dynamic network with hot spots

5 Related Work

The resource discovery in P2P environments has been intensively studied. Although DHTs [15, 18, 19] were designed for very large systems, they provided only rigid mechanisms of search. Lot of research went into finding ways to improve the retrieval process over structured peer-to-peer networks. Peer-to-peer systems use different technologies to support multi-attribute range queries [12, 3, 13, 16, 17]. In this research on multi-attribute range queries, a new kind of overlay, based on tries, has emerged. Trie-structured approaches outperform others in the sense that logarithmic (or constant if we assume an upper bound on the depth of the trie) latency is achieved by parallelizing the resolution of the query in several branches of the trie. For instance, Skip Graphs [1] are similar to a trie but are built based on skip lists. Nodewiz [2] achieves multi-dimensional range queries, but unrealistically assumes

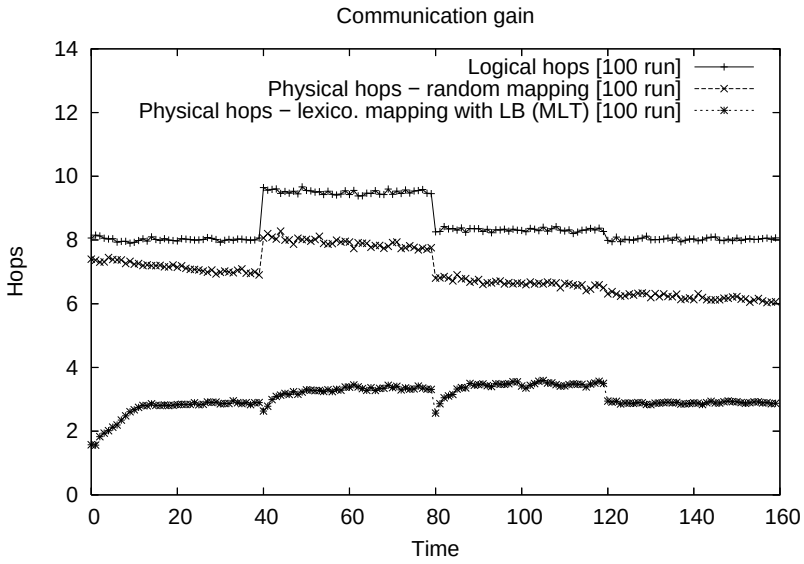


Figure 9: Reduction of the communication by the ***lexicographic*** mapping

Functionality	**P-Grid**	**PHT**	**DLPT**
Tree Routing	$O(log\|\Pi\|)$	$O(D \log P)$	$O(D)$
Local State	$O(log\|\Pi\|)$	$\frac{\|N\|}{\|P\|}\|A\|$	$\frac{\|N\|}{\|P\|}\|A\|$

Table 2: Complexities of close trie-structured approaches

that a set of static reliable nodes host the trie.

To our knowledghe, our closest related work is Prefix Hash Tree PHT [14] and P-Grid [7]. PHT builds a prefix tree over the data set on top of a DHT. The trie is used as an upper logical layer allowing complex searches on top of any DHT-like network. P-Grid builds a trie on the whole key-space, each leaf corresponding to a subset of the key-space. A fair comparison of their complexities can be achieved using Table 2. The complexities of our approach (DLPT) and the two previously mentioned approaches are quite similar. $|\Pi|$ refers to the number of partitions of the key-space, D to the maximal length of the identifiers, A to the set of digits used, N to the set of nodes of the tree and P to the set of peers.

Our contribution over PHT and P-Grid is in the load balancing process. The PHT load balancing assumes the peers homogeneous. It relies on a global threshold on the number of keys each node maintains. P-Grid relies on a set of algorithms periodically checking the load balance. In these two approaches, the heterogeneous capacities of the peers and the popularity of data are ignored. We believe that the number of keys maintained on a node does not accurately reflects its load, since it depends on what users request. Moreover, the capacities of the peers can not be supposed homogeneous in a grid computing context. For these reasons and because our architecture maintains a ***ring*** over the peers, we studied the works on the load balancing issue within DHTs. Karger and Ruhl [10], although still assuming homogeneous peers, proposed some local heuristics based on item balancing. Godfrey et al. [8] used a set of elected nodes gathering the load information and redistributing items with partial

knowledge of loads and capacities. The drawback of this approach is its ***semi-centralized*** fashion. A load balancing strategy for Chord proposed in [4] uses multiple hash function to select a number of candidate peers. Ledlie and Seltzer [11] proposed the similar ***k-choices*** approach, but assuming heterogeneity of both peers and items.

6 Conclusion and future work

The efficiency of service discovery is a crucial point in the development of fully decentralized grid middlewares. Dealing with this issue, trie-structured overlays provide some interesting characteristics. Nevertheless, its costly maintenance and several drawbacks (homogeneity assumptions, ignorance of the popularity of the services) of their load balancing techniques hinder their use within grids.

In this paper, we focused on improving these two aspects in DLPT, based on a particular kind of tries and initially designed for service discovery in a grid computing context. The first contribution is a complete protocol for a self-contained version of this architecture and the avoidance of the use of an underlying DHT. The second contribution is a novel heuristic for the load balancing inside this architecture and the adaptation to our case of recent techniques initially designed for the same purpose within DHTs. Different simulations show the gain obtained by using these heuristics. We finally propose a comparison of close approaches, in terms of complexities and load balancing.

The development of an experimentation prototype of this architecture has been undertaken, to be able to study its behavior on a real grid such as Grid'5000[1], tune its parameters and integrate it into existing grid middlewares.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.

[2] S. Basu, S. Banerjee, P. Sharma, and S. Lee. NodeWiz: Peer-to-Peer Resource Discovery for Grids. In *5th International Workshop on Global and Peer-to-Peer Computing (GP2PC)*, May 2005.

[3] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium*, August 2004.

[4] J. W. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *IPTPS*, pages 80–87, 2003.

[5] E. Caron, F. Desprez, and C. Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In *The Sixth IEEE International Conference on Peer-to-Peer Computing, P2P2006*, pages 106–113, Cambridge, UK., September 6-8 2006.

[6] Philip Chan and David Abramson. A Scalable and Efficient Prefix-Based Lookup Mechanism for Large-Scale Grids. In *3rd IEEE International Conference on e-Science and Grid Computing, e-Science 2007*, Bangalore, India, December, 10-13 2007. IEEE.

[1] www.grid5000.org

[7] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[8] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proc. IEEE INFOCOM*, Hong Kong, 2004.

[9] A. Iamnitchi and I. Foster. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *IPTPS*, pages 118–128, 2003.

[10] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *IPTPS*, pages 131–140, 2004.

[11] J. Ledlie and M. I. Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn. In *INFOCOM*, pages 1419–1430, 2005.

[12] M. Cai and M. Frank and J. Chen and P. Szekely. MAAN: A multi-attribute addressable network for Grid information services. 2(1):3–14, March 2004.

[13] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[14] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree an indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, July 2004.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[16] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[17] Y. Shu, B. C. Ooi, K. Tan, and Aoying Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, 2005.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[19] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.