HAL
open science

P2P4GS : une spécification de grille pair-à-pair de services auto-gérés

Bassirou Gueye, Olivier Flauzac, Ibrahima Niang, Cyril Rabat

To cite this version:

Bassirou Gueye, Olivier Flauzac, Ibrahima Niang, Cyril Rabat. P2P4GS : une spécification de grille pair-à-pair de services auto-gérés. Revue Africaine de Recherche en Informatique et Mathématiques Appliquées, 2014, Volume 17 - 2014 - Special issue CARI'12, pp.155-175. 10.46298/arima.1972 . hal-01300053

HAL Id: hal-01300053

https://inria.hal.science/hal-01300053v1

Submitted on 8 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Special issue CARI'12

P2P4GS : une spécification de grille pair-à-pair de services auto-gérés

Gueye Bassirou*# — Flauzac Olivier* — Niang Ibrahima# — Rabat Cyril*

* CReSTIC, UFR Sciences Exactes et Naturelles
Université de Reims Champagne Ardenne
FRANCE
bassirou.gueye@etudiant.univ-reims.fr, {olivier.flauzac, cyril.rabat}@univ-reims.fr

LID, Département de Mathématiques et d'Informatique
Université Cheikh Anta Diop de Dakar
SENEGAL
bassirou.gueye@ucad.edu.sn, iniang@ucad.sn

RÉSUMÉ. Les grilles basées sur des architectures pair-à-pair (P2P) ont été utilisées soit dans le cadre du stockage et du partage de données; soit dans le cadre de calculs demandant beaucoup de ressouces. En ce qui concerne la mise en place de services, et plus particulièrement de services de grilles, les solutions proposées sont basées sur des grilles hiérarchiques qui présentent un fort degré de centralisation, tant en ce qui concerne le déploiement des services que la recherche et l'invocation.
Dans ce papier, nous proposons une spécification originale d'une "grille pair-à-pair de services auto-gérés" nommée P2P4GS. L'objectif est de concevoir une solution auto-adaptative permettant le déploiement et l'invocation de services tout en respectant le paradigme des réseaux pair-à-pair. Le déploiement, comme l'invocation est totalement délégués à la plateforme et se fait de manière transparente pour l'utilisateur. De plus, cette spécification est générique c'est à dire non liée à une architecture pair-à-pair particulière ou à un protocole de gestion de services défini à l'avance. Une fois la spécification présentée, nous proposons une étude de complexités algorithmiques des primitives de déploiement et de localisation de services définies dans P2P4GS en les plongeant sur les topologies classiques de la pile P2P à savoir l'anneau et l'arbre. Les performances obtenues sont satisfaisantes pour ces différentes topologies.

ABSTRACT. The grid-based peer-to-peer architectures were used either for storage and data sharing or computing. So far, the proposed solutions with respect to grid services are based on hierarchical topologies, which present a high degree of centralization. The main issue of this centralization is the unified management of resources and the difficult to react rapidly against failure and faults that can affect grid users.
In this paper, we propose a original specification, called P2P4GS, that enables selfmanaged service of peer-to-peer grid. Therefore, we design a self-adaptive solution for services deployment and invocation which take account the paradigm of peer-to-peer services. Furthermore, the deployment, and invocation are completely delegated to the platform and are done a transparent manner with respect to the end user. We propose a generic specification that is not related to a particular peer-to-peer architecture or a management protocol services defined in advance. On the other hand, we propose a study of algorithmic complexities of deployment and service localization primitives in P2P4GS by immersing them on the classical topologies of P2P stack ie the ring and tree. The obtained performances are satisfactory for these different topologies.

MOTS-CLÉS : Réseaux pair-à-pair, Grilles de calcul, Services Web

KEYWORDS : Peer to peer network, Grid computing, Web services.

1. Introduction

Les grilles de calcul (Grid Computing) [5, 3, 8, 14] sont une technologie en pleine expansion dont le but est d'offrir aux organisations virtuelles, ainsi qu'à la communauté scientifique des ressources informatiques virtuellement illimitées. Par le biais de ces grilles, les utilisateurs ont pu avoir accès à des ressources de calcul ou de stockage de très grande puissance qu'ils n'auraient pu exploiter autrement. On peut notamment citer, dans le cadre du calcul, le projet SETI@home [4] qui a permis de fédérer les puissances de calcul de plusieurs dizaines de milliers d'ordinateurs à travers le monde. Parallèlement, des projets de stockage de données comme STORAGE@Home [4] ont permis de fédérer les capacités de stockage de plusieurs dizaine de milliers de machines.
L'apparition des Services Web [11] a fourni un cadre qui a initié l'alignement de ces deux technologies et qui a été à l'origine des grilles de services [9, 10]. En effet, les grilles de services représentent le résultat de recherche établi par le Globus Grid Forum (GGF)[1] et ayant abouti à l'Open Grid Service Infrastructure (OGSI) [9] et l'Open Grid Service Architecture (OGSA) [10]. Ces grilles ont permis de distribuer les traitements pour obtenir une utilisation optimale des ressources. Un effort particulier de normalisation est actuellement fait afin d'apporter, comme dans le cas des Services Web, toutes les ressources et les moyens nécessaires au développement d'applications dans ces grilles de services.
Cependant, ces grilles sont basées sur des architectures hiérarchiques fortement centralisées [3, 14]. Cette centralisation implique une gestion unifiée des ressources, mais aussi des difficultés à réagir vis à vis des pannes et des fautes qui impactent la communauté. En effet, la plupart des solutions de tolérance aux fautes se limitent aux essais répétés et au checkpointing global [5, 6].

Dans cet article, nous présentons P2P4GS, une nouvelle spécification de gestion des services dans un environnement de grilles de calcul basé sur une architecture pair-à-pair. Cette spécification présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme d'exécution de services. Ainsi, contrairement à des solutions de type Pastry [7], nous proposons de séparer la couche de gestion de la grille pair-à-pair, des couches de localisation et d'exécution de services. Le déploiement comme l'invocation est totalement délégué à la plate-forme et se fait de manière transparente pour l'utilisateur.
Ce document constitue une extension de nos premiers travaux définis dans [2], en offrant une meilleure description ainsi qu'une étude formelle de la spécification. En effet, outre l'architecture et les différents aspects de la spécification, nous avons décrit les algorithmes de déploiement et de localisation de services définis dans P2P4GS ainsi qu'une étude de performances de ces derniers. Notre objectif est de proposer une spécification qui se voudra la plus générique possible, non liée à une architecture pair-à-pair particulière (anneau, arbre, ...) ou à un protocole de gestion de services défini à l'avance (Pastry, Gnutella, ...). Il serait par conséquent très utile voire fondamental d'étudier l'impact de nos primitives sur les topologies classiques existantes à savoir l'anneau et l'arbre.

Le reste du document est organisé comme suit. Dans un premier temps, nous présentons dans la section 2 l'originalité de notre solution. Ensuite, après un examen de la littérature dans la section 3, nous présenterons le modèle que nous utilisons dans la section 4, ainsi que les différents aspects de la spécification P2P4GS dans la section 5. Puis, la section 6 propose une étude de complexités algorithmiques des primitives de déploiement

1. http ://www.ggf.org/

et de localisation de services définies dans P2P4GS. Enfin, la section 7 conclut et donne quelques perspectives futures de ce travail.

2. Contribution

La spécification P2P4GS que nous proposons présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme d'exécution de services. En effet, contrairement à des solutions de type Pastry [7], notre approche permet de séparer la couche de gestion de la grille pair-à-pair des couches de localisation et d'exécution de services. Cette spécification est générique et tend donc à être applicable sur toute architecture pair-à-pair.
Comme nous venons de le préciser, nous ne nous focalisons pas sur la gestion de l'architecture de la plate-forme pair-à-pair, mais sur l'aspect implantation, déploiement et exécution des requêtes. Nous faisons dans le cadre de la modélisation le choix de ne pas nous aligner sur une architecture typique d'exécution ; mais nous spécifions les opérations de manière détachée. Il sera par conséquent tout à fait possible d'associer cette spécification à différents environnements de contrôle, d'échange et d'exécution de services : .Net, J2EE, Web Services SOAP, Corba, etc. Ainsi, toute combinaison de systèmes pair-à-pair respectant les contraintes de notre modèle pourra être composée avec toute plate-forme d'exécution de services.
Enfin, nous avons proposé des algorithmes pour le déploiement et pour la localisation de services dans le système P2P4GS. Les performances ainsi obtenues à partir de étude de complexités algorithmiques de ces primitives sont d'une manière générale très satisfaisantes tant sur la topologie anneau orienté ou non orienté que sur l'arbre.

3. Travaux connexes

L'exécution de code distant a été définie de différentes façons. L'appel distant de code a donné lieu au schéma général des RPC(*Remote Procedure Call*)[2]. La spécification *RPC* exprime les échanges entre un consommateur d'informations (le client) et la ressource (le serveur). On définit ainsi la possibilité du client à effectuer des requêtes auprès d'un service détenu par le serveur, autrement dit d'invoquer des services. Différentes implémentations qui mettent en œuvre différents langages, protocoles ou systèmes ont été proposées, dont celles :

– basées sur des bibliothèques systèmes : ONC RPC[3] ;

– basées sur des objets à distance : JAVA RMI, Corba ;

– basées sur l'utilisation de composants : J2EE, Net - Mono ;

– basées sur les Services Web [11] : XML-RPC, SOAP.

Le concept de RPC a été transféré dans les grilles, soit à partir des bibliothèques permettant la mise en place de solutions techniques, comme Globus [5], soit à partir des solutions directement conçues pour assurer le GRID RPC [13, 14]. Néanmoins, les solutions proposées sont hiérarchiques et nécessitent des points de centralisation de la connaissance.

2. RFC707, RFC 1057
3. RFC 1790

C'est le cas par exemple de Ninf-G [13] et DIET [14] dans lesquelles les capacités d'exécution s'enregistrent auprès d'un point de centralisation.

Comme dans le cas des exécutions sur le Web, des solutions d'implantation de services dans les grilles [9, 10] et de découverte de ressources [15, 16] dans de tels environnements ont été proposées. Cependant, l'ensemble de ces propositions est basé sur des solutions d'architectures hiérarchiques qui présentent un degré de dynamicité très réduit et dont les services d'infrastructure, comme le déploiement et la localisation de services, sont eux même centralisés. En effet, les approches basées sur les systèmes centralisées ou hiérarchisées souffrent d'un point de défaillance unique, de goulots d'étranglement dans les systèmes hautement dynamiques, et de manque d'évolutivité dans les environnements hautement distribués [22].

La convergence des grilles informatiques et des systemes pair-à-pair semblent être de plus en plus naturelle. En effet, Ian Fooster et al. [1] precisent que les grilles et les systèmes pair-à-pair ont tendance à converger vers le même objectif, tout en partant d'un point de départ différent. L'objectif de l'intégration de ces deux paradigmes est de pallier les inconvénients des systèmes traditionnels de grilles [20, 21, 22].

Soulignons cependant que la plupart des travaux autour des grilles pair-à-pair se limitent sur la découverte de ressources [16, 19, 20, 21, 22, 23, 24] et se distinguent généralement dans le type d'algorithme de recherche utilisé. C'est le cas par exemple de Gnutella [17] qui se base sur une recherche distribuée, tandis que Chord [18] et Pastry [7] utilisent des mécanismes de routage basés sur le principe des tables de hachages distribuées (DHT).

4. Vue d'ensemble de la spécification *P2P4GS*

4.1. Modélisation

Afin de décrire notre spécification, nous devons d'abord définir notre modèle d'application. Le modèle ainsi utilisé dans cet article est basé sur 3 composants principaux : le réseau, les nœuds qui composent le réseau et les services.

4.1.1. Le réseau

Le réseau que nous considérons est le *réseau overlay*. Il s'agit en fait, non pas du réseau physique d'interconnexion, mais du réseau logique qui est mis à notre disposition par la plate-forme pair-à-pair.
Afin d'être exploitable dans notre spécification, on modélise le réseau sous forme d'un graphe non orienté $G = (V,E)$ où V est l'ensemble de nœuds et E l'ensemble des liens de communications. Le réseau offre au minimum les primitives de communication suivantes :

1) émission d'un message :

- envoi d'un message à un voisin `send(Neigh, message)`
- envoi d'un message à un nœud de la communauté `route(id,message)`

2) réception d'un message :

- réception bloquante d'un message `receive()`

- réception non bloquante d'un message `async_receive(callback)`, où `callback` est une procédure à exécuter à la réception d'un message.

REMARQUE. — Nous pouvons ajouter la primitive diffusion (envoie d'un message à l'ensemble de la communauté), qui ne sera qu'une utilisation des primitives précédentes.

Ces définitions permettent la prise en compte des différentes spécificités du réseau :

– éléments de sécurité comme des *firewalls* qui limitent la possibilité de communication des nœuds ;

– éléments de configuration et d'implémentation du réseau comme du NAT.

REMARQUE. — On pourra tout de même remarquer que nous nous attachons uniquement à la communication et non à la connexion : deux nœuds qui ne peuvent communiquer de manière directe (tous deux derrière des *firewalls*) peuvent être considérés comme voisins s'ils exploitent une file d'échange partagée, un *MOM* (Middleware On Message).

4.1.2. Les nœuds

Chacun des nœuds détient un identifiant unique dans le réseau pair-à-pair. Les nœuds sont en charge de la gestion locale du réseau et assurent collectivement les tâches décrites précédemment. Outre ces tâches, ils garantissent le réceptacle des services, c'est à dire la plate-forme d'exécution. Les principales charges de cette plate-forme sont :

– la gestion du déploiement ;

– le cycle de vie des services ;

– la gestion des requêtes et des exécutions.

Les nœuds détiennent de plus une table des services, qui, comme nous le verrons par la suite, répertorie au minimum l'ensemble des services détenus localement, et au maximum l'ensemble de tous les services présents sur la grille.

4.1.3. Les services

Les services sont des objets intégrables aux plate-formes d'exécution détenues en chaque nœud. Un service est caractérisé par :

– la plate-forme d'exécution pour laquelle il a été connçu ;

– les ressources nécessaires à son exécution (ressources de calcul, ressources de données, ressources de connexion ...) ;

– les données, format et contenu nécessaires lors de l'invocation du service ;

– le format et les contraintes des données résultats.

4.2. Architecture de la spécification *P2P4GS*

Nous définissons un modèle d'architecture en couche (Figure 1) dans laquelle nous allons intégrer notre middleware. Cette modularisation nous permet d'hériter des services offerts par les couches basses.

Ainsi, la couche TCP/IP qui représente la couche réelle ou physique pourra offrir des fonctions ou services de sécurité, de fiabilité, etc [12].

Le couche P2P est une couche logique qui exploite un protocole de réseau pair-à-pair (Pastry, Gnuttela, etc.) reposant sur une topologie (anneau, arbre, etc.).

La couche middleware représente notre spécification P2P4GS. Elle est constituée de deux sous couches :

– Une sous couche directement liée à la topologie P2P sous-jacente et qui permet de faire abstraction à cette dernière.

– Une sous couche qui propose un ensemble de primitives (« deploy », « lookup », « invoke », « exec », « save ») décrit dans la section 5 et offrant toutes les solutions de gestion de services dans P2P4GS, qui vont de la création ou implantation d'un service jusqu'à sa consommation.

Ces primitives sont exploitées par les différentes plate-formes auxquelles elles interagissent afin d'offrir des services (service discovery, service runtime, service composition, etc.) à la couche application.

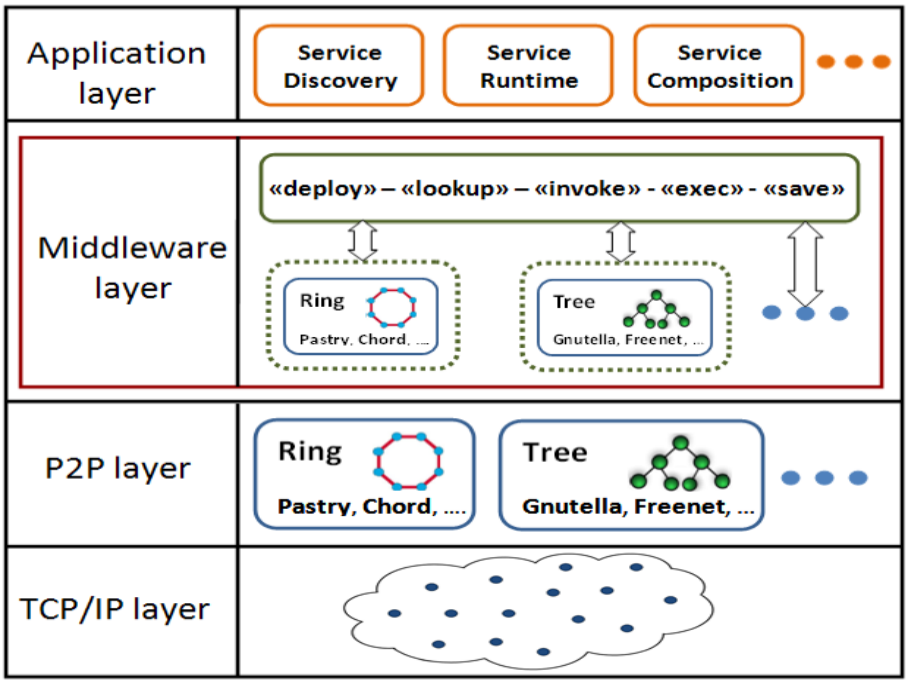

Figure 1 – Architecture de la spécification P2P4GS

5. Spécification des services en environnement P2P

Nous définissons dans ce qui suit, l'ensemble des primitives et opérations spécifiques à l'exécution de services sur notre grille pair-à-pair.

5.1. Déploiement de service : `deploy`

Le déploiement d'un service nécessite une première phase de détection des nœuds de la plate-forme en mesure d'héberger et d'exécuter le service. Un sous ensemble de nœuds de la plate-forme sera donc candidat à l'hébergement du service. Parmi ces nœuds il faudra donc, en appliquant une stratégie particulière, désigner celui qui va héberger le nouveau service.

Nous proposons dans ce travail trois stratégies de placement :

1) une *stratégie aléatoire* qui consiste à tirer au sort le nœud sur lequel sera déployé le nouveau service ;

2) une *stratégie équilibrée* qui consiste à collecter les statistiques des différents nœuds et à tenter d'équilibrer la charge entre ces nœuds candidats ;

3) une *stratégie premier nœud* qui consiste à déployer le nouveau service sur le premier nœud détecté capable de supporter son exécution.

REMARQUE. — La première stratégie implique de connaitre l'ensemble des noeuds candidats au déploiement ; la seconde implique de connaitre l'ensemble des nœuds candidats et des informations sur chacun d'eux ; et la troisième limite la connaissance nécessaire, mais peut clairement mener à une surcharge.

5.2. Localisation des services : `lookup`

La localisation des services est la première étape de la chaîne d'exécution d'un service. Chaque nœud de la grille pair-à-pair détient un registre de services. La fonction *lookup* est considéré comme un service local. En effet, il est disponible sur chacun des noeuds de la communauté ; mais ne peut pas être invoqué depuis une machine distante.
A la réception d'une requête d'exécution de service par un noeud de la plate-forme, ce dernier fait appel à ce service de localisation qui exécute les opérations suivantes :

– si le service est présent sur le nœud, celui-ci est invoqué ;

– si le service n'est pas présent sur le nœud :

- si le nœud connaît la localisation du service, il route la requête vers le nœud qui l'héberge ;

- si le nœud ne connaît pas la localisation du service, il fait suivre la requête vers ses voisins.

REMARQUE. — 1. Le service requis peut être absent de la grille, soit suite à une panne, ou soit qu'il n'est pas encore été déployé. Dans ce cas, on peut utiliser un algorithme de type `PIF (propagation of information and Feedback)`, qui retourne au point

d'invocation un code d'erreur qui sera transmis à l'émetteur de la requête.

REMARQUE. — 2. Comme nous l'avons précisé précédemment, notre spécification n'est pas liée à une architecture particulière. Dans le cas d'une architecture basée sur un anneau orienté ou une chaîne, la procédure `forward` ne permet d'exploiter la communauté que de manière sérielle. Par contre, dans le cadre d'un anneau non orienté ou d'un arbre, la procédure d'exploration initiée par `forward` se fera en parallèle.

5.3. Invocation d'un service : `invoke`

L'invocation d'un service peut être initié sur n'importe quel nœud de la communauté. Cette invocation peut se faire soit à partir d'un nœud de la grille, soit par un nœud extérieur à la communauté. Dans ce second cas, tout nœud de la communauté peut être le «point d'invocation», qui sera en charge de faire l'invocation sur la communauté, de récupérer le résultat et de retourner ce dernier, ou une erreur, à la source de la requête.

5.4. Exécution de service et retour des résultats : `exec`

Une fois que la requête a atteint le nœud qui détient le service, ce dernier l'exécute. La première étape consiste à identifier les paramètres de l'invocation avec ceux de l'appel. Cette exécution peut provoquer soit la génération d'un résultat, soit celle d'une erreur. Le résultat ainsi produit est routé vers le «point d'invocation».

5.5. Enregistrement de service : `save`

Il est possible d'exploiter la plate-forme par l'intermédiaire de recherche sans mémoire des services à chaque invocation. Comme nous l'avons précisé précédemment, nous pouvons augmenter les performances de notre plate-forme en exploitant un registre de services. Nous envisageons la construction de ce registre de services selon deux principales stratégies :

1) la diffusion au déploiement : dans cette première stratégie, la diffusion de l'information sur la localisation du service se fait sous la forme d'une innondation auprès de tous les nœuds de la plate-forme ;

2) la diffusion à l'appel : dans cette seconde stratégie, l'information de localisation est diffusée lors de la réponse après invocation. Tous les sites relayant la réponse vers le point d'invocation seront alors informés de la localisation du service.

REMARQUE. — Afin d'éviter la surcharge des mémoires de sites par le registre de service, il est possible de répartir ce registre sur plusieurs sites consécutifs. Ce qui aura l'intérêt de réduire la charge de chaque site, et de limiter la perte d'information en cas de disparition de site.

6. Etude de complexités algorithmiques des primitives de déploiement et de localisation de services dans P2P4GS

Dans cette section, nous présentons une étude algorithmique des primitives de déploiement (« deploy ») et de localisation (« lookup ») de services définis dans la spécification *P2P4GS*. Pour ce faire, nous décrivons tout d'abord le formalisme de notation que nous utilisons. Par la suite, nous introduisons les algorithmes de ces primitives en faisant une description préalable de leurs principes d'éxécution. Enfin, une étude de performances de ces dernières sera faite.

Soulignons que dans cette étude on a abordé ces deux primitives (`deploy` et `lookup`) vu qu'elles interviennent lors des premières phases du cycle de vie d'un service. En outre, le déploiement et surtout la localisation de ressources (plus généralement de services) constituent les deux problématiques les plus abordées dans la littérature. L'étude de complexités des autres primitives (`invoke`, `exec` et `save`) qui interviennent lors des phases d'exécution et de monitoring du cycle de vie d'un service seront abordées dans nos travaux futurs.

REMARQUE. — Dans cette étude, nous considérons les topologies classiques de la pile P2P à savoir l'anneau orienté, l'anneau non orienté et l'arbre. Rappelons que les topologies chaine et étoile contituent des cas particuliers de la topologie en arbre.

6.1. Formalisme de notation

Afin de définir nos algorithmes ainsi que leurs principes d'exécution décrits dans la section 6.2, nous adoptons le formalisme de notation suivant :

– *id* : identifiant du nœud ;

– *neigh* : voisinage du nœud ;

– *idWin* : identifiant du nœud candidat potentiel ;

– *nbSvcWin* : nombre de services du nœud candidat potentiel ;

– *idOwn* identifiant du nœud qui détient le service recherché ;

– *available* : variable booléenne qui prend la valeur *true* si le nœud est disponible ;

– *isAble*() : retourne vrai si le nœud est capable d'héberger le nouveau service ;

– requests : ensemble des requêtes en attentes sur le nœud en cours :

- Requêtes définies par les tuples (*idInit*, *idSvc*) et (*idInit*, *idSvc*, *idFather*) :

- *idInit* : identifiant du nœud initiateur ;

- *idSvc* : identifiant du Service Web ;

- *nbRcv* : nombre de réponses depuis les voisins ;

- *idFather* : identifiant du nœud depuis lequel on a reçu la requête ;

- *idFather* = *idInit* sur le nœud initiateur ;

- (*idInit*, *idSvc*) : unique dans requests ;

- requests.add(*idInit*, *idSvc*, *idFather*) :sauvegarde d'une nouvelle requête ;

- requests(*idInit*, *idSvc*).exists() : retourne vrai si la requête existe ;

- requests(*idInit*, *idSvc*).delete() : supprime la requête ;
- requests(*idInit*, *idSvc*).increase() : augmente de 1 le nombre de réponses ;
- requests(*idInit*, *idSvc*).nbRcv : retourne le nombre de réponses des voisins ;
- requests(*idInit*, *idSvc*).idFather : retourne l'identité du père ;

Messages :

- queryDeploy(*idInit*, *idWin*, *nbSvcWin*) : élection du nœud sur lequel sera déployé le nouveau service ;
- resultDeploy(*idInit*, *idWin*, *nbSvcWin*) : retourne l'identité du nœud candidat potentiel ; *idWin* = -1 si pas de nœud ;
- queryLookup(*idInit*, *idSvc*) : recherche du service ;
- resultLookup(*idInit*, *idOwn*, *idSvc*) : retourne l'identité du nœud qui posséde le service ; *idOwn* = -1 si pas de nœud ;

lookup(*idSvc*) :

- retourne *id*, si le service *idSvc* est présent localement ;
- retourne *idOwn*, si le service est présent dans la communauté ;
- retourne -1, si le service est absent dans la communauté.

6.2. P2P4GS : algorithmes de déploiement et de localisation de services

6.2.1. Déploiement d'un service dans P2P4GS

Nous décrivons dans cette section, la stratégie de déploiement équilibrée. Les stratégies "aléatoire" et "premier nœud" seront étudiées ultérieurement par le biais de simulations.

Pour déployer un service dans P2P4GS, il suffit de se connecter à un $nudpointd'entre$ qui se chargera d'initier le processus d'élection du nœud sur lequel sera déployé le nouveau service. Précisons que tout nœud dans notre l'architecture peut faire office de point d'entrée.

Principe d'exécution : il faut trouver le nœud capable d'héberger le nouveau service et ayant un plus petit nombre de services. Les étapes suivantes sont prévues :

1) Le nœud qui initie le processus doit être à l'état disponible (*available()*). En effet, si le nœud choisi comme point d'entrée n'est pas diponible, cela signifie qu'un processus d'élection sollicité par un autre utilisateur est en cours d'exécution. Ensuite, le nœud initiateur effectue l'opération (I) s'il est apte ($isAble()$, c'est à dire s'il respecte les contraintes en terme de CPU, RAM, plateforme d'exécution, etc.). L'opération (I) consiste à initialiser les paramètres idWin et nbSvcWin (voir les Agorithm 1, 2 et 3). Par la suite, selon le cas, le nœud initiateur effectue lâopération (II) qui consiste à envoyer la requête $queryDeploy()$ à :

- son voisin (*next*) s'il s'agit d'une topologie anneau orienté ;
- ses deux voisins (*next* et *prev*) pour une topologie anneau non orienté ;
- tout ses voisins (*neigh*) sauf à son père, dans le cas d'une topologie arbre.

2) Tout nœud (sauf le nœud initiateur) qui reçoit la requête *queryDeploy*() effectue l'opération (I).

3) L'opération (II) est effectuée par :

- cas de l'anneau orienté : tout nœud ayant reçu la requête *queryDeploy*() sauf le nœud initiateur ;

- cas de l'anneau non orienté : tout nœud ayant reçu la requête *queryDeploy*() au plus une fois. Si un nœud reçoit une deuxième fois une requête *queryDeploy*() (surement de son autre voisin *next* (respectivement *prev*)), il envoie une requête *resultDeploy*() à son voisin *prev* (respectivement *next*) ;

- cas de l'arbre : tout nœud ayant reçu la requête *queryDeploy*() et ayant au moins 2 voisins. Si un nœud n'a qu'un seul voisin (son père), il lui répond par un message *resultDeploy*() après avoir effectué l'opération (I).

4) Tout nœud (sauf le nœud de l'initiateur) qui reçoit un message *resultDeploy*() :

- retransmet le message à l'autre voisin, s'il a seulement 2 voisins ;

- effectue une opération (III) s'il a plus de 2 voisins. Cette opération consiste à attendre les réponses de ses autres nœuds fils et puis les traiter (mise à jour des paramètres *nbsvcWin* et *idWin*). Par la suite, le nœud envoie le résultat (*resultDeploy*()) à son père.

5) Le processus d'élection se termine lorsque le initiateur nœud reçoit un message *queryDeploy*() (cas de l'anneau orienté) ou un message resultDeploy () (cas de l'anneau non orienté ou de l'arbre). Soulignons que dans le cas de la réception d'un message de *resultDeploy*(), le nœud initiateur exécute d'abord l'opération (III).

6) Après exécution du processus d'élection, le nœud initiateur se remet à l'état disponible.

Algorithme 1: Stratégie de déploiement équilibré sur un anneau orienté

```
/* Initialisation                                                    */
if available = true then
    if (isAble()) then
        idWin ⟵ id
        nbSvcWin ⟵ nbSvc
    end
    send(next, queryDeploy(id, idWin, nbSvcWin))
    available ⟵ false
end

/* A la réception d'un message queryDeploy(idInit, idWin, nbSvcWin)
   sur le nœud i                                                     */
if (isAble() ∧((nbSvcWin = −1) ∨ (nbSvcWin > nbSvc)))then
    idWin ⟵ id
    nbSvcWin ⟵ nbSvc
end
if (id ≠ idInit) then
    send(next, queryDeploy(idInit, idWin, nbSvcWin))
else
    if (idWin ≠ -1) then
        /* deploiement possible sur le nœud d'identifiant idWin  */
    else
        /* deploiement impossible !                              */
    end
    available ⟵ true
end
```

Algorithme 2: Stratégie de déploiement équilibré sur un anneau non orienté

```
/* Initialisation                                                    */
if available = true then
    if (isAble()) then
        idWin ⟵ id
        nbSvcWin ⟵ nbSvc
    end
    send(next, queryDeploy(id, idWin, nbSvcWin))
    send(prev, queryDeploy(id, idWin, nbSvcWin))
    available ⟵ false
end
/* A la réception d'un message queryDeploy(idInit, idWin, nbSvcWin) sur le
   nœud i depuis le nœud j                                           */
if (isAble() ∧((nbSvcWin = −1) ∨ (nbSvcWin > nbSvc)))then
    idWin ⟵ id
    nbSvcWin ⟵ nbSvc
end
if ¬ requests(idInit, idSvc).exists() then
    requests.add(idInit, idSvc)
else
    requests(idInit, idSvc).increase()
end
if (request(idInit).nbRcv < 2) then
    if (j = prev) then
        send(next, queryDeploy(idInit, idWin, nbSvcWin))
    else
        send(prev, queryDeploy(idInit, idWin, nbSvcWin))
    end
else
    if (j = prev) then
        send(next, resultDeploy(idInit, idWin, nbSvcWin))
    else
        send(prev, resultDeploy(idInit, idWin, nbSvcWin))
    end
    requests(idInit, idSvc).delete()
end
/* A la réception d'un message resultDeploy(idInit, idWin, nbSvcWin) sur le
   nœud i depuis le nœud j                                           */
if (id ≠ idInit) then
    if (j = prev) then
        send(next, resultDeploy(idInit, idWin, nbSvcWin))
    else
        send(prev, resultDeploy(idInit, idWin, nbSvcWin))
    end
    requests(idInit, idSvc).delete()
else
    if (idWin ≠ -1) then
        /* deploiement possible sur le nœud d'identifiant idWin      */
    else
        /* deploiement impossible !                                  */
    end
    available ⟵ true
end
```

Algorithme 3: Stratégie de déploiement équilibré sur un arbre

```
/* Initialisation                                                    */
if available = true then
    if (isAble()) then
        idWin ⟵ id
        nbSvcWin ⟵ nbSvc
    end
    forall the (q ∈ neigh) do
        Send(q, queryDeploy(id, idWin, nbSvcWin))
    end
    requests.add(id, idSvc, id)
    available ⟵ false
end
/* A la réception d'un message queryDeploy(idInit, idWin, nbSvcWin)
   sur le nœud i depuis le nœud j                                    */
if (isAble() ∧((nbSvcWin = −1) ∨ (nbSvcWin > nbSvc)))then
    idWin ⟵ id
    nbSvcWin ⟵ nbSvc
end
if (|neigh| > 1) then
    forall the (q ∈ neigh \ j) do
        Send(q, queryDeploy(idInit, idWin, nbSvcWin))
    end
    requests.add(idInit, idSvc, j)
else
    send(j, resultDeploy(idInit, idWin, nbSvcWin))
end
/* A la réception d'un message resultDeploy(idInit, idWin, nbSvcWin)
   sur le nœud i depuis le nœud j                                    */
if ((nbSvcWin = -1) ∨ (nbSvcWin > nbSvc)) then
    idWin ⟵ id
    nbSvcWin ⟵ nbSvc
end
requests(idInit, idSvc).increase()
if (id = idInit) ∧ (requests(idInit, idSvc).nbRcv = |neigh|) then
    if (idWin ≠ -1) then
        /* deploiement possible sur le nœud idWin                    */
    else
        /* deploiement impossible !                                  */
    end
    requests(idInit, idSvc).delete()
    available ⟵ true
 else if (id ≠ idInit) ∧ (requests(idInit, idSvc).nbRcv = |neigh| − 1) then
    send(requests(idInit, idSvc).idFather, resultDeploy(idInit, idWin, nbSvcWin))
    requests(idInit, idSvc).delete()
end
```

6.2.2. Localisation d'un service dans P2P4GS

Pour localiser un service dans P2P4GS, le premier nœud sollicité sera le point d'entrée et ce dernier se chargera d'initier le processus de localisation du service.

Principe d'exécution :

1) Le nœud initiateur vérifie d'abord s'il a déjà initié le même processus de localisation ; auquel cas il ne traitera pas la nouvelle demande mais répondra juste à la fin du processus en cours. Sinon, le processus de localisation du service sera initié.

2) Opération (I), recherche locale du service avec la fonction *lookup()* :

a) La recherche s'arrête si le service est présent sur le nœud auquel *lookup()* a été exécuté ou si le nœud a connaissance de la localisation du service :
- si la localisation est effectuée au niveau du nœud initiateur ou si ce dernier reçoit un *resultLookup()*, il envoie comme résultat l'adresse du nœud détenant le service ;
- sinon, le nœud ayant localisé le service envoie le résultat à son père et de façon récursive le résultat sera acheminé jusqu'au nœud initiateur (donc le chemin inverse de la requête de localisation du service).

b) S'il n'y a pas de connaissance locale sur la localisation du service, alors le nœud effectue l'opération (II) qui consiste à envoyer la requête *queryLookup*() à :
- son voisin (*next*) s'il s'agit d'une topologie anneau orienté ;
- ses deux voisins (*next* et *prev*) pour une topologie anneau non orienté ;
- tous ses voisins (*neigh*) sauf à son père, dans le cas d'une topologie arbre.

3) Tout nœud qui reçoit la requête *queryLookup*() effectue l'opération (I).

4) L'opération (II) est effectuée par :

- cas de l'anneau orienté : tout nœud ayant reçu un *queryLookup*() sauf l'initiateur ;

- cas de l'anneau non orienté : tout nœud ayant reçu la requête *queryLookup*() au plus une fois. Si un nœud reçoit une deuxième fois une requête *queryLookup*() (surement de son autre voisin *next* (respectivement *prev*)), il envoie une requête *resultLookup*() à son voisin *prev* (respectivement *next*) ;

- cas de l'arbre : tout nœud ayant reçu la requête *queryLookup*() et ayant au moins 2 voisins. Si un nœud n'a qu'un seul voisin (son père), il lui répond par un message *resultLookup*() après avoir effectué l'opération (I).

5) Evénement déclenchant l'arrêt du processus de localisation :

- cas de l'anneau orienté : la réception d'un message *queryLookup*() par le nœud initiateur ;

- cas de l'anneau non orienté : la réception d'un message *resultLookup*() par le nœud initiateur. Si un message *resultLookup*() est reçu par un nœud autre que l'initiateur, ce dernier le transmet à son suivant ;

- cas de l'arbre : la réception par le nœud initiateur d'un message *resultLookup*() de tous ses voisins. Si un message *resultLookup*() est reçu par un nœud autre que l'initiateur, ce dernier effectue l'opération 2. Lorsque tous les voisins auquels il avait envoyé une requête *queryLookup*() lui répondent, il transmet le résultat à son père par un *resultLookup*().

6) A la fin du processus de localisation d'un service, le nœud initiateur enlève cette requête de sa pile.

Algorithme 4: Stratégie de localisation d'un service sur un anneau orienté

```
/* Initialisation                                                    */
if (lookup(idSvc) ≠ -1) then
    /* Service présent ou connu localement                           */
    result(lookup(idSvc));
else
    /* forward du message au voisin                                  */
    send(next, queryLookup(id, idSvc));
end
/* A la réception d'un message queryLookup(idInit, idSvc) sur le nœud
   i depuis le nœud j                                                */
if (id = idInit) then
    /* Service non trouvé !                                          */
    result(-1);
else
    if (lookup(idSvc) ≠ -1) then
        /* Service présent ou connu localement                       */
        resultLookup(idInit, idSvc, lookup(idSvc));
    else
        send(next, queryLookup(idInit, idSvc));
    end
end
/* A la réception d'un message resultLookup(idInit, idSvc, idOwn) sur
   le nœud i depuis le nœud j                                        */
if (id = idInit) then
    /* Service trouvé !                                              */
    result(idOwn);
else
    resultLookup(idInit, prev, idOwn);
end
```

Algorithme 5: Stratégie de localisation d'un service sur un anneau non orienté

```
/* Initialisation                                                    */
if (lookup(idSvc) ≠ -1) then
    /* Service présent ou connu localement                           */
    result(lookup(idSvc));
else
    /* diffusion de la requête à ses voisins gauche et doite         */
    send(next, queryLookup(id, idSvc));
    send(prev, queryLookup(id, idSvc));
end
/* A la réception d'un message queryLookup(idInit, idSvc) sur le nœud i
   depuis le nœud j                                                  */
if (lookup(idSvc) ≠ -1) then
    resultLookup(idInit, idSvc, lookup(idSvc));
else
    if ¬ requests(idInit, idSvc).exists() then
        requests.add(idInit, idSvc)
    else
        requests(idInit, idSvc).increase()
    end
    if (request(idInit, idSvc).nbRcv < 2) then
        if (j = prev) then
            send(next, queryLookup(idInit, idSvc))
        else
            send(prev, queryDeploy(idInit, idSvc))
        end
    else
        if (j = prev) then
            send(next, resultLookup(idInit, idSvc, -1))
        else
            send(prev, resultLookup(idInit, idSvc, -1))
        end
        requests(idInit, idSvc).delete()
    end
end
/* A la réception d'un message resultLookup(idInit, idSvc, idOwn) sur le nœud i
   depuis le nœud j                                                  */
if (id = idInit) then
    if (idOwn ≠ -1) then
        result(idOwn); /* Service retrouvé !                         */
    else
        result(-1); /* Service non retrouvé !                        */
    end
else
    if (j = prev) then
        send(next, resultLookup(idInit, idSvc, idOwn))
    else
        send(prev, resultLookup(idInit, idSvc, idOwn))
    end
    requests(idInit, idSvc).delete()
end
```

Algorithme 6: Stratégie de localisation d'un service sur un arbre

```
/* Initialisation                                                    */
if ¬ requests(id, idSvc).exists() then
    if (lookup(idSvc) ≠ -1) then
        /* Service présent ou connu localement                       */
        result(lookup(idSvc))
    else
        /* diffusion de la requête aux voisins                       */
        forall the (q ∈ neigh) do
            Send(q, queryLookup(id, idSvc))
        end
        requests.add(id, idSvc, id)
    end
end
/* A la réception d'un message queryLookup(idInit, idSvc) sur le nœud
   i depuis le nœud j                                                */
if (lookup(idSvc) ≠ -1) then
    resultLookup(idInit, idSvc, lookup(idSvc));
else
    if (|neigh| > 1) then
        /* diffusion aux voisins sauf le père                        */
        forall the (q ∈ neigh \ j) do
            Send(q, queryLookup(idInit, idSvc));
        end
        requests.add(idInit, idSvc, idFather)
    else
        Send(j, resultLookup(idInit, idSvc, -1))
    end
end
/* A la réception d'un message resultLookup(idInit, idSvc, idOwn) sur
   le nœud i depuis le nœud j                                        */
if (idOwn ≠ -1) then
    if (id = idInit) then
        result(idOwn); /* Service retrouvé !                         */
    else
        Send(requests(idInit, idSvc).idFather, resultLookup(idInit, idSvc, idOwn));
    end
    requests(idInit, idSvc).delete()
else
    requests(idInit, idSvc).increase()
    if (id = idInit) ∧ (requests(idInit, idSvc).nbRcv = |neigh|) then
        result(idOwn); /* Service non retrouvé !                     */
        requests(idInit, idSW).delete()
    else if (id ≠ idInit) ∧ (requests(idInit, idSvc).nbRcv = |neigh| − 1) then
        send(requests(idInit, idSvc).idFather, resultLookup(idInit, idSvc, -1))
        requests(idInit, idSvc).delete()
    end
end
```

6.3. Etude de complexités des primitives «deploy» et «lookup»

Pour évaluer les performances de nos algorithmes de déploiement et de localisation, nous utilisons comme critére de mesure le nombre de messages (M) échangés entre les nœuds de la communauté et de temps d'exécution (T). Le tableau 1 ci contre illustre les complexités des nos algorithmes en fonction des topologies classiques de la pile P2P à savoir l'anneau et l'arbre. n représente le nombre de nœuds.

Complexités	**Nombre de messages (M)**		**Temps d'exécution (T)**	
Primitives	deploy	lookup	deploy	lookup
anneau orienté	n	$2 \leq M \leq 2*(n-1)$	n	$2 \leq T \leq 2*(n-1)$
anneau non orienté	$2n$	$3 \leq M \leq 2n$	n	$2 \leq T \leq n$
arbre	$2*(n-1)$	$2 \leq M \leq 2*(n-1)$	$2 \leq T \leq 2*(n-1)$	

Tableau 1 – Complexités des primitives «deploy» et «lookup» en fonction des topologies anneau et arbre

D'une manière générale, nous remarquons que la topologie en anneau orienté offre des performances supérieures. En effet, l'algorithme de déploiement est très efficace sur cette topologie tant en terme de nombre de messages échangés qu'en temps d'exécution. L'algorithme de localisation reste efficace si le nombre de saut nécessaire pour atteindre le service recherché est acceptable. Cependant, plus le nombre de saut pour atteindre le service recherché devient important, plus on tend vers le pire des cas ($2*(n - 1)$).

Nous remarquons également que la topologie en anneau non orienté consomme beaucoup plus de messages tant en ce qui concerne le déploiement que la localisation. Cependant, cette topologie reste meilleure en termes de temps d'exécution (n au pire des cas).

Pour le cas de l'arbre, les bornes sont dues au fait qu'il peut prendre différentes formes de représentation allant de la chaîne, en passant par les arbres équilibrés ou non jusqu'à étoile. De ce fait, son diamêtre peut varier de $n - 1$ à 2. Ainsi, plus le diamêtre de l'arbre est faible plus nos algorithmes deviennent meilleurs avec cette topologie.

7. Conclusion et perspectives

Au constat que les grilles exploitant la notion de services sont basées sur des architectures hiérarchiques fortement centralisées, nous avons proposé dans ce papier une spécification originale d'une "grille pair-à-pair de services auto-gérés" nommée P2P4GS. La spécification est générique, non liée à un réseau P2P particulier ou à un protocole de gestion de services défini à l'avance. Après avoir présenté la spécification, nous avons proposé des algorithmes de déploiement et de localisation de services dans P2P4GS ainsi qu'une une étude de complexités de ces derniers. Les performances que nous avons obtenues sont d'une manière générale très satisfantes tant sur la topologie en anneau orienté ou non orienté que sur l'arbre.

Comme perspective immédiate, une étude de performances des stratégies de gestion de services proposées sera faite par le biais de simulations. Ce travail sera suivi par une implémentation ainsi qu'une étude de solutions de tolérance aux pannes et aux fautes. Parallélement, le développement des démonstrateurs et une comparaison des stratégies de localisation des services et des informations sur les services sont envisagés.

8. Bibliographie

[1] I. FOSTER, A. IAMNITCHI, "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing", *In M. Frans Kaashoek et Ion Stoica, editeurs : IPTPS, volume 2735 de Lecture Notes in Computer Science, pages 118-128*, Springer, 2003.

[2] B. GUEYE, O. FLAUZAC, I. NIANG, "Services pour les grilles pair-à-pair", *Eleventh African Conference on Research in Computer Science and Applied Mathematics (CARI'12)*, October 2012.

[3] I. FOSTER, C. KESSELMAN, S. TUECKE, "The anatomy of the grid : enabling scalable virtual organizations", *IJSA'01*, vol. 3, 2001.

[4] DP. ANDERSON, "BOINC : A system for public-resource computing and storage", *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[5] I. FOSTER, "Globus Toolkit version 4 : Software for Service-Oriented Systems", *JSCT*, vol. 21 n° 4, July 2006.

[6] B.S. WHITE, M. WALKER, M. HUMPHREY, A.S. GRIMSHAW, "LegionFS : A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications", *Supercomputing*, 2001.

[7] A. ROWSTRON, P. DRUSCHEL, "Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems", *In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany*, 2001.

[8] O. FLAUZAC, M. KRAJECKI, L.A. STEFFENEL, "CONFIIT a middleware for peer-to-peer computing", *Journal of supercomputing*, vol. 53 n° 1, July 2010.

[9] S. TUECKE, K. CZAIKOWSKI, I. FOSTER, J. FREY, S. GRAHAM, C. KESSELMAN, D. SNELLING, P. VANDERBILT, "Open Grid Services Infrastructure", *Global Grid Forum Draft Recommendation*, 2003.

[10] I. FOSTER, H. KISHIMOTO, A. SAVVA, D. BERRY, A. DJAOUI, A. GRIMSHAW, B. HORN, F. MACIEL, F. SIEBENLIST, R. SUBRAMANIAM, J. TREADWELL, J.V. REICH, "The Open Grid Services Architecture", *Informational Document, Global Grid Forum (GGF)*, vol. 6, n° 6, 2005.

[11] W.A. NAGUY, F. CURBERA, S. WEERAWARANA, "Web services : Why and how ?", *ACM OOPSLA, Workshop on Object-Oriented Web Services*, 2001.

[12] O. FLAUZAC, F. NOLOT, C. RABAT, L.A. STEFFENEL, "Grid of Security : a decentralized enforcement of the network security", *in Manish Gupta, John Walp, and Raj Sharman (Eds.), Threats, Countermeasures and Advances in Applied Information Security, IGI Global, pp. 426-443.*, April 2012.

[13] Y. TANAKA, H. NAKADA, S. SEKIGUCHI, T. SUZUMURA, SATOSHI MATSUOKA, "Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing", *Journal of Grid Computing*, 2003.

[14] E. CARON, F. DESPREZ, "DIET : A Scalable Toolbox to Build Network Enabled Servers on the Grid", *International Journal of High Performance Computing Applications*, vol. 20 n° 3 2006.

[15] F. BUTT, S.S. BOKHARI, A. ABHARI, A. FERWORN, "Scalable Grid resource discovery through distributed search", *International Journal of Distributed and Parallel Systems (IJDPS), Vol.2, No.5*, September 2011.

[16] A. IAMNITCHI, I. FOSTER, "A Peer-to-Peer approach to resource location in Grid environments", *(Eds.), Grid Resource Management, Kluwer*, 2003.

[17] GNUTELLA V2, "The Gnutella Protocol Development" [En ligne] http ://rfc-gnutella.sourceforge.net/developer/index.html, July 2003.

[18] I. STOICA, R. MORRIS, D. KARGER, M.F. KAASHOEK , H. BALAKRISHNAN, "Chord : A Scalable Peer-to-Peer Lookup Service for Internet Applications", *In Proceedings of SIGCOMM. San Deigo, CA*, 2001.

[19] P. TRUNFIO, D. TALIA, H. PAPADAKIS, P. FRAGOPOULOU, M. MORDACCHINI, M. PENNANEN, K. POPOV, V. VLASSOV, S. HARIDI, "Peer-to-Peer resource discovery in Grids : Models and systems", *Future Generation Computer Systems, Volume 23, Issue 7, Pages 864-878, ISSN 0167-739X*, August 2007.

[20] P. MERZ, K. GORUNOVA, "Fault-tolerant resource discovery in P2P grids", *Journal of Grid Computing, Vol. 5, pp. 319-35*, 2007.

[21] T. KOCAK, D. LACKS, "Design and analysis of a distributed grid resource discovery Protocol", *Cluster Computing, pp. 37-52*, 2012.

[22] Y. DENG, F. WANG, A. CIURA, "Ant colony optimization inspired resource discovery in P2P grid systems", *Journal of Super Computing, 49 :4-21*, 2009.

[23] D. CHEN, G. CHANG, X. ZHENG, D. SUN, J. LI, X. WANG, "A Novel P2P Based Grid Resource Discovery Model", *In Proceedings of Journal Of Network. Vol. 6, pp. 1390-1397*, October 2011.

[24] J.A. TORKESTANI, "A distributed resource discovery algorithm for P2P grids", *Journal of Network and Computer Applications 35(2012) 2028-2036*, 2012.