A Dynamic Prefix Tree for Service Discovery within Large Scale Grids

Eddy Caron, Frédéric Desprez, Cédric Tedeschi
LIP. UMR CNRS-ENS Lyon-UCB Lyon-INRIA 5668. France
{*Eddy.Caron, Frederic.Desprez, Cedric.Tedeschi*}*@ens-lyon.fr*

Abstract

Within computational grids, some services (software components, linear algebra libraries, etc.) are made available by some servers to some clients. In spite of the growing popularity of such grids, the service discovery, although efficient in many cases, does not reach several requirements. Among them, the flexibility of the discovery and its efficiency on wide-area dynamic platforms are two major issues. Therefore, it becomes crucial to propose new tools coping with such platforms. Emerging peer-to-peer technologies provide algorithms allowing the distribution and the retrieval of data items while addressing the dynamicity of the underlying network.

We study in this paper the service discovery in a pure peer-to-peer environment. We describe a new trie-based approach for the service discovery that supports range queries and automatic completion of partial search strings, while providing fault-tolerance, and partially taking into account the topology of the underlying network. We validate this approach both by analysis and simulation. Traditional metrics considered in peer-to-peer systems exhibits interesting complexities within our architecture. The analysis' results are confirmed by some simulation experiments run using several grid's data sets.

1 Introduction

Over the last decade, grids connecting geographically distributed resources (computing resources, data storage, instruments, etc.) have become a promising infrastructure for solving large problems. However, several factors (scheduling, scalability, security, resource discovery, etc.) still hinder their worldwide adoption. Among them, the service discovery is a crucial feature to be considered. The services of a grid is the set of software components made available by some servers within the grid to some clients. Traditional service discovery approaches, efficient in a static and relatively small scale environment and based on centralized or semi-centralized architectures, lose their effectiveness in dynamic large scale environments, where future grids shall take place.

The peer-to-peer technologies provide algorithms able to retrieve objects (data items, files, etc.) in dynamic large scale environments. Iamnitchi and Foster suggested in [7] that grids, that provide the infrastructure for sharing resources but do not cope with the dynamic nature of today's platforms, would take advantage of adopting the peer-to-peer technology.

In this paper, we restrict ourselves to the following issues:

1. **Automatic completion/range queries.** For instance, a user may want to discover all the services of the SUN S3L library, whose every routine's name begins with the `S3L` string. Note that a prefix can be expressed as a range, for instance: `S3L*` $\equiv$ `[S3L;S3M[` and range queries processed similarly as partial string queries.

2. **Multicriteria search.** As services are described by a set of attributes (name of the routine, operating system, etc.), an important feature is the support of queries on several attributes.

3. **Fault-tolerance.** The tool must remain effective facing the dynamic nature of the underlying network, i.e., dynamic joins and leaves of nodes.

4. **Locality awareness.** To avoid poor routing performance, it is required to take into account the locality of nodes in the underlying physical network.

Our first intuition was to use Distributed Hash Tables (DHTs). DHTs, fully distributed self-organizing fault-tolerant systems, were initially designed for extremely large systems (such as music file sharing systems). They are scalable in the sense that the `lookup` operation, by key-based routing (KBR) requires a number of hops and a local state that typically grow logarithmically in the number of nodes. Unfortunately, DHTs present two major drawbacks. First, the logical construction of the overlay does not reflect the physical locality (IPs are randomly hashed), resulting in

poor routing performance. Second, they only support exact match queries. These drawbacks led us to propose our own architecture.

The contribution of this paper is called the *Distributed Lexical Placement Table* (DLPT) system. The DLPT is a novel architecture based on a longest prefix tree built dynamically as services are declared, supporting automatic completion of partial search string, range queries and multicriteria searches. To be effective over peer-to-peer platforms, the DLPT provides some fault-tolerance by replication and partial dynamic locality awareness. The developed algorithms are detailed in a message passing fashion. We give a validation of this architecture by detailing its complexities and then by simulating the behavior of the DLPT with several data sets reflecting services commonly available on computational grids.

Section 2 gives a brief overview of the state of the art in peer-to-peer technologies providing flexible discovery mechanisms and locality awareness. After having exposed how we model services in Section 3, the *Distributed Lexical Placement Table* (DLPT) is introduced in Section 4. Sections 5 and 6 detail the algorithms used within the DLPT. Finally, validations of the DLPT are provided in Section 7 by analysis and comparison to related works and in Section 8 by simulation.

2 Related work

As we already stated, DHTs do not address several of our requirements. First they support only exact match queries and second, their logical connections do not reflect the locality of peers in the physical network, resulting in poor performance routing.

Many solutions to inject some locality into DHTs have been formulated [9, 15, 18, 19, 20, 21]. Unfortunately, those solutions apply mainly to tori and rings, and are not trivial to adapt to prefix trees.

Dealing with the flexibility of searches over peer-to-peer networks, a series of works initiated by Harren et al. [11] and still in progress, aims at enhancing DHTs with more complex mechanisms of discovery.

INS/Twine [3] provides XML-based descriptions of resources. [17] extends traditional database operations to DHTs. Several approaches, based on space filling curves, such as [8, 16] supports multi-dimensional range queries. [1] maps one-dimensional data space to d-dimensional Cartesian space by using the inverse Hilbert mapping. Built on top of multiple DHTs, SWORD [13] is an information service aiming at discovering computing resources on the grid by answering multi-attribute range queries.

Closer to our approach, several works deal with trie-structured peer-to-peer solutions. A trie-based approach outperforms other ones in the sense that logarithmic latency is achieved by parallelizing the processing of range queries in the several subtree pertained by the range. Skip graphs [2] is a trie-structured approach also supporting range queries. The complexity in term of messages for processing range queries is in $O(m \log(n))$, m being the number of nodes pertained by a range query and n the total number of resources. PHT [14] is also close to our approach, but relies on a DHT, each routing hop in the logical trie requires a DHT lookup. Nodewiz [4], also based on a trie, do not address the dynamic joins and leaves of peers, assuming them reliable. Finally, [5] structures the overlay itself as a trie containing the complete key-space. All these approach do not consider the locality awareness issue.

The key idea of our approach is to dynamically build a reduced logical trie *a.k.a., longest prefix tree* of services being declared. Each node in the logical tree is mapped on the physical network, using a mapping mechanism, like a DHT. However, our approach is different of [14] in the sense that we use the DHT as a pool of peers, the routing is done using only the links of the tree. Finally, our scheme copes with the dynamic nature of the underlying network while partially and dynamically taking into account its locality, still using only the tree topology. It is important, to distinct our approach with previous trie-based schemes to remind the following aspects of our approach. First, our logical tree is built according to services effectively declared. Then, we achieve replication and partial locality awareness within the tree itself, periodically, without relying on an external device and in a time logarithmic in the size of the tree.

3 Modeling services

In the remainder, we restrict to the following set of attributes: **1 - The name of the service**, i.e., the name under which it is known, e.g., `DGEMM` from the BLAS [6] or `S3L_mat_mult_addto` from the SUN S3L library. **2 - The processor type of the server**, for instance to avoid users to send miscoded data, e.g., Power PC, x86, etc. **3 - The operating system of the server** that presents different characteristics and functionalities, inducing performance variations, e.g., Linux Mandrake, MAC OS X, etc. **4 - The location of the peer** allowing a client to specify a machine or a cluster he's close to or trusts. To ease the automatic completion, we specify machines/clusters/networks in reverse notation, e.g., fr.grid5000.*, edu.*, etc. The location can be specified with its IP address, too. As illustrated on the Figure beneath for the service `S` describing a `DGEMM` service, available on a server equipped with a Debian operating system and a Power PC processor, the value of the services is its location (to allow clients to connect to it.) To allow the retrieval of the service according to each of its attributes, a $(key, value)$ pair is created and stored for each

of them.

S = { DGEMM, Linux Debian 3, PowerPC G5, com.grid.n1 }

↓

```
(key, value)
(DGEMM, n1.grid.com)
(Linux Debian 3, n1.grid.com)
(PowerPC G5, n1.grid.com)
(com.grid.n1, n1.grid.com)
```

4 The Distributed Lexical Placement Table: a general description

In this section, we make a general description of the contribution of this paper, the *Distributed Lexical Placement Table* (*DLPT*).

- **DLPT functionalities** The DLPT stores services' references under the shape of $(key, value)$ pairs. The DLPT supports exact match requests, on a given key, partial search strings by providing automatic completion. For instance, let us assume services are described by their name, a client sending the request DTR will receive all services whose name begins with the DTR string, for instance DTRSM, DTRMM or DTRSV. It also supports, similarly, range queries. Multi-attribute search can be achieved by a simple extension.

- **Logical architecture.** The logical structure used within DLPT is a reduced trie *a.k.a.*, a longest prefix tree. We call the basic entity of this trie a **logical node**. Each logical node are identified by one given key. We consider two types of keys: A node identified by a **real key** stores the reference of at least one service. For instance, DGEMM is considered as a real key as soon as a server has declared a service under the DGEMM name. Note that by construction, the leaves of the tree are identified by real keys. A node identified by a **virtual key** is the root of a subtree whose nodes' IDs share this virtual key as common longest prefix. Figure 1 shows the construction of such a tree, when three services are declared sequentially.

- **Mapping the logical tree on the physical network** The logical nodes are distributed on the physical nodes of the underlying network. Let's call them **peers**. A logical node is *hosted* by a peer. A peer has the ability to host zero, one or more logical nodes, each logical node being a process running on it. This mechanism can be achieved in different ways. One approach is to use a DHT, but any tool acting as a repository (distributed or not) can replace it.

- **Routing complexity.** Whereas logical nodes of DHTs represent physical nodes, logical nodes of the DLPT represent keys of declared services. Thus, the trie grows according to the number of distinct real keys declared. We detail complexity considerations in Section 7.

- **Fault-tolerance** The DLPT is designed to take place in a dynamic environment. It provides a mechanism of replication of the nodes and links of the trie, in order to remain efficient facing the departure of peers.

- **Locality awareness** A greedy heuristic periodically determines a spanning trie of the replicated one thus providing a partial locality awareness within the trie.

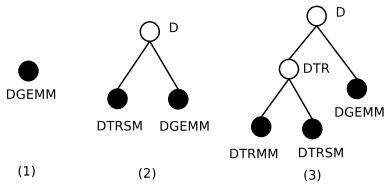


Figure 1. Construction of a longest prefix tree. Nodes storing some services' references (labeled by real keys) are black filled, the others are labeled by virtual keys. (1) First a DGEMM is declared. (2) A DTRSM is declared resulting in the creation of their parent, whose ID is their longest common prefix D. (3) Finally, a DTRMM is declared and the node DTR is created.

5 Creation and maintenance of the DLPT

5.1 Constructing and mapping the tree

First recall that services are declared in a dynamic manner. We do not build a trie of the entire key-space and then map it dynamically on the network, as several previous approaches, we dynamically build it according to services being declared.

We now consider the insertion of one $(key, value)$ pair. The pair is placed inside the tree according to the key. Like in a DHT, the server that declares a service obtains the address of a peer hosting a logical node of the tree by an out-of-band mechanism (name server, web page, ...) and sends an **insertion request** to it. The request is routed within the prefix tree until reaching the node that will effectively insert the pair. A gain of time could be achieved if sending all requests to the root, but it would require to know it from anywhere. Each node, on receipt of an insertion request on

the $S = (key = k, value = v)$ pair applies the following routing algorithm, considering four distinct cases:

k is equal to the local node identifier. In this case, k is already in the tree, no node need to be added, the logical node inserts v into its table.

k is prefixed by the local node identifier. The local node search among its children identifiers, one key that shares one more character than itself with k. If such a child exists, the request is forwarded to it, else, no node identifier in the tree prefixes k with more characters than the local node identifier. A new logical node is created as a child of the local node and hosted by a peer, v is inserted in the table of the new node.

The local node identifier is prefixed by k. In this case, if the identifier of the parent of the local node is equal to or prefixed by k too, S must be inserted upper in the trie and the local node forwards the request to its parent. Otherwise, S must be inserted in this branch, between the local node and its parent. Such a logical node is created, hosted and given to insert v into its table.

Default If the local node has a parent and if the identifier of the parent of the local node is equal to or prefixed by the common prefix of k and the local node identifier, the local node forwards the request to it. Otherwise, the logical node storing k and the logical node are siblings. However, their common parent does not exist (recall the example on the Figure 1). Two nodes must be created, the future node identified by k and storing S (sibling of the local node) and their parent whose identifier is the common longest prefix of k and the local node identifier (possibly the empty string).

We now briefly discuss how to map the tree onto peers. A solution is to structure the network within a DHT and then to choose a peer to host a given node by using the DHT hash function on the node ID. Indeed, any DHT could be used. Remind that we only use the DHT as a pool of peers. Thus the insertion of a new peer inside the DHT and the resulting possible redistribution of the data between peers is not applied to the logical nodes. An issue we do not consider in this paper is related to load balancing. Obviously, using a DHT to uniformly distribute the logical nodes on the peers does not achieve an efficient balancing of the workload, mainly for the following reason. The load of a node depends on the popularity of the service it stores and on its depth in the tree (nodes close to the root are more solicited than leaves when routing requests). A first simple solution is to tune the replication factor locally to balance requests for a given logical node among its different replicas. Another solution is to rely on the DHT for this issue. DHTs make two common assumptions. First, they consider the capacities of peers homogeneous what can not be ensured on real grids. They also assumes that each data item has the same probability to be requested. We do not discuss more this issue in this paper and let it for future work. We consider that the load balancing is achieved independently within the DHT. We rely on several recent works addressing the heterogeneity of both the capacity of peers and popularity of keys inside DHTs [10, 12]. To adapt these works to our case, it suffices to replace data items traditionally considered in DHTs by our logical nodes.

Algorithm 5.1 gives the detailed pseudo-code executed on a node receiving an insertion request. The **COMMONPREFIX** function returns the longest common prefix of two strings. The **NEWNODE** function creates a new logical node. The **GETPEER** function calls the underlying mapping mechanism and returns the reference of a peer. The `hostReq` request is sent to the peer designated to host a newly created node. The `updateChild` and `addChild` requests are sent to nodes that must update their references to their children. The code executed inside these functions and on receipt of these requests are not given because they are algorithmically trivial.

5.2 Fault-tolerance and locality

To face the dynamic nature of the underlying network and to ensure the consistency of the routing, we propose a replication scheme. The replication factor k, statically fixed, denotes the number of distinct peers on which each logical node must be present. Such a replicated trie is shown in Figure 2 with $k = 2$.

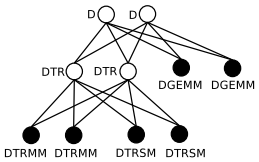


Figure 2. Example of a replicated trie.

To achieve locality awareness within the trie, we try to minimize the communication time in the replicated trie, by choosing the best peer/replica for each logical node. For this purpose, recall that each node of the trie have a semantic and we must keep one instance of each node in this spanning trie, making this process somewhat different of a traditional spanning tree algorithm. In addition, each node has knowledge only about its parent and its children. An optimal spanning trie would require the storage on each node of a routing table of size linear in the size of the network and a complexity of the algorithm quadratic in the size of the trie. Because these aspects would compromise the scalability of

Algorithm 5.1 Insertion of a new service

Constants:
loc: local logical node
$loc.ID$: ID of loc

Variables:
$loc.children$: set of children of loc
$loc.parent$: parent of loc
$loc.host$: address of the peer hosting loc
$prefix$: string
$prefixParent$: string

Upon **RECEIPT** of $<$ logReq, $ID>$
$prefix$:= **COMMONPREFIX**($ID, loc.ID$)
if (**SIZEOF**($prefix$) = **SIZEOF**($loc.ID$) = **SIZEOF**(ID)) **then**
// Node found. Storing the new service.
elseif (**SIZEOF**($prefix$) = **SIZEOF**($loc.ID$)) **then**
if ($\exists f \in loc.children$ | **SIZEOF**(**COMMONPREFIX**($f.ID, ID$)) $>$ **SIZEOF**($loc.ID$)) **then**
SEND $<$ logReq, $ID>$ TO f
else // A node n is created as a child of the local node and hosted
n:= **NEWNODE**($ID, parent = loc, children = \emptyset$)
$n.host$:= **GETPEER**()
SEND $<$hostReq, $n>$ TO $n.host$
$loc.children$+= $\{n\}$
endif
elseif (**SIZEOF**($prefix$) = **SIZEOF**(ID)) **then**
if ($loc.parent = \perp$) **then**
//loc is the current root
n:= **NEWNODE**($ID, parent$:=$\perp, children$:=$\{loc\}$)
$n.host$:= **GETPEER**() // but its parent is created
SEND $<$hostReq, $n>$ TO $n.host$ // and hosted
$loc.parent$:=n
else
$prefixParent$:=**COMMONPREFIX**($ID, loc.parent.ID$)
if (**SIZEOF**($prefixParent$)=**SIZEOF**(ID)) **then**
SEND $<$logReq, $ID>$ TO $loc.parent$ // going up
else // A node is created between loc and $loc.parent$
n:= **NEWNODE**($ID, parent$:=$loc.parent, children$:=$\{loc\}$)
$n.host$:= **GETPEER**()
SEND $<$hostReq, $n>$ TO $n.host$
SEND $<$updateChild, $n>$ TO $loc.parent$
$loc.parent$:=n
endif **endif**
else
if ($loc.parent = \perp$) and ((**COMMONPREFIX**($prefix, loc.parent.ID$) = **SIZEOF**(prefix)) **then**
SEND $<$ logReq, $ID>$ TO $loc.parent$
else // loc and the new node n are siblings, they need a parent p
p:=**NEWNODE**($prefix, parent$:=$loc.parent, children$:=$\{loc\}$)
$p.host$:=**GETPEER**()
n:=**NEWNODE**($ID, parent$:=$p, children$:=$\{loc\}$)
SEND $<$ hostReq, $p>$ TO $p.host$
SEND $<$ hostReq, $n>$ TO $n.host$
SEND $<$ addChild, $n>$ TO p
$loc.parent$:=p
endif
endif

the system, the only possible minimization is a local one. We use a greedy heuristic locally choosing the best peer among the replicas of each logical node. This heuristic is integrated to the replication process, without modifying its time complexity, bounded by the depth of the trie thanks to the parallelism achieved by treating each branch in parallel.

The replication process enhanced with greedy locality awareness, fully described by the part executed only by the root in Algorithm 5.2, periodically initiated by one of the current roots of the tree (on the Figure, there's only one root (1)), starts by the replication of the root itself. The roots of the tree, and only them shape a fully-connected network, so each root has knowledge about its replica. Each root being a potential starter of the replication process, we use a simple mutual exclusion scheme, not detailed here. The elected root initiates the wave by testing the number of its replicas, let k' be this number. It replicates $k - k'$ times itself on peers it discovers via the mapping mechanism used. Once the root is replicated, it sends a `scanReq` request to itself, initiating the replication of the trie (2).

On receipt of a `scanReq` request, a node behaves as described in the part common to all nodes in Algorithm 5.2. It treats its logical children one by one. For each of them, the local node tests the number of reachable replicas, gets the references of peers needed to reach k replicas for this child and sends a `replicationReq` request to one of the current available replicas that will send its logical node structure to the peers obtained. It then determine the best peer/replica after replication for this child (through the **GETBESTREPLICA** function) and sends a `scanReq` request to the peer/replica which minimizes the communication time with itself thus launching the replication in this subtrie, then continuing asynchronously under each children of the root (3, 4). Note that the purpose of the local choice of the best replica of each logical child is twofold. First it determines which replica/peer will be used for the routing to this child until the next replication process starts. Then it designates the replica/peer responsible for the replication of the subtrie of this child.

Algorithm 5.2 Initialization of the scanning process

Constants:
loc: the local node
k: replication factor

Variables:
$loc.children$: set of children of loc
$n.R$: set of replicas of the node n

// On the root only
// Replicating the root, periodically
k':=**GETNBREPLICAS**(loc)
while $k' < k$ **do**
p:=**GETPEER**()
SEND $<$ hostReq, $loc>$ TO p
k'++
for all $\{f \in loc.children\}$ **do**
// Informing my children of their new parent
SEND $<$ addParent, $p>$ TO f
done
$loc.R$+=$\{p\}$
done
// Launching the replication in the trie
SEND $<$scanReq$>$ TO loc
// On every node
Upon **RECEIPT** of $<$ scanReq$>$
for all $\{f \in loc.children\}$ **do**
k':= **GETNBREPLICAS**(f)
while $k' < k$ **do**
p:= **GETPEER**()
SEND $<$replicationReq, $p>$ TO f
$f.R$+=$\{p\}$
done
$next$:= **GETBESTREPLICA**($f.R$)
// Launch the scan in this subtrie
SEND $<$ scanReq$>$ TO $next$
done

6 Interrogating the DLPT

We now describe the mechanisms allowing the service discovery according to a key or a range of keys.

To process a discovery request according to a key, i.e., the traditional `lookup` operation of DHTs, the DLPT executes the algorithm illustrated in Figure 4(a). The request is sent to a given node of the tree by the client, is routed in a way similar to the one used for an insertion request. The

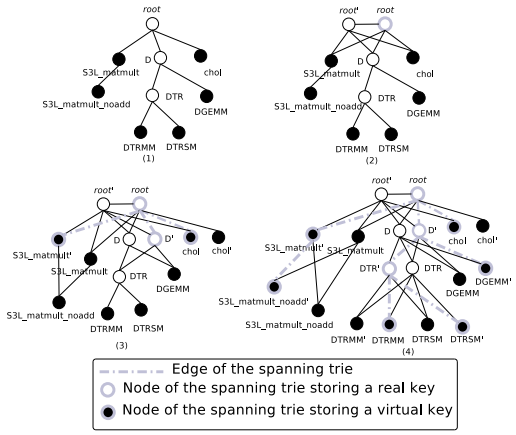
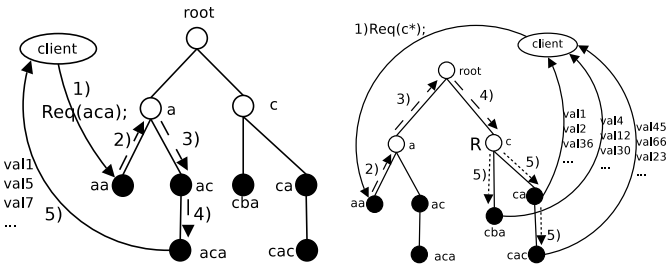


Figure 3. Replication and locality.



a. Full key b. Partial key

Figure 4. The client sends a discovery request to a node it knows (1). The request is routed (2,3,4). Responses are sent back to the client from the node storing the key or from the subtree whose root is the responsible node.

destination node is the one that stores the key requested by the client, i.e., the node whose identifier is the requested key. Finally, the node storing the key wanted sends the corresponding values of services back to the client.

The processing of partial keys request is made of two steps, shown on Figure 4(b). Let us consider the `DTR*` request. The request is first routed according to `DTR`, as for an exact key, except that the destination node is not the node identified by the requested key, but the node identified by the smallest key in the tree prefixed by the requested key. Let us call this node the *responsible node* of this request. The requested keys are in the subtree whose root is this responsible node. Once the responsible node found, it remains to traverse in parallel every nodes of its subtree. Each node sends its values to the client and forwards the request to its children. The client can stop listening the responses if satisfied with the values received. Note that a range query can be achieved in a similar way than automatic completion: The bounds of a range query have a common prefix. It suffices to route the request according to this prefix and then to launch the asynchronous traversal of its subtree, forwarding the request on each receiving node only to its own subtrees whose set of IDs potentially covers the range.

Finally, for multi-attribute searches, we create one tree for each attribute, and each $(key, value)$ pair is indexed within the tree corresponding to the attribute described by the key. To maintain only one tree for every attributes could result in undesired behavior for instance if a service is called like a peer. To be sure of the nature of the information searched, we build one tree per attribute. Considering our model described in Section 3, four longest prefix trees are built. The value (location) of the service will be stored by sending an insertion request to each tree. To perform an interrogation on several attributes, the client sends one request to a node of each tree. For instance, to discover services matching the request {`DTRSM, Linux*, PowerPC*, *`}, the client will send three requests (the reversed address is here not requested by the client). The request on `DTRSM` will be sent to the services' tree, `Linux*` to the system tree and `PowerPC` to the processors tree. Requests are independently processed by each tree and the client asynchronously receives the values and finally just needs to intersect the locations obtained to keep what really matches its request.

7 Analysis of the DLPT

We now detail the complexities of the DLPT dealing with the metrics used in P2P networks. Let us consider a prefix tree of size n, A the alphabet that would be used to generate keys stored in the tree. If we assume a max bound T_{max} on the size of the keys, what seems realistic, the depth of the tree is also bounded by T_{max}. In the worst case, a request must be routed from a leaf to another via the root, what induces that the number of hops is bounded by $2 \times T_{max} = O(T_{max}) = O(1)$. Otherwise, the depth is in average logarithmic in the size of the tree. For requests requiring the completion of a partial string, the number of hops required to reach the responsible node is again $O(T_{max})$. Then, one can not avoid the traversal of all the nodes in the subtree. This traversal is done in parallel in each branch of the tree, again resulting in a time complexity in $O(T_{max})$. The number of messages required is in $O(n)$. A multicriteria request is also achieved in parallel within each tree, resulting in a time complexity bounded by the maximum of the T_{max} of the trees. Also considering A as a finite set, each node maintaining, by construction, an entry for each potential character within its routing table, the size of the routing table is bounded by $|A|$. Practically, it means that the routing table can be statically allocated (for instance as a vector of $|A|$ cells). As a consequence, the routing de-

Functionality	Skip Graphs	PHT	P-Grid	DLPT
Insertion	$O(\log(n))$	$O(D)$	$O(log(\Pi))$	$O(T_{max})$
Lookup	$O(\log(n))$	$O(\log(D)\log(N))$	$O(log(\Pi))$	$O(T_{max})$
Range messages	$O(m\log(n))$	$O(o)$	$O(\Pi_R)$	$O(m)$
Range time	$O(\log(n))$	$O(D)$	$O(log(\Pi))$	$O(T_{max})$
Fault-tolerance	repair	DHT-based	replication	replication
Locality	-	-	-	greedy

Table 1. DLPT and other approaches

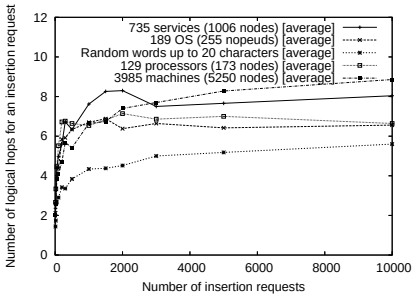


Figure 5. Average number of logical hops.

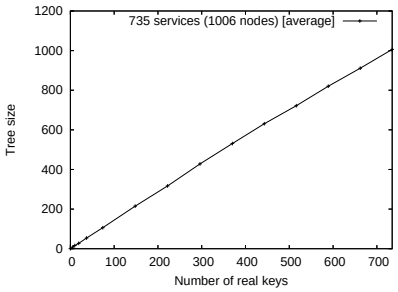


Figure 6. The size of the tree is proportional to the number of inserted keys

cision on each node can be achieved in $O(1)$ by scanning the cell corresponding to the next character searched.

Table 1 summarizes several aspects of our related work compared to us. Let us briefly compare each approach with ours. Skip Graph builds a skip lists based trie in which each resource is a node. The number of messages required to process a range query within Skip graphs is in $O(m\log(n))$, m denoting the number of resources within the range *i.e.*, a $log(n)$ factor more than in our architecture. Prefix Hash Tree builds a logical trie whose leaves managed the keys corresponding to its branch and are mapped onto peers of an underlying DHT. Since the trie is built on top of a DHT, the lookup complexity is in $O(\log(D)\log(N))$, N denoting the size of the DHT and D the max size of the keys. o denotes the size of the output of a range query. P-Grid builds a trie with the whole key-space, which size is denoted Π. Each leaf corresponds to a given prefix and is associated with a peer. The depth of the P-Grid trie is static in $O(\log(\Pi))$. Π_R the size of the interval of a range query R. Nodewiz assumes a stable underlying network, what makes it difficult to use in peer-to-peer environments. Contrary to those approaches, our architecture builds a dynamic longest prefix tree that better reflects the set of services declared, thus avoiding useless hops, and practically rarely reaching the max bound T_{max}. l denotes the size of the subtree pertained by a range query. As a more general comment, only the DLPT, even partially, achieves some locality awareness.

8 Simulation

A simulator implementing the dynamic creation of the tree and its interrogation with exact and partial keys has been developed. It has been tested with computational grids data sets taken from real grids: 735 names of services, 129 names of processors, 189 OS names and 3985 names or IPs of machines. We first tested the number of logical hops when processing an insertion request. Figure 5 shows the number of logical hops to process the request by choosing a random contact node. For these experiments, the four data sets plus a data set containing 10000 random strings have been used. The curve follows a logarithmic behavior, even for the set of 10000 random strings, illustrating the scalability of the system.

We have also studied how the tree grows according to the number of distinct declared keys. Each key of each data set is now inserted once. As we see on Figure 6, the total number of nodes in the tree (identified by virtual keys or real keys) is proportional to the inserted keys (real keys). The whole set of experiments shows a reasonable proportion of nodes storing virtual keys, near 30% with a standard deviation of 2.4%.

Finally we have studied the number of logical hops on the submission of interrogation requests. The results illustrated on Figure 7 are similar to those observed on insertion requests.

9 Conclusion

We have described a novel tool, enhancing computational grids with a peer-to-peer approach offering a flexible large scale service discovery by supporting multicriteria range queries, while providing fault-tolerance and taking into account the underlying locality. Traditional metrics exhibit interesting complexities within our architecture. This is, to our knowledge, the first tree-based approach injecting some locality directly within the tree structure. We are currently studying some repair mechanisms within the tree, as an alternative to the replication process. We are also car-

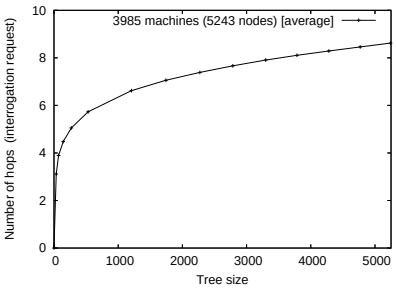


Figure 7. Number of logical hops for interrogation requests on exact keys.

rying out a more theoretical study of the potential gain of mapping trees over DHT-like networks. We also focus on locality issue in the same way. Finally, we plan to develop an implementation of the DLPT, to validate it on large scale platforms and tune parameters like the replication factor.

References

[1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Peer-to-Peer Computing*, pages 33–40, 2002.

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

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Pervasive*, 2002.

[4] 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.

[5] 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.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[7] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *IPTPS*, pages 118–128, 2003.

[8] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *WebDB '04: The 7th International Workshop on the Web and Databases*, pages 19–24, 2004.

[9] L. Garces-Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. Urvoy-Keller. Hierarchical Peer-to-Peer Systems. *Parallel Processing Letters Volume 13, 2003*.

[10] 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.

[11] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-Based Peer-To-Peer Networks, 2002.

[12] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *Proc. IEEE INFOCOM*, Miami, 2005.

[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*, St. John's, Newfoundland, Canada, 2004.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, 2001.

[16] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[17] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P 2003*, September 2003.

[18] Z. Xu, M. Mahalingam, and M. Karlsson. Turning Heterogeneity into an Advantage in Overlay Routing. In *INFOCOM*, 2003.

[19] Z. Xu, R. Min, and Y. Hu. HIERAS: A DHT Based Hierarchical P2P Routing Algorithm. In *ICPP*, 2003.

[20] Z. Xu and Z. Zhang. Building Low-Maintenance Expressways for P2P Systems. Technical report, Hewlett-Packard Labs, April 2002.

[21] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *IPTPS 2002*.