Découverte de services pour les grilles de calcul dynamiques large échelle

Cédric Tedeschi

Laboratoire de l'Informatique du Parallélisme

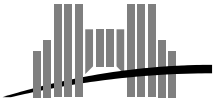
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Découverte de services pour les grilles de calcul dynamiques large échelle

Cédric Tedeschi Nov 2006

Research Report N° 2006-44

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



Découverte de services pour les grilles de calcul dynamiques large échelle

Cédric Tedeschi

Nov 2006

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.

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

Résumé

Dans les grilles des serveurs offrent des services aux clients afin de réaliser des calculs. Avant de pouvoir les utiliser, les clients doivent être à même de les retrouver. Bien que les différentes solutions proposées depuis l'émergence des grilles soient efficaces sur des plates-formes relativement statiques et de petite échelle, elles ne sont plus en adéquation avec la nature dynamique et à large échelle des grilles futures. Pour de tels environnements de nouveaux outils doivent être proposés, notamment des mécanismes pour la découverte de services, qui devront être *flexibles* et passer à l'échelle dans des environnements dynamiques. Nous étudions dans ce papier la découverte de services pour des grilles de calcul pair-à-pair. Nous proposons une nouvelle architecture basée sur un arbre de plus long préfixes.

Mots-clés: Découverte de services, grilles de calcul, pair-à-pair, arbres de préfixes

1 Introduction

Ces dernières années, les grilles connectant des ressources distribuées sont devenues une alternative efficace pour résoudre des problèmes de grande taille. Toutefois, le déploiement des grilles de calcul est une tâche difficile et de nombreux problèmes sont à prendre en considération tels que la sécurité, l'ordonnancement, la gestion des ressources, etc. Dans cet article nous nous intéresserons à la découverte de services.

Les services d'une grille sont un ensemble de composants logiciels (bibliothèques de calcul, exécutables, etc.) que des serveurs mettent à la disposition des clients. Depuis l'émergence des grilles, différents mécanismes de découverte des services ont été utilisés. Ces outils traditionnels, efficaces pour les plates-formes relativement statiques et à petite échelle et fondés sur des mécanismes centralisés ou semi-centralisés, perdent leur efficacité dans des environnements large échelle où les nœuds peuvent quitter le réseau à tout instant sans prévenir. Iamnitchi et Foster ont récemment suggéré que les grilles, qui fournissent une infrastructure pour le partage de ressources, mais perdent leur efficacité sur des plates-formes dynamiques grande échelle, gagneraient à s'inspirer des mécanismes pair-à-pair [6]. Les technologies pair-à-pair fournissent des algorithmes permettant la recherche d'objets. Parmi elles, les tables de hachage distribuées (DHT) [13, 15] ont été initialement développées pour des systèmes très large échelle tel que le partage de fichiers sur Internet. Bien que les DHT passent à l'échelle dans le sens où une recherche demande un nombre de sauts et une taille de table de routage logarithmiques en la taille du système et sont tolérantes aux pannes, elles présentent deux défauts majeurs. D'abord, le placement des nœuds dans le réseau logique ne tient pas compte de la topologie physique du réseau, ce qui entraîne une perte importante des performances. Ensuite, leurs mécanismes de découverte sont très rigides, puisqu'ils ne permettent que des recherches exactes, sur une clef précise. La section 2 décrit les travaux actuels cherchant à résoudre ces problèmes.

Dans notre approche, les services sont préinstallés sur les serveurs et les clients cherchent à les découvrir. Un service est décrit par un ensemble d'attributs, par exemple le nom du service, le processeur et le système du serveur qui le fournit. Nous découpons le problème selon différents critères.

Tout d'abord, nous souhaitons supporter la **complétion automatique** des chaines de recherche partielle et les requêtes sur des **intervalles** de clefs. On remarquera que ces deux fonctionnalités sont similaires. Une autre fonctionnalité importante est la recherche **multi-critères**. Enfin, pour faire face aux besoins des environnements pair-à-pair, il est crucial d'intégrer de la **tolérance aux pannes** pour assurer la cohérence du routage si des nœuds quittent le réseau. Enfin, pour éviter de grandes latences lors des requêtes et ainsi rendre l'architecture inutilisable à large échelle, il est important que la topologie logique tienne compte de la **localité physique** du réseau sous-jacent.

Notre contribution, la Table de Placement Lexicographique Distribuée (*TPLD*) est une nouvelle architecture qui utilise une approche pair-à-pair basée sur un arbre de plus long préfixes pour la découverte de services en fournissant les mécanismes nécessaires à la mise en œuvre des fontionnalités décrites au-dessus.

Dans la section suivante, nous donnons un rapide résumé de l'état de l'art des solutions pour des recherches complexes dans les systèmes pair-à-pair et pour la prise en compte de la topologie du réseau physique (on parle aussi de localité). Ensuite, après avoir précisé la modélisation des services, nous exposons le principe général de la TPLD section 3. Nous détaillons ses mécanismes et algorithmes dans les sections 4 et 5. La section 6 présente une

validation analytique de nos travaux qui est accompagnée par nos résultats par simulation dans la section 7.

2 Etat de l'art

Comme nous l'avons déjà vu, les deux principaux inconvénients actuels des DHT sont la rigidité de leur recherche et la non prise en compte de la topologie du réseau sous-jacent.

La formulation de solutions pour l'injection de localité physique dans le réseau logique a engendré une littérature importante [20, 7, 17, 13, 19, 18]. Toutefois, ces travaux s'appliquent à des topologies en anneau et en tore, leur exploitation dans un arbre de préfixe n'est pas trivial.

L'autre inconvénient majeur des DHT est leur mécanisme de recherche rigide qui est basé sur une clef unique. Une série de travaux initiée par [9] a pour but de supporter des requêtes complexes dans des systèmes pair-à-pair structurées. INS/Twine [3] permet une description semi-structurée (en XML) des objets recherchés. [16] étend les opérations traditionnelles des bases de données dans les systèmes pair-à-pair. Squid [14] permet des recherches multi-critères et la complétion automatique des chaînes de recherche. Les travaux d'Andrzejak et Xu [1] par exemple supportent les recherches unicritères sur un intervalle de valeurs en utilisant les courbes de remplissage de l'espace de Hilbert. Construit au-dessus de plusieurs DHT, SWORD [11] est un service d'information supportant les requêtes multi-critères sur des intervalles dont le but est la découverte de ressources de calcul.

Plus proche de notre architecture, différents travaux se basent sur des arbres lexicographiques, dont le parallélisme permet une latence pour les requêtes sur un intervalle logarithmique en la taille de l'arbre. Skip graphs [2] est une structure de données distribuée fondée sur les skip list, où chaque ressource est un nœud du graphe. PHT [12] construit un arbre lexicographique des clefs au-dessus d'une DHT, dont les complexités se multiplient. Nodewiz [4] s'appuie de même sur un arbre lexicographique, mais se place sur une plate-forme où les nœuds sont considérés stables. P-Grid [5] construit de même un arbre lexicographique avec l'ensemble de l'espace des clefs. Une caractéristique commune à toutes ces approches est qu'elles ne prennent pas en compte la topologie physique de la plate-forme. Nous comparons ces approches avec notre architecture dans la section 6.

L'idée clef de la TPLD est de construire dynamiquement un arbre logique des services, et de le plonger dans le réseau physique, grâce à un mécanisme distribué comme une DHT. Il est toutefois important de dissocier plongement et routage des requêtes dans l'arbre qui se fait indépendamment, contrairement à [12]. Nous utilisons un mécanisme de réplication pour prévenir les déconnexions de nœuds et nous nous adaptons partiellement à la topologie physique.

3 Table de Placement Lexicographique Distribuée

Nous faisons ici une description générale de notre contribution. Il est pour cela nécessaire de mieux définir ce que sera un service dans la TPLD. Les services sont traditionnellement décrits par un ensemble d'attributs. On considère les attributs suivants: **1 - Le nom du service**, sous lequel il est connu par les utilisateurs. Par exemple `DGEMM` de la bibliothèque BLAS. **2 - Le processeur du serveur**, par exemple pour des problèmes de codage des données. **3 - Le système du serveur** pour des questions de fonctionnalités ou de performances. **4 - La**

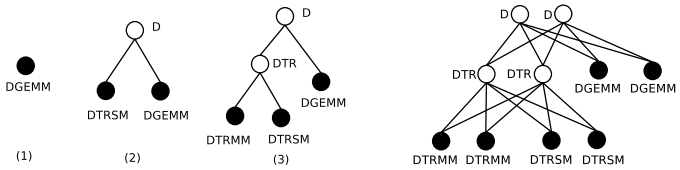


Figure 1: Exemple de construction de l'arbre: Les nœuds coloriés représentent des nœuds aux clefs réelles. (1) Un service `DGEMM` est déclaré. (2) Un service `DTRSM` est déclaré nécessitant la création du nœud à la clef virtuelle `D`, plus long préfixé des deux clefs réelles. (3) Un service `DTRMM` est déclaré. A droite, un exemple d'arbre répliqué.

localisation du serveur pour des questions de confiance ou de proximité avec le client. Pour faciliter l'autocomplétion des localisations, on spécifiera les machines en notation inversée, par exemple, fr.grid5000.* si l'on veut une machine de la grille Grid'5000, où encore edu.*, etc. L'exemple ci-dessous illustre la description d'un service `DGEMM` fourni par un serveur de Grid'5000 dont le système d'exploitation est une Debian et le processeur un PowerPC. Chacun des attributs est associé à la localisation du service pour former les couples $(clef, valeur)$ (ici, à droite) qui seront déclarés et indexés dans la TPLD.

Service = { `DGEMM`,
`Linux Debian 3.0`,
`PowerPC G5`,
`fr.grid5000.lyon` }

`(DGEMM, lyon.grid5000.fr)`
`(Linux Debian, lyon.grid5000.fr)`
`(PowerPC G5, lyon.grid5000.fr)`
`(fr.grid5000.lyon, lyon.grid5000.fr)`

En terme de fonctionnalités, la TPLD indexe et recherche des références de services sous forme de couples $(clef, valeur)$. Elle fournit l'autocomplétion des chaînes de recherche partielles. Par exemple, un client peut découvrir les services BLAS préfixés par la chaîne `DTR`. Il recevra tous les services `DTRMV`, `DTRSV`, `DTRMM` et `DTRSM`. La TPLD s'appuie sur un placement lexicographique des attributs dans l'arbre.

L'architecture logique de la TPLD sont des arbres lexicographiques réduits, aussi appelés *arbres de plus long préfixe*. L'entité de base de la TPLD est un nœud de ces arbres, que nous nommerons *n*œud logique. Un nœud logique stocke les références des services déclarés par les clients. L'identifiant d'un nœud correspond à la clef des services qu'ils stockent. On distingue deux types de clefs. Des clefs *réelles* qui correspondent à des services réellement déclarés. On remarquera que les feuilles de l'arbre correspondent à des clefs réelles. Des clefs *virtuelles* qui sont nécessaires à la construction de l'arbre. Notons qu'une clef virtuelle est la racine du sous-arbre dont les nœuds ont pour identifiant des clefs préfixées par cette clef virtuelle. La figure 1 décrit la construction dynamique d'un arbre qui reçoit la déclaration de trois services.

Les nœuds de l'arbre logique doivent être distribués sur les nœuds physiques du réseau physique sous-jacent. On appellera les nœuds physiques des *p*airs qui vont héberger, sous forme de processus un ou plusieurs nœuds logiques. Ce plongement peut-être réalisé de plusieurs manières. Un exemple distribué est d'utiliser une DHT et de placer un nœud logique sur le nœud physique qui correspond au hachage de l'identifiant du nœud physique. Enfin, on se place dans un environnement dynamique. La TPLD offre un mécanisme de réplication des arbres de préfixes pour assurer le routage en cas de départ de pairs. Par ailleurs, on utilisera une heuristique gloutonne permettant l'adaptation partielle à la topologie physique du réseau

sous-jacent.

Algorithme 3.1 Insertion et plongement (gauche), réplication et localité (droite)

```
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 (∃f ∈ 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.parent, children = ∅)
            n.host:= GETPEER ()
            SEND <hostReq, n> TO n.host
            loc.children+= {n}
        endif
    elseif (SIZEOF (prefix) = SIZEOF (ID)) then
        if (loc.parent = ⊥) then
            //loc is the current root
            n:= NEWNODE (ID, parent:=⊥, 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 = ⊥) 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
```

```
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 ∈ loc.children} do
        // Informing my children of their new parent
        SEND < addParent, p> TO f
    done
done
loc.R+={p}
// Launching the replication in the trie
SEND <scanReq > TO loc
// On every node
Upon RECEIPT of < scanReq >
    for all {f ∈ 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
```

4 Construction et maintenance de la TPLD

4.1 Construction et plongement dynamique de l'arbre

Nous donnons ici une idée de l'algorithme d'insertion d'un service déclaré sous la forme ($clef$=c, $valeur$=v). Le serveur envoie une requête d'insertion de son service à un nœud quelconque de l'arbre (dont il obtient la référence par un mécanisme extérieur). La requête est ensuite routée selon la clef c. Chaque nœud, sur réception de la requête, applique l'algorithme suivant, en distinguant quatre cas possibles:

c est égal à l'id du nœud local : c est donc déjà présente dans l'arbre, le nœud local

stocke v.

c est préfixé par l'id du nœud local : le nœud cherche alors parmi ses enfants un nœud dont l'id partage avec c plus de caractères que lui-même. Si un tel nœud existe, la requête lui est transmise, sinon, c donne lieu à la création d'un nouveau nœud fils du nœud local.

l'id du nœud local est préfixé par c : si l'id du père du nœud local préfixe c ou est égal à c, la requête doit remonter dans l'arbre et est transmise au père du nœud local. Sinon, c donne lieu à la création d'un nœud entre le nœud local et son père.

Défaut : si le nœud local n'est pas la racine et que l'identifiant de son père est "égal à" ou "préfixe" le préfixe commun de c et du nœud local, la requête est transmise au père du nœud local. Sinon, le nœud local et c sont des clefs *s*œurs mais leur père doit être de même créé. Deux nœuds sont alors créés, un nœud d'id c et le père du nœud local et du nouveau nœud dont l'id est leur plus long préfixe (éventuellement la chaîne vide).

Nous discutons ici brièvement comment plonger les nœuds logiques sur les pairs. Une approche simple est d'utiliser un répertoire central supposé stable auquel chaque machine se déclare et à laquelle on fait appel lors de la création d'un nœud logique pour l'héberger. Pour une architecture totalement décentralisée, une solution est de structurer le réseau physique avec une DHT et d'utiliser la fonction de hachage de la DHT pour placer les nœuds logiques en appliquant la fonction de hachage de la DHT sur les identifiants des nœuds de l'arbre. Un problème de répartition de la charge apparaît. En effet, l'utilisation d'une DHT garantira la distribution probabiliste uniforme des nœuds sur les pairs. Cependant, la charge de travail sera également distribuée sous deux conditions habituellement admises dans les DHT. La charge des nœuds logiques d'une part et la capacité des pairs doivent être homogènes. La charge d'un nœud dépend de la popularité des services qu'ils stockent et de sa hauteur dans l'arbre. En effet, plus un nœud sera proche de la racine, plus la proportion de requête qu'il devra router sera importante. Nous ne traiterons pas ici spécifiquement ce sujet et considérons la répartition de la charge effectuée au niveau de la DHT qui a donné lieu à une importante littérature [8, 10]. Adapter le facteur de réplication localement en fontion de la charge et répartir les requêtes sur chaque réplica est également une solution.

Le pseudo-code de gauche de l'algorithme 3.1 détaille le traitement d'une requête d'insertion. La fonction **COMMONPREFIX** retourne le plus long préfixe de deux chaînes. La fonction **NEWNODE** crée un nouveau nœud logique. La fonction **GETPEER** appelle le mécanisme de plongement (DHT par exemple) et retourne la référence d'un pair choisi par le mécanisme de plongement. La requête `hostReq` est envoyée au pair choisi pour héberger un nœud logique nouvellement créé. Les requêtes `updateChild` et `addChild` sont envoyées aux nœuds concernés par un changement dans les références vers leurs fils. Le code exécuté sur réception de ces messages n'est pas détaillé car trivial.

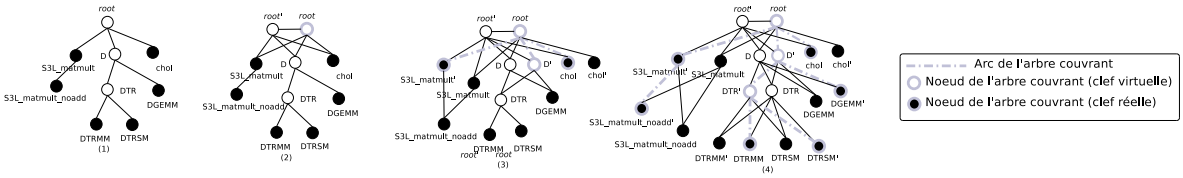
4.2 Tolérance aux pannes et prise en compte de la localité physique

Le routage logique est maintenant fixé. Pour faire face à la dynamicité du réseau, nous adoptons un mécanisme de réplications de l'arbre pour éviter la perte de nœuds ou de liens rendant le routage impossible. Le facteur de réplication k, indique le nombre de pairs distincts sur lesquels chaque nœud doit être présent, comme illustré sur la figure 1 pour $k = 2$. Pour

prendre en compte la localité physique dans l'arbre, on cherche à minimiser les temps de communication dans l'arbre répliqué. Nous devons prendre en compte différentes contraintes. Dans notre arbre, chaque lien a une sémantique dont une instance au moins doit être gardée dans l'arbre, ce qui rend le problème différent d'un problème classique d'arbre couvrant. De plus, pour garder une route *optimale* dans le sens **physique** du terme sur chaque nœud vers chaque nœud de l'arbre, il est nécessaire de garder une table de routage de taille linéaire en la taille de l'arbre non répliqué, ce qui serait néfaste au passage à l'échelle. L'optimalité ne pouvant être atteinte sans cette condition, nous optons pour une solution raisonnable basée sur une heuristique gloutonne. Chaque nœud va déterminer pour chacun de ses fils logique le réplica qui minimise à un instant donné le temps de communication avec lui. Cette heuristique est complètement intégrée à l'algorithme de réplication et ne modifie ni la complexité de l'algorithme ni la complexité de la taille de la table de routage.

Le processus de réplication avec adaptation à la topologie physique, décrit par le pseudo-code de droite de l'algorithme 3.1 et qui est illustré par la figure ci-dessous est initié périodiquement par l'une des racines de l'arbre (sur la figure, il n'y en a qu'une (1)). Les racines de l'arbre et seulement les racines forment un réseau complet, chaque racine connait donc l'ensemble des racines. Un algorithme simple d'exclusion mutuel prévient l'initialisation de l'algorithme par plusieurs racines en même temps. La racine élue commence par déterminer le nombre de racines joignables, mettons k'. La racine lance donc la réplication d'elle-même $k - k'$ fois en obtenant $k - k'$ pairs et en envoyant à chacun de ces pairs sa propre structure de nœud logique. Une fois la racine répliquée, la racine initie la réplication dans l'arbre en s'envoyant à lui-même une requête `scanReq` (2).

Sur réception d'une requête `scanReq`, un nœud exécute le pseudo-code détaillé par la partie commune à tous les nœuds du pseudo-code de droite de l'algorithme 3.1. Sur réception, donc, un nœud traite ses fils un par un. Pour chacun d'eux, il teste le nombre de réplicas joignables et demande à l'un d'eux d'envoyer sa structure de nœud logique vers un nombre de pairs nécessaires pour atteindre k via la requête `replicationReq`. Il détermine ensuite parmi les réplicas de ce fils le pair qui minimise le temps de communication avec lui (par la fonction **GETBESTREPLICA**) et envoie à l'élu une requête `scanReq` afin d'initier la réplication dans le sous-arbre de ce fils logique. Ainsi, déterminé le meilleur réplica localement pour chaque fils a deux fonctions: déterminer le pair/réplica qui représentera ce fils dans l'arbre couvrant (et qui sera ensuite utilisé pour router les requêtes) et d'élire le pair/réplica qui lance la réplication dans le sous-arbre de ce fils. Cet algorithme effectuant la réplication en parallèle dans les différentes branches de l'arbre (3, 4), possède une latence logarithmique en la taille de l'arbre.



5 Requêtes de découverte

Un client envoie sa requête à un nœud quelconque dans l'arbre. Comme l'illustre la figure 2(a), la requête est ensuite routée dans l'arbre de façon simple en remontant tant que l'id du nœud rencontré ne préfixe pas la clef recherchée. Puis la requête redescend jusqu'à trouver le nœud

dont l'id est la clef recherchée auquel cas ce nœud renvoie les valeurs des services qu'il stocke ou qu'elle retombe sur un nœud dont l'id ne préfixe plus la clef, auquel cas on répond négativement au client.

Le traitement d'une requête sur une chaîne partielle se fait en deux phases illustrées sur la figure 2(b). Considérons la requête `DTR*`. On route d'abord normalement selon `DTR` jusqu'à trouver le nœud avec l'id la plus petite préfixé par `DTR`, éventuellement `DTR` seul. Les clefs recherchées se trouvent dans le sous-arbre de ce nœud. Il nous reste à inonder le sous-arbre avec une latence logarithmique en la taille de l'arbre, grâce au parallélisme. Chaque nœud envoie ses réponses au client. Le traitement d'une requête sur un intervalle sera très similaire, en utilisant le préfixe des deux bornes et en appliquant le processus précédent sur ce préfixe mais sur chaque nœud, en limitant la diffusion aux branches de son sous-arbre concernées potentiellement par la requête.

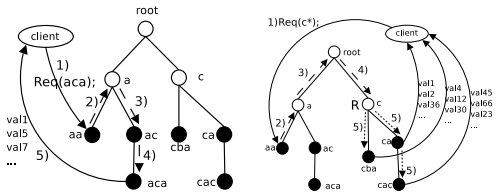


Figure 2: Découverte (gauche: requête simple, droite:autocomplétion). Le client envoie sa requête à un nœud logique quelconque (1). La requête est routée (2,3,4). Les valeurs sont retournées au client(5).

Notre architecture n'est pas **naturellement** multicritère. Nous créons un arbre par critère, où chaque couple ($clef$, $valeur$) est enregistré dans l'arbre correspondant à l'attribut décrit par $clef$. Mettre tous les attributs dans le même arbre entraînerait des comportements indésirables si un service s'appelle comme un système par exemple. De même, pour une interrogation sur d critères, le client lance pour chacun des attributs qui l'intéresse une requête dans l'arbre correspondant. Chaque arbre traite la requête en parallèle résultant encore en une latence logarithmique en la taille agrégée des arbres. Par exemple, la requête `{DTRSM, Linux*, PowerPC*, *}` donnera lieu à trois requêtes traitées en parallèle dans l'arbre des noms, l'arbre des systèmes et l'arbre des processeurs. Sur réception des réponses, le client fait l'intersection des valeurs reçues pour ne garder que les services qui respectent les trois clefs spécifiées.

6 Analyse

Nous analysons ici rapidement les complexités de la TPLD pour les différentes métriques des réseaux pair-à-pair et ses avantages/inconvénients par rapport aux autres approches basées sur des arbres de préfixes.

Considérons un arbre de plus long préfixe de taille n. Appelons A l'alphabet qui servirait à générer les clefs qui seront stockées dans l'arbre. En considérant la taille des clefs bornée par T_{max}, ce qui semble réaliste, c'est pourquoi nous ne nous lancerons pas ici dans des calculs poussés de complexité, le nombre de sauts au pire est bornée par $2 \times T_{max}$. Pour les requêtes nécessitant l'autocomplétion, le nombre de messages nécessaires pour arriver à la racine du sous-arbre concerné par la requête est de la même façon bornée par T_{max}. Ensuite, on ne peut éviter de passer par tous les nœuds touchés par l'intervalle considéré et obtenir un nombre

de messages linéaire en la taille du sous-arbre. En revanche, grâce au parallélisme entre les branches, la latence de ce parcours est encore bornée par la profondeur de l'arbre et donc par T_{max}. De même, si l'on considère A fini, chaque nœud ayant par construction une entrée par caractère dans la table de routage, (k en comptant la réplication, la taille de la table de routage est toujours bornée par $|A|$. Dans la pratique, cela signifie que la table de routage peut-être allouée statiquement, par exemple sous la forme d'un tableau de $|A|$ cases. En conséquence, la décision local de routage sur chaque nœud peut se faire en temps constant, en examinant la case correspondant au prochain caractère de la clef recherchée.

Critère	**Skip Graphs**	**Prefix Hash Tree**	**P-Grid**	**TPLD**
Nb de messages pour une recherche simple	$O(\log(n))$	$O(\log(D)\log(N))$	$O(log(\Pi))$	$O(T_{max})$
Nb de messages pour un intervalle	$O(m\log(n))$	$O(o)$	$O(\Pi_R)$	$O(l)$
Latence pour un intervalle	$O(\log(n))$	$O(D)$	$O(log(\Pi))$	$O(T_{max})$
Tolérance aux pannes	réparation	basée sur une DHT	réplication	réplication
Localité	-	-	-	gloutonne

Le tableau ci-dessus résume les complexités et fonctionnalités de la TPLD et des autres approches basées sur des arbres lexicographiques. Bien que l'on puisse montrer différentes propriétés intéressantes de Skip graph concernant la tolérance aux pannes, le nombre de messages nécessaires au traitement d'une requête sur un intervalle est en $O(m\log(n))$ où n est la taille totale de l'arbre et m le nombre de nœuds concerné par l'intervalle. Prefix Hash Tree construit un arbre lexicographique de l'ensemble des clefs potentiels dont chaque feuille gère l'ensemble des clefs de la branche et est affecté à un nœud de la DHT sous-jacente. Par l'utilisation systématique de la DHT, la complexité d'une requête simple est ainsi en $O(\log(D)\times\log(N))$, chaque saut dans l'arbre de profondeur bornée par la taille maximum des clefs D nécessitant une recherche dans la DHT de taille N, on note o la taille de la sortie engendrée par une requête sur un intervalle. P-Grid construit un arbre de préfixe statiquement sur l'espace des clefs tout entier, en associant chaque pair avec une partie de l'espace, dont la taille est notée Π. Π_R note la taille d'un intervalle R. Nodewiz adopte une approche similaire multi-critères mais ne se place pas dans un environnement dynamique, considérant les nœuds stables, hypothèse qui semble peu réaliste. Notre architecture construit un arbre lexicographique réduit, ou *de plus long préfixe* qui reflète mieux l'ensemble des services effectivement déclarées et évitent des sauts inutiles, l dénote le nombre de nœuds du sous-arbre touché par une requête sur un intervalle. Enfin, seulement notre architecture s'adapte au réseau sous-jacent, en utilisant la topologie de l'arbre.

7 Simulation

Un simulateur a été développé en Java, implémentant la création et l'interrogation de l'arbre. Nous avons utilisé des jeux de données réels contenant 735 noms de services, 129 noms de processeurs, 189 noms de systèmes d'exploitation ainsi que près de 4000 noms de machines. Nous avons d'abord testé le nombre de sauts logiques moyen nécessaire à l'insertion d'un nouveau service, en prenant les clefs à insérer aléatoirement dans chacun des jeux d'attributs ainsi que dans un ensemble de 10000 mots de taille maximum 20 générés aléatoirement dans un alphabet de 65 caractères. La courbe (a) de la figure 3 donne le résultat de cette simulation pour chacun des jeux de données. Le comportement est clairement logarithmique. Nous avons de même testé le nombre de sauts logiques nécessaires au routage d'une requête d'insertion

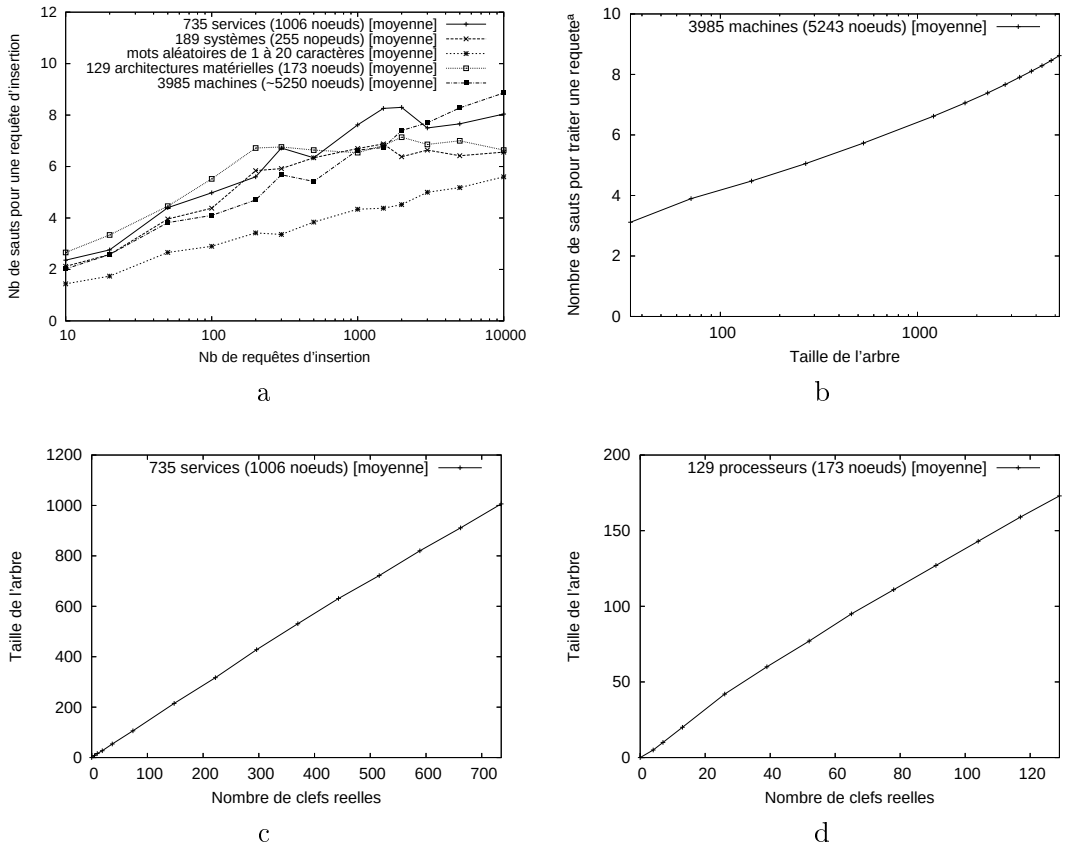


Figure 3: (a) et (b) : nombre moyen de sauts nécessaire au routage. (c) et (d) : proportionnalité entre nombre de clefs réels et taille de l'arbre.

dans un arbre de taille borné par le nombre total de nœuds (clefs réels et virtuels) résultant de l'insertion de tous les attributs du jeu de données. Le résultat de l'expérience nous est donné par la courbe (b) de la figure 3 qui exhibe de même un comportement logarithmique, ici sur le jeu d'attributs des machines (mais le comportement est similaire pour tous les jeux de données.)

Enfin, les courbes (c) et (d) de la figure 3 montrent comment évolue la taille de l'arbre en fonction du nombre de clefs réelles déclarées. Les deux courbes sont clairement proportionnelles. L'ensemble des expériences montre que la part de clefs virtuelles est en moyenne de 30 % (avec un écart type de 2,4%).

8 Conclusion

Nous avons décrit une nouvelle architecture pour une découverte flexible des services permettant l'autocomplétion des chaînes de recherche partielles. Notre outil est tolérant aux pannes et s'adapte de manière gloutonne à la topologie du réseau sous-jacent. Les complexités ainsi que nos expériences de simulation montrent des caractéristiques et des résultats intéressants. Nous développons actuellement un autre mécanisme de tolérance aux pannes fondé sur la réparation des arbres suite à la perte d'une branche, alors que le mécanisme de réplica-

tion présenté ici est préventif. Nous étudions actuellement le développement d'un prototype de la TPLD permettant d'observer son comportement sur une plate-forme large échelle afin d'ajuster certains paramètres comme le facteur de réplication.

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

[7] 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.

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

[10] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *IPTPS*, pages 131–140, 2004.

[11] 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.

[12] 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, july 2004.

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

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

[15] 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.

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

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

[18] Z. Xu, R. Min, and Y. Hu. HIERAS: A DHT Based Hierarchical P2P Routing Algorithm. In *ICPP*, pages 187–, 2003.

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

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