

Services auto-adaptatifs pour les grilles pair-à-pair

Bassirou Gueye

▶ **To cite this version:**

Bassirou Gueye. Services auto-adaptatifs pour les grilles pair-à-pair. Informatique [cs]. Université de Reims Champagne-Ardenne, 2016. Français. NNT : . tel-02883206

HAL Id: tel-02883206
https://hal.science/tel-02883206

Submitted on 28 Jun 2020

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.

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

ÉCOLE DOCTORALE SCIENCES TECHNOLOGIE SANTE (547)

THÈSE EN CO-TUTELLE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ CHEIKH ANTA DIOP DE DAKAR

Discipline : ***INFORMATIQUE***

Et

DOCTEUR DE L'UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Discipline : ***INFORMATIQUE***

Présentée et soutenue publiquement par

Bassirou GUEYE

Le 26 mai 2016

SERVICES AUTO-ADAPTATIFS POUR LES GRILLES PAIR-À-PAIR

Thèse dirigée par M. **Ibrahima NIANG, Maître de conférences, Université Cheikh Anta Diop**

Et par M. **Olivier FLAUZAC, Professeur, Université de Reims Champagne-Ardenne**

JURY

M. Michel MISSON,	, Professeur,	à l'Université de Clermont-Ferrand 2 Blaise Pas,	**, Président**
M. Eddy CARON,	, Professeur associé,	Ens Lyon Ens Sciences,	**, Rapporteur**
M. Ousmane THIARE,	, Professeur,	à l'Université Gaston Berger, Saint-Louis,	**, Rapporteur**
M. Olivier FLAUZAC,	, Professeur,	à l'Université Reims Champagne-Ardenne,	**, Examinateur**
M. Ibrahima NIANG,	, Maître de Conférences,	à l'Université Cheikh Anta Diop, Sénégal,	**, Examinateur**
M. Cyril RABAT,	, Maître de Conférences,	à l'Université Reims Champagne-Ardenne,	**, Examinateur**

À la mémoire de mon père et à celle de ma mère.
Puisse Dieu, le tout puissant, vous avoir en sa sainte miséricorde !

À ma femme

À toute ma famille

Remerciements

Je rend grâce à Allah, le Très Miséricordieux, pour m'avoir illuminé et mené jusqu'ici.

Je tiens à exprimer mes très sincères remerciements et toute ma gratitude à mes directeurs de thèse *Olivier Flauzac*, Professeur à l'Université de Reims Champagne-Ardenne et *Ibrahima Niang*, Maître de Conférences HDR à l'Université Cheikh Anta Diop de Dakar, pour la confiance qu'ils m'ont accordée en m'offrant l'opportunité de faire une thèse. Je les remercie également de leur soutien, de leur conseils tout au long de ce travail et surtout de l'indépendance qu'ils m'ont donnée dans la réalisation de ce projet tout en étant exigeant pour un travail de qualité. Enfin, je les remercie une nouvelle fois pour leur humanisme et toute l'aide qu'ils m'ont apportée. Qu'ils trouvent dans ce travail l'expression de mon infinie reconnaissance.

J'adresse mes sincères remerciements à l'Agence Universitaire de la Francophonie (AUF) pour avoir financé et ainsi permettre la réalisation de cette thèse.

Je tiens à remercier vivement mon encadrant *Cyril Rabat*, Maître de Conférences à l'Université de Reims Champagne-Ardenne, pour son aide à l'élaboration de ce travail. Un grand merci pour les remarques constructives, les suggestions pointues, les discussions intéressantes et les relectures attentives.

J'adresse mes vives remerciements à *Eddy Caron*, Maître de Conférences HDR à l'École Normale Supérieure de Lyon et *Ousmane Thiaré*, Professeur à l'Université Gaston Berger de Saint-Louis, pour m'avoir fait l'honneur d'accepter de rapporter cette thèse et pour toutes les remarques et suggestions constructives pour améliorer la qualité du manuscrit. Je remercie également *Michel Misson*, Professeur à l'Université de Clermont-Ferrand, pour avoir accepté de participer au jury.

Je remercie tout le personnel de la Section Informatique et tout le personnel du Centre de Calcul de l'UCAD. J'associe à ces remerciements tous les membres du laboratoire LID. Je remercie *Bamba Gueye* pour la lecture et les corrections du manuscrit malgré son calendrier chargé. Un grand merci également à *Mandicou Bâ* pour nos différents échanges. Merci aux doctorants, *Abel Diatta*, *Malick Gaye* et mention spéciale à *Papa Dame Bâ*.

Je remercie également tous les membres du laboratoire CReSTIC. Je remercie *Florian Legendre* et *Luiz Steffenel* pour leurs conseils et leur sympathie. Merci aux doctorants, *Mahamat Charfadine,Carlos Gonzalez* et un grand merci à *Thierno Diallo*. Je remercie également les secrétaires du CReSTIC pour leur disponibilité et leur sympathie.

Les encouragements de mes amis étaient la bouffée d'oxygène qui me ressourçait dans les moments difficile où l'on a besoin d'un petit mot, d'un petit geste, aussi humble soit-il, de soutien moral. Je pense notamment à ..., je ne les citerai pas, ils se reconnaîtront.

J'exprime ma profonde reconnaissance à mon cher père et à ma chère mère. Je suis très fière de vous et je ne cesserai de vous remercier et de prier pour vous. Puisse Dieu, le tout puissant, vous avoir en sa sainte miséricorde !

A tous les membres de ma famille, de près ou de loin, à mes sœurs, à mon petit frère, à *Cheikh Gueye*, pour leurs prières et leurs encouragements.

Enfin, je remercie très sincèrement ma femme *Soda*, pour son amour, sa tendresse, sa patience, ses encouragements, pour avoir été à mes côtés durant les moments difficile et avoir supporté que je m'éloigne d'elle très souvent afin de réaliser ce projet.

MERCI...

Résumé

La gestion de ressources distribuées à l'échelle planétaire dans plusieurs organisations virtuelles implique de nombreux défis. Dans cette thèse, nous proposons un modèle pour la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle. Ce modèle, nommé P2P4GS, présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme d'exécution de services. De plus, il est générique, c'est-à-dire applicable sur toute architecture pair-à-pair. Pour garantir cette propriété, vu que les systèmes distribués à large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous avons proposé de structurer le système de grille pair-à-pair en communautés virtuelles (*clusters*). L'approche de structuration est complètement distribuée et se base uniquement sur le voisinage des nœuds pour l'élection des responsables de clusters appelés PSI (*Proxy Système d'Information*). D'autre part, afin de bien orchestrer les communications au sein des différentes communautés virtuelles et aussi permettre une recherche efficace et exhaustive de service, lors de la phase de structuration, un arbre couvrant constitué uniquement des PSI est maintenu. Les requêtes de recherche vont ainsi être acheminées le long de cet arbre. Outre la découverte de services, nous avons proposé des mécanismes de déploiement, de publication et d'invocation de services. Enfin, nous avons implémenté et analysé les performances de P2P4GS. Afin d'illustrer sa généricité, nous l'avons implémenté sur Gia, Pastry et Kademlia des protocoles pair-à-pair opérant de manières totalement différentes. Les tests de performances ont montré que le P2P4GS fournit une bonne résistance aux pannes et garantit un passage à l'échelle en termes de dimensionnement du réseau et également de coût de communications.

Mots-clés : Systèmes pair-à-pair, Grilles de services, gestion de ressources, modélisation, structuration, tolérance aux pannes.

Abstract

Resource management is a key issue for Grid systems which consist of several geographically dispersed virtual organizations. In this thesis, we propose a model for dynamic services management in large-scale peer-to-peer Grid environments. This model named P2P4GS, presents originality not to link peer-to-peer infrastructure to the execution services platform. In addition, the middleware is generic and can be applied on any peer-to-peer architecture. Meanwhile, the increasing size of resources and users in large-scale distributed systems has lead to a scalability problem. To ensure scalability, we propose to organize the peer-to-peer Grid nodes in virtual communities so called clusters. The structuring approach is completely distributed, and only requires local knowledge about nodes neighborhood for election of cluster managers called ISP (Information System Proxy). On the other hand, in order orchestrate communications in the various virtual communities and also enable an efficient service discovery, during structuring process, a spanning tree only constituted of ISP is maintained. Therefore, search queries will be routed along the spanning tree. Besides the service discovery, we proposed service deployment, publication and invocation mechanisms. Finally, we implemented and analyzed the performance of P2P4GS. To illustrate its genericity, we implemented it on protocols which operate in fully different way. These protocols are Gia, Pastry and Kademlia. Performance tests show that, on the one hand, the P2P4GS provides good fault tolerance and ensures the scalability in terms of the clusters distribution and communication cost.

Keywords : Peer-to-peer Systems, Grid services, resources management, modelisation, clustering, Fault tolerance.

Table des matières

Remerciements **i**

Résumé **iii**

Abstract **v**

Table des matières **vii**

Liste des figures **x**

Liste des algorithmes **xi**

Liste des tableaux **xii**

Introduction **1**

1 Etat de l'art sur les systèmes distribués **7**

1.1 Généralités sur les systèmes distribués 8

1.1.1 Définitions 8

1.1.2 Modélisation d'un système distribué 8

1.1.3 Modèles de communication 11

1.1.4 Algorithmes distribués 13

1.1.5 Notions de panne et de tolérance aux pannes 14

1.1.6 Les modèles d'architecture 17

1.2 Les systèmes pair-à-pair 19

1.2.1 Définition et terminologies 19

1.2.2 Caractéristiques du pair-à-pair 20

1.2.3 Architectures de réseaux pair-à-pair 22

1.3 Conclusion 28

2 Les services dans les systèmes distribués **31**
2.1 Généralités sur les services . . . 32
2.1.1 Notion de service . . . 32
2.1.2 Approche conceptuelle orientée service (SOA) . . . 33
2.2 Services Web . . . 36
2.2.1 Contexte . . . 36
2.2.2 Définitions de la notion de Service Web . . . 37
2.2.3 Les technologies des Services Web . . . 38
2.3 Services de grilles informatiques . . . 44
2.3.1 Concept de grille informatique . . . 44
2.3.2 Architecture d'une grille . . . 47
2.3.3 Topologies de grilles . . . 49
2.3.4 Les services de grille . . . 50
2.4 Gestion de services dans un environnement de grille informatique à large échelle . . . 52
2.4.1 Objectifs et motivations . . . 53
2.4.2 Mécanismes de découverte de services dans les grilles . . . 54
2.5 Conclusion . . . 61

3 Spécifications de services dans un environnement de grille P2P à large échelle **63**
3.1 Motivations et objectifs . . . 64
3.2 Spécifications de P2P4GS . . . 65
3.2.1 Concepts de bases . . . 65
3.2.2 Modèle d'architecture . . . 68
3.3 Approche de structuration du système en communautés . . . 72
3.3.1 Présentation de la solution de structuration . . . 72
3.3.2 Algorithme de structuration . . . 75
3.3.3 Mécanismes d'adaptation à la dynamique du système . . . 78
3.4 Synthèse . . . 80

4 Gestion des services dans P2P4GS **83**
4.1 Motivations . . . 84
4.2 Primitives de gestion de services . . . 85
4.2.1 Formalisme de notations . . . 85
4.2.2 Service de déploiement : `deploy` . . . 86
4.2.3 Enregistrement d'un service : `save` . . . 90
4.2.4 Service de découverte : `lookup` . . . 91

4.2.5 Invocation et exécution de service : `invoke` et `exec` 94
4.3 Synthèse 95

5 Expérimentation et évaluation de la spécification P2P4GS 97
5.1 Environnement de simulation 98
5.2 Description des overlays implémentés 99
5.2.1 Le protocole Gia 100
5.2.2 Le protocole Pastry 101
5.2.3 Le protocole Kademlia 102
5.3 Mesures de performances 104
5.3.1 Métriques de performance et paramètres de simulations 104
5.3.2 Performances de la première approche de structuration 105
5.3.3 Impact du degré minimal requis sur les performances du système . . 106
5.3.4 Impact des pannes sur la découverte de services 114
5.4 Synthèse 115

Conclusion 117

Bibliographie 121

Table des figures

1.1 Graphe non-orienté et graphe orienté 9
1.2 Quelques topologies classiques des systèmes distribués 11
1.3 Illustration d'un arbre couvrant du graphe (a) 12
1.4 La chaîne fondamentale des entraves 15
1.5 Modèle client-serveur et modèle pair-à-pair 17
1.6 Taxonomie des architectures P2P 22
1.7 Un exemple d'architecture pair-à-pair décentralisée structurée [SMK+01]. . 26
1.8 Architecture pair-à-pair hybride 28

2.1 Modèle d'interaction SOA 35
2.2 Structure d'un document WSDL 40
2.3 Structure d'un annuaire UDDI 41
2.4 Structure d'un message SOAP 42
2.5 Quelques grilles populaires à travers le monde 45
2.6 Architecture d'une grille informatique 47
2.7 Convergence Grille et Service Web 52
2.8 Mécanismes de découverte de services dans un environnement de grille . . . 54
2.9 Architecture de DIET 56

3.1 Architecture du réseau de recouvrement 66
3.2 Architecture du système P2P4GS 68
3.3 Connexion d'un nœud dans le système 73
3.4 Evolution de la structuration et choix d'un PSI passerelle 74

4.1 Scénario d'exécution du service S_6 95
4.2 Scénario d'exécution du service S_{10} 96

5.1 Structure des modules du simulateur OMNeT++ 99
5.2 Pourcentage de PSI en fonction du protocole P2P et de la taille du réseau . 106

5.3 Protocole Gia : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 107
5.4 Protocole Pastry : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 108
5.5 Protocole Kademlia : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 108
5.6 Protocole Gia : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 110
5.7 Protocole Pastry : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 110
5.8 Protocole Kademlia : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau 111
5.9 Nombre total de messages échangés en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau 112
5.10 Diamètre de l'arbre couvrant, en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau 113
5.11 Taux de succès d'une recherche de service en fonction du pourcentage de nœuds en panne 114

Liste des Algorithmes

1 À la réception d'un message responseStatus(id_v, $status_v$) du nœud v 76
2 À la réception d'un message updateStatus(id_v, $status_v$) du nœud v 77
3 À la réception d'un clusterManagement($serviceList_v$, $neighList_v$) ou à l'expiration de $\mathcal{T}_{timeout}$ 77
4 À la réception d'un message masterRouteManagement(id_v) du nœud v 78
5 À la réception d'un message responseElection(id_v, $degree_v$) ou à l'expiration du *timeout* 80
6 À la réception de la description des contraintes de déploiement depuis le nœud *nodeRequester* 88
7 À la réception d'un message serviceDiscovery(*keyReq*, *constraints*, *entryPoint*, TTL) depuis le nœud v 89
8 À la réception d'une requête de recherche *serviceQuery* depuis le nœud *nodeRequester* 92
9 À la réception d'un message discoveryRequest(*keyReq*, *serviceQuery*, *entryPoint*, *distance*) depuis le nœud v 93
10 À la réception d'un message discoveryResponse(*keyQuery*, $serviceFindList$, *distance*) ou à l'expiration du temps de recherche 94

Liste des tableaux

2.1 Synthèse des différentes approches de découverte de services dans un environnement de grille 60

3.1 Paramètres, variables et structure des messages de P2P4GS 75

4.1 Paramètres, variables et structure des messages de gestion de P2P4GS . . . 86

5.1 Table de routage d'un pair d'identifiant 10233102 (extrait de Pastry [RD01]) 102
5.2 Paramètres de simulation pour l'évaluation de la spécification P2P4GS . . 105

Introduction

Contexte et motivations

Depuis son avènement, l'Internet a accéléré le développement d'applications dans tous les domaines. L'évolution des systèmes informatiques a pour effet le développement de nombreuses ressources informationnelles, matérielles et logicielles distribuées, telles que les ressources de calcul et de stockage, les bases de données, les divers types d'applications et d'utilitaires, etc.

Parallèlement à ce développement, les besoins en termes de puissance de calcul, de capacité de stockage, etc. sont de plus en plus grands. Face à cette demande croissante de puissance, la communauté informatique s'est intéressée aux architectures distribuées à large échelle de type grilles, afin d'offrir des solutions pour le stockage de données et le calcul réparti à un plus grand nombre d'applications et d'utilisateurs. Le principe des grilles est de mettre en commun des ressources partagées, distribuées et hétérogènes [FKT01]. Par le biais des grilles, les utilisateurs ont la possibilité d'accéder à des ressources distantes de calcul et de stockage, de lancer des applications qui demandent des ressources inexistantes ou non disponibles localement.

D'autre part, cette évolution des systèmes informatiques a apporté une modification profonde dans la manière d'utiliser les ressources informatiques. En effet, avec l'émergence de l'architecture orientée service pour répondre aux problèmes d'interopérabilité, les ressources informatiques sont de plus en plus exposées sous la forme de services. Cette tendance de l'orientée service s'est manifestée dans le cadre du Web avec notamment les Services Web [CDK+02] mais aussi dans le cadre des grilles informatiques avec les grilles de services. Ces grilles de services visent ainsi à définir des mécanismes pour virtualiser les ressources et les restituer sous forme de services, afin de pouvoir les assembler et les désassembler en fonction des besoins des utilisateurs [FKNT02a].

Dès lors, avec cette démocratisation des grilles, la gestion de ressources géographiquement dispersées et disponibles dans plusieurs organisations virtuelles (Universités, insti-

tuts, entreprises, etc.) implique de nombreux défis. En effet, les ressources sont de nature très hétérogènes et dynamiques. Une gestion intégrale et transparente des ressources est donc nécessaire pour conserver la consistance d'un tel système. De plus, les demandes en puissance de calcul et en capacité de stockage sont de plus en plus énormes. Le système doit ainsi être en mesure de passer à l'échelle tout en limitant les coûts de gestion et de communication.

Pour répondre à ces exigences, les grilles n'ont pas cessé d'évoluer en termes d'architectures qui guident le développement de tous les composants d'une application. En effet, les systèmes de grilles traditionnelles présentent des architectures centralisées ou hiérarchiques. De telles architectures sont sensibles aux pannes et généralement inaptes à passer à l'échelle [NRNH14, TTP+07].

La conception des grilles s'est par la suite orientée vers une approche Pair-à-Pair (abrégé P2P) [FI03, IFN02]. Dans un tel modèle où chaque entité est à la fois client et serveur, les services sont répartis sur l'ensemble des acteurs du système. De plus, les systèmes pair-à-pair offrent de nombreux avantages grâce à leurs propriétés fondamentales et inhérentes telles que l'auto-organisation, la tolérance aux pannes, le passage à l'échelle, le changement dynamique de topologie, etc. [ATS04].

On distingue plusieurs modèles d'architectures de grille pair-à-pair à savoir, le modèle décentralisé structuré, le modèle décentralisé non-structuré et le modèle hybride ou super-pair. Chacun de ces modèles bénéficie de plusieurs avantages qu'offrent les systèmes pair-à-pair. En effet, suivant l'environnement où ces modèles s'exécutent (peu ou très dynamique, homogène ou hétérogène, large échelle ou échelle moyenne, etc.), certaines architectures sont plus adaptées que d'autres.

Soulignons que la découverte de ressources constitue un des défis essentiels dans un environnement de grille à large échelle [NRNH14, TTP+07]. En effet, les ressources d'une grille pouvant être distribuées à l'échelle planétaire, rechercher et localiser un service résolvant un besoin spécifique devient un enjeu majeur. Il est donc important de tenir en compte les caractéristiques de l'environnement des grilles lors du choix du modèle d'architecture. Afin d'éviter de saturer le réseau, la recherche doit induire le moins de messages possible tout en restant exhaustive. En outre, le mécanisme de recherche doit être flexible pour permettre une grande expressivité des requêtes utilisateurs.

Contributions

En considérant toutes les motivations énoncées ci-dessus, nous proposons dans cette thèse un modèle nommé P2P4GS (*Peer-To-Peer For Grid Services*), pour la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle.

Ce modèle présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plateforme d'exécution de services. En fait, la couche de gestion de la grille pair-à-pair est séparée de celle de la localisation et d'invocation de services. Nous faisons dans le cadre de la modélisation le choix de ne pas nous figer sur une architecture pair-à-pair typique d'exécution ; mais nous spécifions les opérations de manière détachée. Nous proposons ainsi un modèle d'architecture constitué de quatre couches d'abstraction. Ces couches superposées mettent en évidence les différents mécanismes sous-jacents à l'environnement de grille pair-à-pair ainsi que les interactions entre les différentes entités du système.

De plus, soulignons que le modèle P2P4GS est générique, c'est-à-dire qu'il est applicable sur toute architecture pair-à-pair. Pour garantir cette propriété, étant donné que les systèmes distribués à large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous proposons de structurer le système de grille pair-à-pair en communautés virtuelles aussi appelées groupes virtuels ou simplement *clusters*. Cette structuration présente deux caractéristiques intéressantes à savoir la limitation des communications et le passage à l'échelle. En effet, comme le soulignent les auteurs de [Bas99, MSC+12, SM04], une structuration efficace d'un réseau permet de garder les performances satisfaisantes même avec l'augmentation de la taille du système.

Notre approche de structuration est, d'une part, complètement distribuée et se base uniquement sur le voisinage des nœuds pour l'élection des responsables de clusters appelés PSI (*Proxy Système d'Information*). En vue de ne pas surcharger les PSI, nous proposons de répartir certaines tâches sur d'autres types de nœuds distingués. Ainsi, lorsqu'un processus de découverte d'un service aboutit, le nœud ayant déclenché ce processus devient temporairement, en fonction de ses capacités (CPU, RAM, etc.), PI (*Proxy Invoquant*) ou PL (*Proxy Localisant*) pour ce service. De cette manière, les nœuds proxys vont assurer collectivement la gestion des ressources partagées.

D'autre part, afin de bien orchestrer les communications au sein des différentes communautés virtuelles et aussi permettre une recherche efficace de services dans le système, un arbre couvrant constitué uniquement des PSI est maintenu. La particularité de la solution est que le processus de structuration du système P2P4GS et la construction de l'arbre couvrant se font simultanément. Ce qui permet de minimiser le coût des communications en termes de messages de gestion du réseau *overlay*. Par ailleurs, les requêtes de recherche vont être acheminées le long de cet arbre. Ceci permet une recherche exhaustive puisque tous les services partagés dans le système sont indexés au niveau des différents PSI constituant l'arbre couvrant.

Outre le mécanisme de découverte de services, nous proposons des mécanismes de déploiement, de publication et d'invocation de services.

Enfin, par le biais de simulations sous OverSim [BHK07], nous avons analysé les performances du système P2P4GS. Afin d'illustrer sa généricité, nous l'avons implémenté sur des protocoles P2P opérant de manière totalement différentes, à savoir Gia [CRB+03] (qui construit un overlay non structuré), Pastry [RD01] (qui construit un overlay structuré en anneau) et Kademlia [MM02] (qui construit un overlay structuré en hypercube bien que souvent modélisé en arbre).

Les résultats de simulations ont montré, d'une part, que notre solution garantit un passage à l'échelle en termes de dimensionnement du réseau et aussi de coût communications. D'autre part, parmi ces trois protocoles P2P implémentés, l'intergiciel P2P4GS fournit une meilleure résistance aux pannes lorsqu'il exploite Gia ou Kademlia en tant que protocole P2P sous-jacent.

Organisation du manuscrit

Ce manuscrit est structuré en cinq chapitres.

Dans le chapitre 1, nous présentons le contexte général dans lequel se place nos travaux, à savoir les systèmes distribués. Nous décrivons ainsi les concepts qui les spécifient. Par la suite, nous présentons les systèmes pair-à-pair qui constituent un modèle spécifique des systèmes distribués.

Le chapitre 2 est consacré à l'étude des services ainsi que leurs environnements d'exécution. Après avoir défini la notion de service et présenté l'Architecture Orientée Service (SOA), nous mettrons l'accent sur le modèle des services Web ; puis celui des services de grilles. Ensuite, nous faisons un état de l'art sur les mécanismes de gestion de services dans un environnement de grille. Une synthèse de ces mécanismes nous permet d'exposer notre problématique de recherche.

Dans le chapitre 3, nous présentons les spécifications de P2P4GS. Nous définissons les principaux concepts de base sur lesquels s'appuient le système P2P4GS. Par la suite, nous décrivons le modèle d'architecture et présentons notre approche de structuration d'un système de grille pair-à-pair à large échelle. Finalement, nous décrivons ensuite le principe d'exécution de P2P4GS ainsi que son comportement face aux pannes dans un tel environnement.

Le chapitre 4 est consacré à la description des différents mécanismes de gestion de services qu'offre le système P2P4GS. Les différentes primitives qui entrent en jeu dans le cycle de vie d'un service, à savoir le déploiement, l'enregistrement (ou la publication), la localisation et l'invocation ainsi que l'exécution sont décrites dans cette partie.

Dans le chapitre 5, nous présentons une série de simulations en vue d'analyser les performances du système P2P4GS. Nous décrivons notre environnement de simulation ainsi que les protocoles implémentés à savoir Gia, Pastry et Kademlia. Ensuite, nous définissons les différentes métriques d'évaluation de performances de notre système. Puis, nous présentons les résultats de simulations obtenus.

Finalement, nous clôturons ce manuscrit par une synthèse générale. Nous résumons ainsi les principales contributions faites dans ce travail et soulevons en guise de perspectives, des pistes de réflexions futures.

Chapitre 1

Etat de l'art sur les systèmes distribués

Résumé. *Ce chapitre présente le contexte dans lequel se place nos travaux, à savoir les systèmes distribués et en particulier les réseaux pair-à-pair. Dans la première partie, nous présentons les systèmes distribués, ainsi que leurs propriétés inhérentes. De plus, nous détaillons ses modèles de communication ainsi que quelques algorithmes classiques. Par la suite, nous définissons les notions de pannes et la tolérance aux pannes. Dans un deuxième temps, nous présentons le modèle des systèmes pair-à-pair qui est un modèle de systèmes distribués auquel nous nous sommes intéressés dans cette thèse. Nous proposons ainsi un ensemble de définitions et de terminologies qui le spécifient. Puis, nous présentons les caractéristiques ainsi que les propriétés qu'elles lui confèrent. Enfin, nous passons en revue les différentes architectures de systèmes pair-à-pair.*

Sommaire

1.1 Généralités sur les systèmes distribués 8
1.1.1 Définitions 8
1.1.2 Modélisation d'un système distribué 8
1.1.3 Modèles de communication 11
1.1.4 Algorithmes distribués 13
1.1.5 Notions de panne et de tolérance aux pannes 14
1.1.6 Les modèles d'architecture 17
1.2 Les systèmes pair-à-pair 19
1.2.1 Définition et terminologies 19
1.2.2 Caractéristiques du pair-à-pair 20
1.2.3 Architectures de réseaux pair-à-pair 22
1.3 Conclusion 28

1.1 Généralités sur les systèmes distribués

1.1.1 Définitions

Un système distribué est défini comme étant une collection d'entités de traitement et de stockage qui sont autonomes, interconnectées à l'aide d'un réseau de communication et pouvant s'échanger des informations les unes avec les autres [Tel94].

Les entités peuvent être des machines (ordinateurs), des processus, des processeurs, etc. et sont généralement appelées nœuds ou sites du système distribué. Pour être qualifié d'autonome, chaque nœud doit avoir son propre système de contrôle local indépendant des autres nœuds.

L'objectif d'un système distribué est de fournir un service qui ne pourrait pas être réalisé par une seule entité en termes de fonctionnalité, disponibilité, puissance de traitement, capacité de stockage, temps de réponse, fiabilité, etc. Pour ce faire, les nœuds coopèrent à l'aide d'un modèle de communication afin d'accomplir la tâche globale du système dont le déroulement est induit par un algorithme distribué.

Un algorithme distribué définit les règles de fonctionnement d'un système distribué [Lyn96]. Il est composé d'une collection d'algorithmes locaux. Au niveau de chaque nœud, l'exécution d'un algorithme local est déclenchée par un événement qui peut être de type réception de message ou épuisement d'un temps prédéfini (compte-à-rebours ou *timeout*).

Les systèmes distribués offrent plusieurs utilisations qui ont favorisé leur développement massif. Ils peuvent permettre le partage de ressources, l'augmentation de la fiabilité des données par le biais de réplications, la haute disponibilité, l'amélioration des performances par le parallélisme, la simplification de la conception d'un système par une structuration modulaire, etc.

1.1.2 Modélisation d'un système distribué

Dans cette section, nous décrivons dans un premier temps le modèle classiquement utilisé pour représenter un système distribué. Par la suite, nous donnons quelques définitions liées à ce modèle. Enfin, nous présentons les topologies de graphes les plus couramment rencontrées dans la littérature.

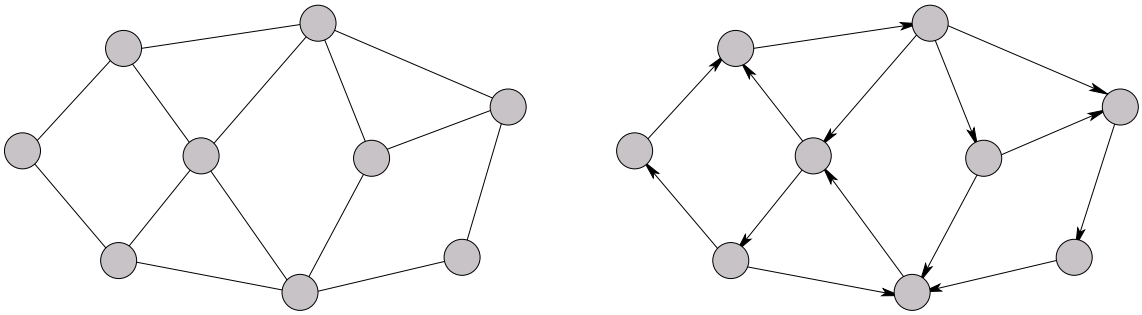
Définitions

Un système distribué est modélisé sous la forme d'un graphe $G = (V, E)$ où V représente l'ensemble des nœuds du système et E l'ensemble des liens de communication entre les différents nœuds.

Définition 1.1. ***Lien de communication***
Lorsque deux noeuds u et v du graphe sont reliés, il existe un lien (u, v) ∈ E.

Si le graphe est non-orienté, alors nous avons (u, v) ∈ E ⇔ (v, u) ∈ E. Dans ce cas, le lien est appelé arête. La Figure1.1(a) illustre un exemple de graphe non-orienté.

Si par contre (u, v) ∈ E ⇏ (v, u) ∈ E, on dit alors que le graphe est orienté. Dans ce cas, le lien est appelé arc et est symbolisé par une flèche. La Figure1.1(b) illustre un exemple de graphe orienté.



(a) Graphe non-orienté (b) Graphe orienté

FIGURE 1.1 – Graphe non-orienté et graphe orienté

Définition 1.2. ***Chemin***
Soit u et v deux nœuds du graphe. Un chemin existe entre u et v, s'il existe une suite finie de liens consécutifs reliant u à v.

Définition 1.3. ***Voisin***
Soient u et v deux nœuds du graphe. On dit que v est voisin de u si et seulement si (u, v) ∈ E. En d'autres termes, le nœud u peut communiquer directement avec le nœud v.

Définition 1.4. ***Voisinage***
Le voisinage d'un nœud u, noté $Neigh_u$, représente l'ensemble de tous les nœuds qui sont voisins de u. En d'autres termes, c'est l'ensemble des nœuds avec lesquels u peut communiquer directement.

Définition 1.5. ***Degré***
Le degré d'un nœud u, noté Δ_u, représente le nombre de nœuds dans le voisinage de u. D'où, $\Delta_u = |Neigh_u|$.

Définition 1.6. ***Graphe connexe***
Le graphe G est connexe si et seulement s'il existe un chemin entre tout couple de nœuds (u, v) de E. C'est-à-dire que le réseau d'interconnexion permet le dialogue entre deux sites arbitrairement choisis qui ne sont pas nécessairement voisins.

Dans nos travaux de recherche, nous considérons des systèmes distribués représentés par des graphes connexes. Toutefois, les solutions que nous proposons tolèrent la non-connexité du système durant un temps relativement court à la suite de changements topologiques. Au-delà d'un temps borné, un nœud non-atteignable est considéré comme déconnecté.

Topologies classiques

Le graphe d'un système distribué peut être aléatoire ou régulier. Dans le premier cas, on dit que la topologie du système distribué suit un graphe quelconque ; c'est-à-dire que les interconnexions entre les différents nœuds du système ne respectent aucune règle particulière. Dans le second cas, on dit que la topologie obéit à un graphe régulier ; c'est-à-dire que les interconnexions entre les différents nœuds du système respectent certaines règles de structuration.

Dans la suite, nous détaillons les topologies classiques des systèmes distribués.

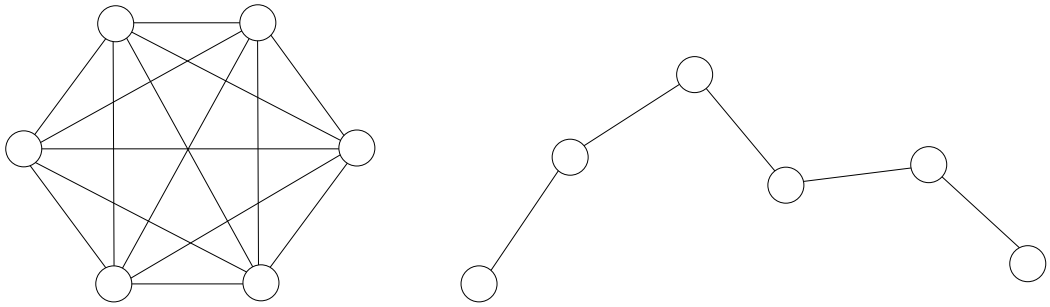
Définition 1.7. ***Graphe complet***
C'est un réseau où pour tout couple de nœuds (u, v), u est voisin de v. Dans ce modèle, chaque nœud peut ainsi communiquer directement avec tous les autres nœuds du sytème. La Figure1.2(a) est un exemple de graphe complet constitué de 6 nœuds.

Définition 1.8. ***Chaîne***
Une chaîne de N nœuds, est un graphe connexe où les N – 2 nœuds ont un degré égal à 2 et les deux autres nœuds (qui sont en extrémités) ont un degré égal à 1. La Figure1.2(b) est un exemple de chaîne constitué de 6 *nœuds.*

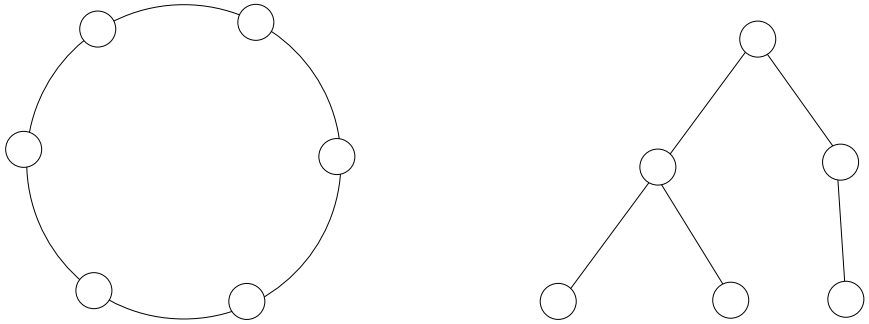
Définition 1.9. ***Anneau***
Un anneau est un cas particulier d'une chaîne où les deux nœuds à l'extrémité sont reliés. L'anneau peut être orienté ou non-orienté. La Figure1.2(c) est un exemple d'anneau constitué de 6 nœuds.

Définition 1.10. ***Arbre***
Un arbre de N nœuds est un graphe connexe contenant exactement N – 1 liens de communication. Sa particularité est qu'il ne contient pas de cycle. Donc, il existe qu'un unique chemin entre deux nœuds u et v quelconques. La Figure1.2(d) est un exemple d'arbre constitué de 6 *nœuds.*

L'un des nœuds de l'arbre est appelé la racine. Si u est plus proche que v de la racine, alors u est dit père de v et v est un fils de u. Ainsi, le nœud racine n'a pas de père. Un nœud qui ne possède pas de fils est appelé une *feuille*. Son degré est alors égal à 1.



(a) Graphe complet (b) Chaîne



(c) Anneau (d) Arbre

FIGURE 1.2 – Quelques topologies classiques des systèmes distribués

Remarque 1.1. *Il est possible de construire un arbre couvrant [Epp99, Gär03, Kru56] sur tout graphe quelconque connexe.*

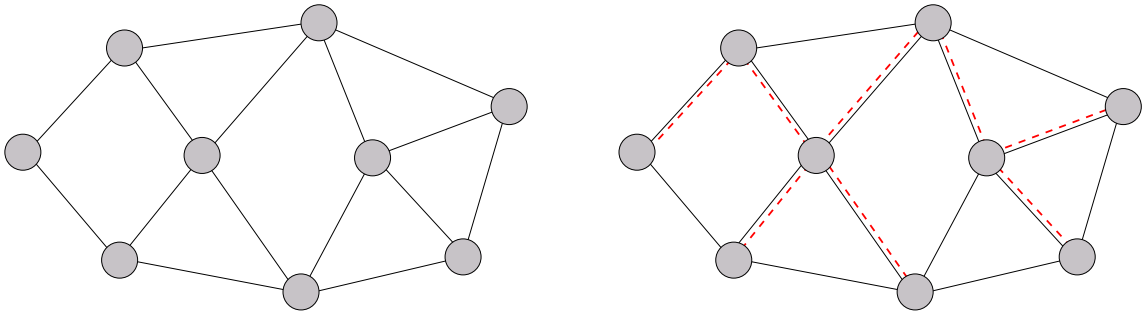
Définition 1.11. ***Arbre couvrant***
Soit $G = (V, E)$, un graphe non orienté et connexe. Un arbre couvrant de G est un arbre inclu dans ce graphe et qui présente les propriétés suivantes :

- *il connecte tous les sommets du graphe ;*
- *il existe exactement un unique chemin entre tout couple nœuds (u, v) : $E_{AC} \subseteq E_G$.*

Le fait qu'il soit un arbre lui confère sa propriété d'acyclité. La Figure 1.3(b) illustre un exemple d'arbre couvrant (les liens sont représentés en pointillés) d'un graphe quelconque illustré à gauche (Figure 1.3(a)).

1.1.3 Modèles de communication

Un système distribué est constitué d'un ensemble d'entités qui coopèrent à l'aide d'un modèle de communication afin d'accomplir la tâche globale du système. Nous distinguons dans la littérature le modèle de communication à mémoire partagée et celui à passage de messages [WA99].



(a) Graphe quelconque (b) Arbre couvrant

FIGURE 1.3 – Illustration d'un arbre couvrant du graphe (a)

La communication dans le modèle à mémoire partagée est implicite. L'information est transmise lors de l'écriture dans une zone de la mémoire partagée, et récupérée quand un autre processus vient lire cette zone.

Dans le modèle à passage de messages, chaque nœud possède donc sa propre mémoire privée et est le seul à y avoir accès. De ce fait, les nœuds communiquent entre eux par émission et réception de messages via des canaux de communication.

On parle de transmissions en mode FIFO (*First-In First-Out*) lorsque les canaux de communication délivrent les messages suivant l'ordre d'émission de ceux-ci. Toutefois, l'ordre des messages reçus peut éventuellement être différent de l'ordre des messages transmis ; dans ce cas on dit qu'il y a déséquencement des messages.

Dans nos travaux de recherche, nous nous intéressons au modèle à passage de messages car il est plus réaliste. En effet, comme nous l'avons décrit plus haut, pour que les nœuds d'un système distribué soient qualifiés d'autonomes, l'auteur de [Tel94] précise que chacun d'eux doit avoir son propre système de contrôle local (donc propre mémoire privée), indépendant des autres nœuds.

Soulignons que les communications dans un modèle à passage de messages peuvent être synchrones ou asynchrones. On parle de communications synchrones quand le transfert d'informations n'est possible qu'après une synchronisation globale des nœuds émetteurs et récepteurs. Par contre, lorsque les temps de communication entre nœuds répartis du système ne peuvent être prévus de façon précise et absolue, on dit alors que le système fonctionne en mode asynchrone. Dans ce cas de figure, les délais d'acheminement des messages le long des canaux de communication sont finis mais non bornés.

Nous pouvons ainsi conclure que le modèle de communication synchrone semble être plus contraignant que le modèle de communication asynchrone. En conséquence, pour nous rapprocher le plus possible des conditions réelles, nous nous alignons dans ce travail au modèle de communication asynchrone à passage de messages.

1.1.4 Algorithmes distribués

Nous avons vu que les nœuds du système distribué coopèrent par échange d'informations afin de réaliser un objectif commun. Les étapes de calcul qui caractérisent cette collaboration sont décrites par des algorithmes distribués.

Un algorithme distribué définit les règles de fonctionnement d'un système distribué [Lyn96, Ray85, Tel94]. Il est composé d'une collection d'algorithmes locaux. Au niveau de chaque nœud, l'exécution d'un algorithme local est déclenchée par un événement qui peut être de type réception de message ou échéance d'un *timeout*.

Nous allons dans ce qui suit présenter quelques algorithmes distribués classiques qui constituent des primitives de base pour la mise en œuvre d'un système distribué.

Algorithmes à vagues

Les algorithmes à vagues [Cha82] sont utilisés pour la diffusion d'une information dans le réseau. Cette information peut être de type synchronisation, envoi d'un ordre, calculs locaux (calcul d'une fonction dont chaque nœud possède une partie des entrées), demande d'état, etc. Après la diffusion d'une information, chaque site peut éventuellement prendre localement une décision.

Un cas particulier des algorithmes à vagues est l'algorithme PIF (*Propogation of Information with Feedback*) [Seg83] qui signifie littéralement propagation d'informations avec retour. L'objectif de cet algorithme est de propager une information et d'en faire le retour jusqu'au site initiateur.

Algorithmes d'exclusion mutuelle

L'objectif des algorithmes d'exclusion mutuelle est d'éviter que des ressources partagées d'un système ne soient utilisées en même temps par plusieurs nœuds du système.

Ce paradigme, introduit par Edsger Dijkstra [Dij65], permet d'assurer que l'exécution d'une portion de code manipulant une ressource partagée (communément appelée *section critique*) se fera toujours de manière exclusive (*propriété de sureté*). De plus, tout nœud souhaitant la ressource partagée y accédera en temps fini (*propriété de vivacité*).

Pour ce faire, trois états sont définis dans le comportement de chaque nœud :

- état demande de section critique ;
- état en section critique ;
- état sortie de section critique.

Plusieurs solutions pour la résolution du problème d'exclusion mutuelle ont été proposées dans la littérature. Parmi celles-ci, on peut citer [Lam78, LL77, Ray91, RA81].

Algorithmes d'élection

Ces algorithmes ont pour but d'élire un nœud du système parmi d'autres [LL77]. Donc, à partir d'une configuration initiale où tous les nœuds actifs sont candidats, le système atteint une configuration où un nœud est déclaré élu et les autres candidats sont déclarés battus. Le nœud élu est appelé *leader* et il sera doté d'un certain nombre de privilèges dépendant des objectifs prédéfinis pour le fonctionnement global du système.

L'algorithme d'élection possède ainsi les propriétés de sureté et de vivacité :

- parmi tous les candidats, un seul nœud élu (*sureté*) ;
- parmi tous les candidats, un nœud doit être élu en temps fini (*vivacité*).

Le processus ou l'algorithme d'élection peut être déclenché par un nœud quelconque mais aussi éventuellement par plusieurs nœuds.

Algorithmes de détection de terminaison

Dans un environnement distribué dépourvu de structures de contrôle centrales, comment un nœud peut-il prendre conscience de la terminaison de l'application d'un algorithme sur le système ? L'objectif des algorithmes de détection de terminaison consiste à vérifier une propriété de stabilité globale (*la terminaison*). Pour ce faire, deux conditions doivent être remplies :

- détecter que chacun des processus est à l'arrêt ;
- vérifier que le travail accompli soit conforme au calcul voulu.

Une solution classique à ce problème de détection de terminaison consiste à construire une structure particulière et de déterminer un type de parcours sur cette structure. Par exemple dans [DFvG83], un anneau unidirectionnel est utilisé avec un jeton circulant pour parcourir la structure ; tandis que les auteurs de [DS80] proposent d'utiliser un arbre couvrant pour la vérification de la propriété de terminaison.

1.1.5 Notions de panne et de tolérance aux pannes

Dans cette section, nous décrivons les notions de pannes et de tolérance aux pannes dans les systèmes distribués.

Les systèmes informatiques, comme probablement tous les systèmes physiques, peuvent être victimes de défaillances. En particulier, les systèmes distribués peuvent être constitués d'un nombre important de composants collaborant pour l'accomplissement de la tâche globale d'un système. De ce fait, il existe une probabilité que certains de ces composants soient sujets à des dysfonctionnements. En effet, un composant peut tomber en panne

suite à une défaillance matérielle ou logicielle (un bug par exemple). De plus, avec la nature dynamique de certains systèmes distribués, des nœuds peuvent se connecter et se déconnecter du système à tout moment. En outre, les canaux de communication peuvent également être non fonctionnels sur une période donnée.

Tous ces aspects peuvent occasionner des perturbations ou dysfonctionnements appelés aussi pannes du système.

Définition 1.12. ***Panne***
Une panne est définie comme étant une défaillance temporaire ou permanente d'un ou de plusieurs composants d'un système [ALR04, Lap88]. Un composant est défaillant lorsqu'il ne remplit plus ses spécifications requises.

La panne est temporaire ou encore transitoire ou intermittente lorsqu'elle est présente pour une durée limitée, c'est-à-dire sa durée est inférieure au temps d'exécution de l'algorithme. Par contre, une panne est permanente ou définitive lorsque sa durée est supérieure au temps d'exécution de l'algorithme.

Une panne ou *défaillance* est ainsi l'évènement qui survient lorsque le comportement du système dévie de sa fonction. L'état du système qui est susceptible d'entrainer une défaillance est appelé *erreur*. Un état d'erreur désigne un état anormal du système et résultant de l'activation d'une *faute*. La chaîne qui lie ces différents mécanismes est illustrée sur la Figure 1.4 [ALR04, Lap88].

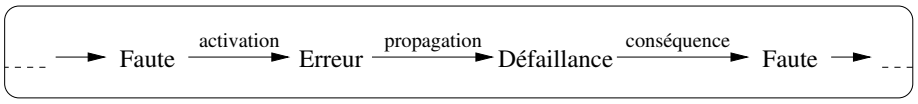


FIGURE 1.4 – La chaîne fondamentale des entraves

Les flèches de la chaîne expriment la relation causale entre fautes, erreurs et défaillances. Elles doivent être interprétées de façon générique : par propagation, plusieurs erreurs sont généralement générées avant qu'une défaillance ne survienne.

Ces notions (faute, erreur et défaillance) sont toutefois tributaires du système considéré. En effet, un même événement pourra être défini comme une défaillance, une erreur ou une faute suivant le point de vue fonctionnel considéré.

Il est à noter également que nombre d'erreurs n'affectent pas l'état externe du système, et donc ne causent pas de défaillance (panne) [Lap88].

Étant donné que les environnements distribués sont sujets à des pannes, il est nécessaire de mettre en place des mécanismes de résistance à d'éventuels dysfonctionnements pouvant survenir au cours de l'accomplissement de la tâche globale du système. On introduit ainsi la notion de tolérance aux pannes.

Définition 1.13. ***Tolérance aux pannes***
La tolérance aux pannes est un mécanisme permettant d'assurer le bon fonctionnement du système et de remplir les spécifications requises malgré la présence de dysfonctionnement dans ses composants [ALR04, Lap88, LA12].

Afin de tolérer les pannes qui peuvent survenir dans un système distribué, deux approches sont proposées : les algorithmes robustes et les algorithmes auto-stabilisants.

Les algorithmes robustes sont l'une des catégories de solutions apportées au problème de pannes dans les systèmes distribués.

Définition 1.14. ***Algorithme robuste***
Un algorithme robuste est un algorithme qui est capable de garantir le respect des spécifications du système malgré l'occurrence de pannes.

Dans [MW87], les auteurs ont montré que l'étude des algorithmes robustes peut se réduire à l'étude du problème de décision commune connu aussi sous le nom de problème du consensus : tous les processus d'un système doivent tomber d'accord sur le choix d'une valeur binaire, celle-ci devant être la valeur initiale d'au moins un processus.

Les auteurs de [FLP85] ont toutefois prouvé que le consensus est impossible dans le cas d'un système asynchrone. En effet, ils démontrent que sans hypothèse supplémentaire sur le modèle, le problème du consensus n'admet aucune solution déterministe lorsqu'une panne sur un composant peut survenir. Ainsi, ces auteurs ([FLP85]) supposent qu'un protocole fiable termine en un nombre fini d'étapes. Toutefois, dans [BT85], les auteurs rejettent cette hypothèse en proposant deux algorithmes qui peuvent ne jamais se terminer.

D'autres part, des solutions basées sur la détection des pannes pouvant apparaître dans le système ont été proposées. C'est le cas par exemple dans [CT91] où les auteurs proposent la notion de détecteurs de pannes non fiables afin de résoudre le problème du consensus.

Les algorithmes auto-stabilisants constituent l'autre catégorie de solutions apportées au problème de pannes dans les systèmes distribués.

Définition 1.15. ***Algorithme auto-stabilisant***
Un algorithme auto-stabilisant est un algorithme qui est capable de converger, au bout d'un temps fini, vers un état stable et légitime quel que soit son état initial considéré.

Le concept d'auto-stabilisation a été présenté par Dijkstra en 1974 dans [Dij74]. Il considère un système comme étant auto-stabilisant, si quel que soit son état, il est capable

de retrouver un état légal, ou légitime, sans intervention extérieure en un nombre fini d'étapes.

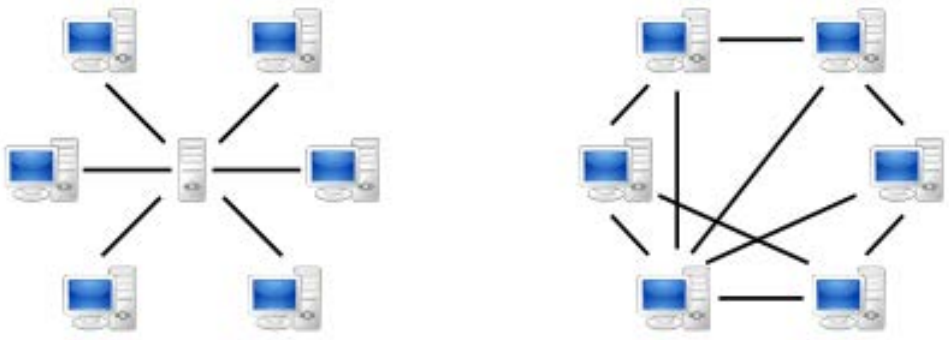
L'auto-stabilisation a été développée pour permettre aux systèmes distribués de résister aux pannes transitoires. Celles-ci peuvent être provoquées par des ruptures de liens de communication ou aussi par des corruptions de données locales. Le principe de l'auto-stabilisation est d'assurer qu'un système finira par adopter un comportement qui respecte les spécifications du problème, après avoir subit une défaillance transitoire.

Un algorithme est ainsi qualifié d'auto-stabilisant, s'il présente les deux propriétés suivantes [AG94, Gou95] :

- La **convergence** qui garantit qu'à partir d'un état quelconque, le système retrouvera au bout d'un temps fini un comportement correct ;
- La **clôture** qui garantit qu'à partir d'un état correct atteint, le système restera dans cet état correct si aucune panne n'apparaît dans le système.

1.1.6 Les modèles d'architecture

Le modèle architecture décrit la manière dont les composants d'un système sont interconnectés et communiquent pour réaliser un objectif commun. Les modèles d'architecture couramment utilisés dans les systèmes distribués sont le modèle client-serveur et le modèle pair-à-pair [CDK05].



(a) Modèle client-serveur (b) Modèle pair-à-pair

FIGURE 1.5 – Modèle client-serveur et modèle pair-à-pair

Le modèle client-serveur

Le modèle client/serveur est un modèle constitué essentiellement de deux types d'entités : l'entité centrale, qualifiée de passive appelée *serveur* et l'entité périphérique, qualifiée d'active appelée *client* (Figure 1.5(a)).

Les clients se connectent au serveur afin d'accéder aux services offerts par ce dernier. Le serveur est en général doté de capacités supérieures à celles des clients en termes de puissance de calcul, de ressources mémoires, etc.

La nature centralisée des ressources fait que modèle client/serveur est très sensible aux pannes et généralement inapte à passer à l'échelle. En effet, l'élément central (le serveur) est un point critique car s'il tombe en panne, l'ensemble de son service devient inaccessible. D'autre part, le serveur doit être en mesure d'accepter un grand nombre de connexions simultanément, ce qui implique un débit important. De plus, pour traiter toutes les demandes, le serveur doit avoir une puissance de calcul suffisamment importante. Tous ces paramètres rendent le passage à l'échelle plus difficile.

Le modèle pair-à-pair

Le modèle pair-à-pair connu également sous le nom de modèle poste-à-poste ou d'égal-à-égal est un modèle où chaque entité du système peut à la fois jouer le rôle de client et de serveur d'où l'appellation de *servent* ou simplement pair (Figure 1.5(b)).

Contrairement au modèle client-serveur, le modèle pair-à-pair présente une infrastructure décentralisée. En effet, il permet aux nœuds du système de communiquer, de collaborer et de partager des ressources entre eux.

Ce modèle pair-à-pair connait un très grand succès depuis la fin des années 90, époque à laquelle Napster [SGG01], un système pair-à-pair de partage de fichiers permit à des millions d'usagers connectés à Internet de télécharger et de partager des fichiers multimédias.

Les premières applications de ce modèle étaient exclusivement liées au partage de fichiers dont certains étaient soumis à des droits d'auteur. Par la suite, ce modèle s'est ouvert à d'autres horizons tels que :

- Le **stockage de données** où l'on trouve des plateformes qui proposent à des usagers une infrastructure robuste pour le stockage et l'accès à leurs données personnelles depuis n'importe quel point d'accès à Internet. Parmi ces plateformes on peut citer OceanStore [KBC+00] qui est l'un des premiers travaux décrivant une architecture complète de stockage persistant en pair-à-pair déployée à l'échelle d'Internet. Plus récemment, la plateforme Wuala [MBM12] a été aussi déployée à l'échelle d'Internet.
- La **communication** où des solutions de communication et de voix sur IP qui permettent à chacun d'échanger et de dialoguer avec ses connaissances indépendamment

de tout opérateur téléphonique et quel que soit son emplacement. Skype [BS06] est une des applications offrant ce type de services.

- Le **calcul distribué** où la puissance de calcul résultant de l'agrégation des processeurs de machines connectées repousse les limites des super-calculateurs actuels. On peut notamment citer l'exemple d'Ourgrid [ACGC05] qui est une plateforme pair-à-pair pour partager les cycles de calcul à travers diverses sociétés ou organismes.

Le modèle pair-à-pair se trouve actuellement de plus en plus utilisé. En effet, permettant de décentraliser la réalisation des services et de mettre à disposition des ressources partagées dans un réseau, ce modèle présente de nombreux avantages tels que l'auto-organisation, la tolérance aux pannes, le passage à l'échelle, etc.. Effectivement, l'absence d'élément de stockage central permet de mieux pallier aux pannes. De plus, la répartition des ressources entre les nœuds du système favorise un meilleur équilibrage du trafic sous-jacent. Enfin, la mutualisation des ressources induit une réduction des coûts liés à l'achat et à la maintenance des équipements.

En raison de toutes ces propriétés, nous nous intéressons dans nos travaux de recherche à ce modèle pour les nombreux avantages qu'il offre. Ainsi, la deuxième partie de ce chapitre sera consacrée entièrement à l'étude des systèmes pair-à-pair.

1.2 Les systèmes pair-à-pair

Dans cette section, nous donnons d'abord un ensemble de définitions et de terminologies qui spécifient les systèmes pair-à-pair. Ensuite, nous présentons leurs caractéristiques. Enfin, nous passons en revue les différentes architectures des systèmes pair-à-pair.

1.2.1 Définition et terminologies

Définition

Plusieurs définitions du modèle pair-à-pair ont été proposées dans la littérature [ATS04, LCP+05, MKL+02, Sch01]. Nous résumons ces définitions comme suit :

Définition 1.16. ***Système pair-à-pair***
Un système P2P (pair-à-pair) est un modèle système distribué collaboratif constitué d'un ensemble de participants appelés pairs. Un pair est une entité ou nœud du système pair-à-pair qui peut prendre successivement ou simultanément le rôle de client (lorsqu'il demande une ressource dans le système) et de serveur (lorsqu'il offre une ressource dans le système). Le pair partage ainsi des ressources informatiques accessibles de manière directe par les

autres pairs du système, dans le but d'établir un service collaboratif. Cette décentralisation des ressources du système permet de mieux résister aux pannes et favorise également le passage à l'échelle.

Terminologies

Dans ce qui suit, les notations relatives aux systèmes pair-à-pair et utilisées dans ce mémoire sont détaillées dans la terminologie suivante.

Dans un système pair-à-pair, chaque pair crée des connexions vers un ensemble de pairs, en utilisant les services de télécommunications disponibles localement. Les pairs vont donc établir des connexions entre eux en utilisant les protocoles du système P2P pour former ce que l'on appelle un *réseau pair-à-pair*.

Le réseau P2P ainsi formé possède ses mécanismes de nommage, d'adressage, de communication, de routage, etc. Ces mécanismes sont en général distincts du réseau de télécommunication utilisé. On parle à cet effet de *réseau logique* en superposition (appelé par analogie en anglais : *overlay*) pour insister sur le fait que le réseau pair-à-pair devient alors distinct du *réseau physique* de télécommunication en-dessous (appelé par analogie en anglais : *underlay*).

Un réseau logique est donc l'interconnexion qui relie virtuellement les nœuds d'un système P2P au-dessus d'un réseau physique. C'est alors l'appartenance à ce réseau logique qui définit les limites d'un système pair-à-pair.

Un lien entre deux pairs du système P2P symbolise une connexion virtuelle permettant la communication entre ces deux noeuds. Ce lien est souvent appelé lien virtuel ou logique. La communication entre deux pairs peut passer par un ou plusieurs liens physiques du réseau de télécommunications sous-jacent. Ainsi, analogiquement, les liens dans le réseau de télécommunication sont appelés liens réels ou physiques.

Les réseaux pair-à-pair sont dynamiques, c'est-à-dire caractérisés par l'arrivée et le départ continu de nœuds au sein du système. Le terme *churn* désigne la dynamique d'un environnement pair-à-pair. Il est caractérisé par le taux d'arrivée et de départ de nœuds au sein du système pendant une période donnée.

1.2.2 Caractéristiques du pair-à-pair

Dans cette section, nous passons en revue les caractéristiques générales des systèmes pair-à-pair. La synthèse des travaux définis dans [RD10, JEB05, MKL+02], nous permet ainsi de caractériser les systèmes pair-à-pair comme des systèmes distribués présentant les propriétés suivantes :

Décentralisation. Les pairs prennent le rôle de client et de serveur, et la majeure

partie de l'état du système et des tâches sont dynamiquement distribués parmi les pairs. L'ensemble des besoins de calcul, de stockage et de communication liés à l'exécution du système est alors fourni de façon collaborative par les membres du système pair-à-pair. Toutefois, selon l'architecture considérée, certains nœuds peuvent posséder un état centralisé en jouant un rôle d'annuaire, mais, dans tous les cas, les ressources restent toujours distribuées entre les pairs du système.

Auto-organisation. Elle définit le processus dans lequel l'organisation d'un système évolue spontanément sans pour autant que cela soit contrôlé par un système extérieur [MKL+02]. Dans un système P2P, les pairs peuvent s'organiser et se regrouper, selon les activités dont ils se chargent, leurs intérêts communs ou encore la topologie du réseau, comme par exemple, l'établissement d'un réseau logique virtuel structuré ou non-structuré ou hiérarchique.

Dynamicité. Les systèmes P2P sont de nature dynamique. Un pair peut se connecter et de se déconnecter du réseau à n'importe quel moment. Les départs de nœuds peuvent être volontaires, dans le cas par exemple d'un service accompli ou bien involontaires, quant il s'agit de nœuds défaillants ou de nœuds mobiles quittant une zone de couverture ou le cluster. Rappelons que la dynamique des environnements pair-à-pair est aussi désignée sous le terme de *churn*.

Réseau virtuel. Les nœuds d'un système pair-à-pair forment souvent un réseau virtuel ou logique. Ce réseau présente généralement des propriétés de transparence vis-à-vis des nœuds du système. En effet, les nœuds sont vus comme des pairs dans l'*overlay* alors qu'ils peuvent être de types différents sur le plan matériel, par exemple des ordinateurs portables, des stations de travail, des clusters, etc.

Large échelle. Un système pair-à-pair est dit *large échelle* ou *grande échelle* s'il est composé d'un très grand nombre de nœuds. En effet, un système pair-à-pair peut atteindre une taille très importante de participants, de l'ordre de plusieurs milliers de nœuds. Généralement, ces nœuds sont géographiquement répartis.

Passage à l'échelle. Le fait d'augmenter de manière importante le nombre de nœuds dans un système pair-à-pair n'affecte généralement pas le fonctionnement de l'ensemble du système. De plus, la répartition des ressources entre les nœuds du système favorise un meilleur équilibrage du trafic sous-jacent.

Tolérance aux pannes. Les systèmes pair-à-pair sont généralement résistants aux pannes, car il existe peu, ou pas, de nœuds dont la présence est critique pour l'exécution du service associé.

1.2.3 Architectures de réseaux pair-à-pair

Depuis leur émergence à la fin des années 90, les systèmes pair-à-pair ont beaucoup évolués et se sont diversifiés dans leurs architectures. Ces architectures reposent sur la construction d'un réseau virtuel sur le réseau physique (typiquement Internet).

Plusieurs propositions de classification d'architectures P2P ont été faites dans la littérature [AH01, MKL+02, Ora01, Sch01]. D'ordinaire, c'est le degré de décentralisation qui est retenu pour la classification des architectures P2P. Tout comme les propositions faites dans [AH01, Ora01], nous distinguons aussi, en fonction du degré de décentralisation, trois modèles d'architectures P2P à savoir le modèle centralisé, le modèle pur ou décentralisé et le modèle hybride ou hiérarchique.

En outre, en fonction du mode d'organisation des pairs, le modèle d'architecture P2P décentralisé peut être scindé en deux classes : le modèle décentralisé non-structuré et le modèle décentralisé structuré.

La Figure1.6 présente la taxonomie des architectures P2P.

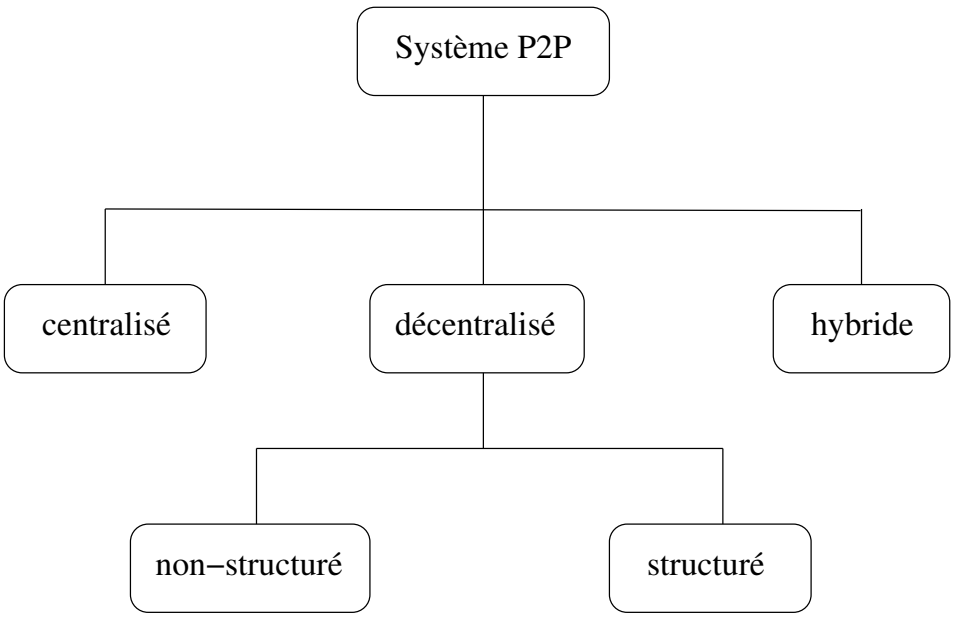


FIGURE 1.6 – Taxonomie des architectures P2P

Nous classifions ces modèles d'architectures principalement en trois générations.

- Première génération qui correspond à la définition du modèle d'architecture P2P centralisé.
- Deuxième génération qui correspond à la définition du modèle d'architecture P2P décentralisé. Dans cette génération on distingue :

  - le modèle d'architecture P2P décentralisé non-structuré ;
  - le modèle d'architecture P2P décentralisé structuré.
- Troisième génération qui correspond à la définition du modèle d'architecture P2P hybride.

Nous allons, dans les sections qui suivent, décrire ces différentes générations d'architectures et pour chacune d'elles, exposer les avantages, les inconvénients ainsi que quelques protocoles P2P qui l'implémentent. Nous trouvons dans [ATS04, LCP+05] un survol plus exhaustif des protocoles P2P.

Architecture P2P centralisée

La première génération de réseaux pair-à-pair est l'architecture centralisée qui est un peu similaire à l'architecture client/serveur. Dans ce modèle, un serveur central joue le rôle d'annuaire en indexant les pairs connectés ainsi que les informations sur leur ressources partagées. Lorsqu'un pair se connecte dans un tel système, il envoie une copie de son index au serveur central. Quand le serveur central reçoit une requête qui est émise depuis un pair, il la traite puis retourne au pair émetteur une liste des pairs qui contiennent l'information recherchée. Par la suite, ce dernier (le pair émetteur) contacte directement les pairs qui possèdent les fichiers correspondant aux critères de recherche et les télécharge.

Le système le plus populaire qui rentre dans la catégorie des systèmes pair-à-pair centralisés est Napster [SGG01].

En centralisant les index, ce type d'architecture rend les algorithmes de recherche exhaustive particulièrement efficaces, en minimisant les communications. De plus, cette architecture est facile à mettre en place. Cependant, le serveur central est un point faible du système. En effet, il doit être en mesure d'accepter un grand nombre de connexions simultanément, ce qui implique un débit important. Faute de quoi, il peut engendrer un goulot d'étranglement. De plus, la gestion centralisée des requêtes rend le système fragile vis-à-vis d'une panne du serveur et pose des problèmes de passage à l'échelle.

Architecture P2P décentralisée

La deuxième génération de réseaux pair-à-pair correspond à des architectures décentralisées qui ne s'appuient sur aucun serveur. Ainsi, le modèle d'architecture ici est purement décentralisée, il n'existe donc pas la notion d'index centralisé. Les pairs ont tous les mêmes responsabilités et chacun peut se comporter en tant que client ou serveur en même temps.

En fonction du mode d'organisation des pairs, on distingue les architectures décentralisées non-structurées et les architectures décentralisées structurées.

a) Architecture P2P non-structurée

Dans les systèmes pair-à-pair non-structurés, le réseau *overlay* est construit d'une manière non-déterministe (ad hoc). La topologie qui en résulte est donc un graphe aléatoire.

Pour rejoindre le réseau, un pair doit connaître l'adresse d'un autre pair déjà connecté et qui sert alors de nœud d'insertion appelé aussi *nœud point d'entrée* ou nœud d'amorçage (de l'anglais *bootstrap*). Par ce noeud, le nouveau pair entrant découvre progressivement d'autres noeuds du réseau et établit des liens avec eux, selon un algorithme propre au protocole, pour ensuite y propager les messages.

D'autre part, le placement des données est complètement indépendant de la topologie du réseau *overlay*. Il n'y a aucune restriction sur la manière de décrire la ressource désirée (expressivité des requêtes). Cela veut dire qu'on peut utiliser la recherche par mots-clés, ou des requêtes par intervalles, ou d'autres approches.

Les approches les plus souvent utilisées pour la recherche ou découverte de ressources dans un réseau pair-à-pair non-structuré sont l'inondation et la marche aléatoire.

L'inondation (de l'anglais *flooding*) est la première approche de recherche proposée sur ce type de réseau. La recherche par inondation propose de retransmettre récursivement la requête de recherche à tous les voisins d'un pair (sauf celui dont il a reçu la requête) jusqu'à la localisation du service ou l'expiration du TTL. Le champ TTL est en fait une valeur associée au message de recherche pour comptabiliser le nombre de retransmissions restantes. Quand celui-ci est nul, alors le message n'est plus renvoyé.

Gnutella 0.4 [Sol01] est un exemple de protocole P2P non-structuré fonctionnant selon le principe d'inondation.

La marche aléatoire (de l'anglais *random walk*) [LCC+02] a été proposée dans le but d'améliorer le coût de l'inondation. La recherche par marche aléatoire consiste à retransmettre récursivement la requête de recherche à un unique voisin choisi aléatoirement jusqu'à la localisation du service ou l'expiration du TTL.

Des améliorations de cette approche ont été proposées dans la littérature. On peut citer par exemple le protocole Gia [CRB+03] qui utilise la technique de marche aléatoire biaisée (*biased random walks*). Dans cette approche, la requête de recherche est retransmise en choisissant de manière déterministe le voisin qui a le plus fort degré.

Les réseaux non-structurés sont particulièrement appréciés pour leur résistance à la dynamique des environnements pair-à-pair. En effet, puisque les pairs ont les mêmes responsabilités (chacun est client et serveur à la fois), la déconnexion de certains d'entre eux n'empêchera pas au système de continuer à fonctionner. Ceci confère à ce type de réseau une résistance aux pannes relativement élevée. De plus, l'absence de contraintes liées à une structure quelconque permet de relâcher certaines tâches complexes, comme le maintien d'index ou de structures de données réparties.

Cependant, ce modèle présente des inconvénients non moins négligeables. En effet, les mécanismes de recherche, telle que l'inondation, sont très coûteux en termes de trafic et ainsi de charge qu'ils font subir au réseau. De plus, un tel système ne garantit pas des recherches fiables puisque les requêtes sont limitées par le TTL.

C'est en réponse à ces problèmes que les réseaux pair-à-pair décentralisés structurés sont apparus.

b) Architecture P2P structurée

Contrairement aux réseaux pair-à-pair non-structurés, les réseaux pair-à-pair structurés proposent d'utiliser la théorie des graphes pour imposer une organisation spécifique des nœuds et de leurs interconnexions. La topologie qui en résulte est donc un graphe structuré [ATS04, LCP+05].

Les systèmes P2P structurés reposent sur le principe que si les services de recherche centralisés sont basés sur des associations (*clé, objet*), il est possible de décentraliser cette indexation en donnant la responsabilité de chaque clé à un identifiant de nœud du réseau. Pour atteindre cet objectif, la plupart de ces systèmes utilisent des tables de hachage distribuées (de l'anglais « *Distributed Hash Tables* » en abrégé DHT) [ATS04, LCP+05].

Une DHT est en fait un mécanisme permettant l'identification et l'obtention d'une information dans un système distribué. Elle associe par hachage une clé (ou identifiant) à chaque pair ainsi qu'à chaque ressource dans le système. Donc les pairs et les objets (ressources) sont identifiés dans un même espace de nommage. L'ensemble de la table de hachage est réparti entre les différents pairs du système. Chaque nœud (pair) est responsable de l'indexation des objets dans le système dont l'identifiant est le plus proche de sa clé selon la métrique de distance utilisée.

Les DHTs fournissent deux primitives de base à savoir *put* ou *store* et *get* ou *lookup*. Ainsi, *store(clé, valeur)* permet de stocker une clé et sa valeur associée et *lookup(clé)* permet de récupérer la valeur associée à la clé.

Lorsqu'un nœud souhaite partager un objet (ressource) dans un tel système, il le publie. La publication consiste à calculer la clé associée à l'objet, puis à envoyer un message au nœud responsable de la clé à travers le réseau *overlay*. À la réception du message, le nœud responsable de la clé met à jour sa table.

Lorsqu'un nœud cherche un objet (une ressource) associé à une clé, il cherche directement la clé correspondante. L'envoi d'un message vers une clé permet donc d'atteindre le nœud responsable de cette clé et donc de la ressource. Quand le nœud responsable est trouvé, ce dernier répond au nœud source et envoie les éventuelles informations relatives à l'objet. Notons que la recherche basée sur la clé (en anglais : *Key Based Routing*) dans les DHTs est appelée recherche exacte.

Il existe plusieurs protocoles ou implémentations de réseaux pair-à-pair structurés basés sur les DHTs. Parmi ceux-ci, on peut citer Chord [SMK+01], Pastry [RD01], Kademlia [MM02], CAN [RFH+01].

On distingue différentes topologies basées sur ces DHTs. L'anneau, l'arbre et l'hypercube sont les topologies les plus implémentées.

La Figure1.7 illustre le schéma d'une architecture P2P décentralisée structurée en anneau (il s'agit ici de l'anneau de Chord [SMK+01]).

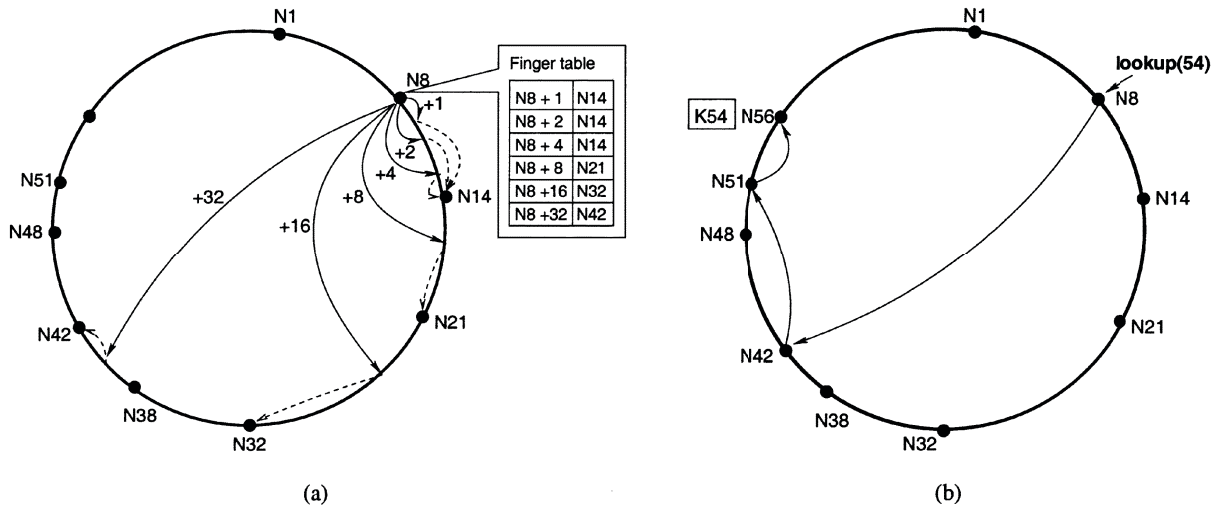


FIGURE 1.7 – Un exemple d'architecture pair-à-pair décentralisée structurée [SMK+01].

La recherche sur les réseaux structurés est plus rapide et plus efficace que sur les systèmes non-structurés. En effet, la recherche s'opère en temps logarithmique pour la plupart des systèmes P2P structurés. De plus, le faible diamètre de la DHT ainsi que la taille logarithmique des tables de routage permettent à ces systèmes une bonne propriété de passage à l'échelle. En outre, la nature distribuée de la table de hachage entre les différents pairs confère à ce type de système une certaine robustesse face aux pannes. Des mécanismes de redondance sont souvent mis en place pour éviter la perte définitive ou l'inaccessibilité des ressources.

Cependant, l'inconvénient des DHTs utilisées est celui de la "recherche exacte". En effet, l'expression de recherche sur des mots-clés n'est pas possible dans ce type de système. D'autre part, ce type de système nécessite un protocole assez lourd pour la maintenance de la structure de topologie.

Architecture P2P hybride

La troisième génération de réseaux pair-à-pair correspond à des architectures hybrides qui associent à la fois l'architecture centralisée et celle décentralisée.

Dans ce type d'architecture certains pairs spécifiques, appelés *super-nœuds* ou *super-pairs* (en anglais *ultrapeers*) vont avoir certaines responsabilités dans le système. En effet, les super-pairs effectuent les fonctions complexes comme le traitement des requêtes, le contrôle d'accès et la gestion des méta-données. C'est eux aussi qui assurent généralement les fonctions relatives à la localisation, au routage ou à l'organisation des pairs. Ainsi, chaque super-pair a la responsabilité d'indexer et de contrôler l'ensemble des pairs auxquels il est rattaché au système. Les pairs rattachés aux super-pairs sont souvent appelés *nœuds ordinaires* ou *nœuds simples* ou *nœuds feuilles* (en anglais *leaves*).

Le choix des super-pairs dépend généralement des besoins du système P2P à mettre en place et varie ainsi d'une application à une autre. Par exemple, Skype [BS06, GD06] utilise les super-pairs pour la découverte et la localisation des pairs. Un nœud (pair) peut être nommé super-pair suivant plusieurs critères : cela peut dépendre de ses ressources matérielles (puissance de calcul, capacité mémoire, etc.), de sa bande passante, de l'instant depuis lequel il est connecté au réseau (son degré de stabilité), de sa fiabilité, de son système d'exploitation, etc. [SR01]. Les super-pairs peuvent ainsi être élus dynamiquement (se basant sur certains critères) et remplacés en cas de pannes.

Dans son fonctionnement, chaque nœud ordinaire se rattache à un ou plusieurs super-pairs. Les super-pairs sont connectés entre eux sur le niveau haut de la hiérarchie, suivant le modèle de l'architecture pair-à-pair décentralisée.

Les ressources partagées par un nœud ordinaire sont indexées par son ou ses super-pairs responsables. Ainsi, lorsqu'un nœud ordinaire recherche une ressource, il envoie sa requête à son ou ses super-pairs.

À la réception d'une requête qui est émise depuis un nœud ordinaire, le super-pair la traite localement en consultant son annuaire. Si la ressource recherchée est trouvée, le super-pair retourne le résultat au nœud émetteur. Ce dernier contacte directement le pair qui possède la ressource et l'échange ou le transfert entre eux peut commencer. Si par contre, la ressource n'est pas trouvée localement, le super-pair retransmet la requête à un ou plusieurs autres super-pairs selon l'algorithme dépendant du modèle d'architecture P2P décentralisé implémentée.

Les systèmes Gnutella v0.6 [KM02] et FastTrack [Fas03], sont des exemples de réseau pair-à-pair de cette génération.

La Figure1.8 illustre un schéma d'une architecture pair-à-pair de troisième génération.

Cette génération d'architecture présente les avantages des deux modèles précédents : la tolérance aux pannes et la réduction du trafic des requêtes et du temps de recherche

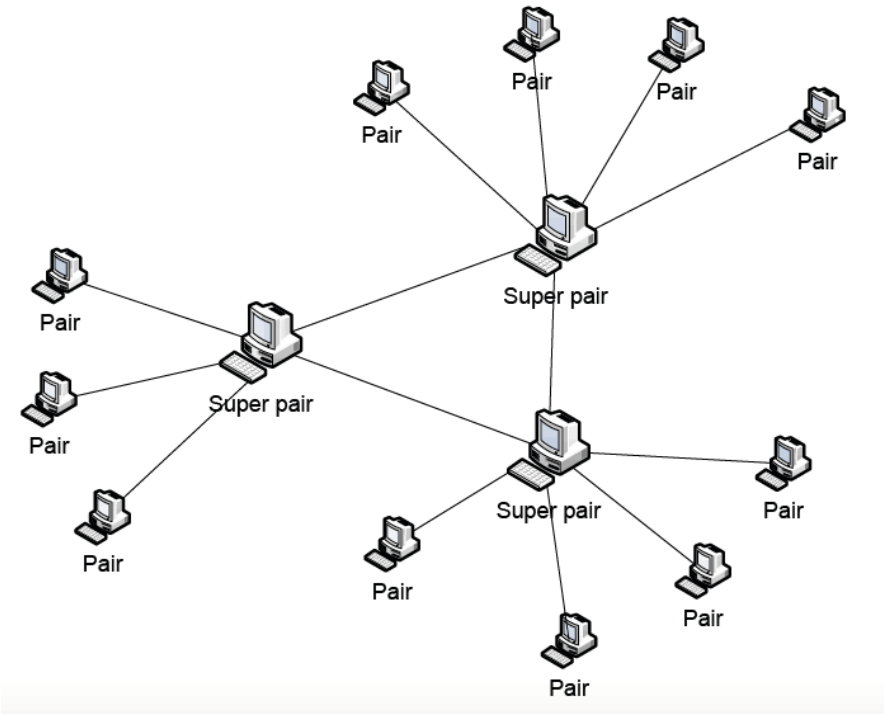


FIGURE 1.8 – Architecture pair-à-pair hybride

vu que les nœuds ordinaires ne participent pas ce processus.

Toutefois, ce type de système est plus complexe à mettre en œuvre. De plus, le choix optimal des super-pairs n'est pas trivial, plusieurs critères sont à définir selon les besoins de l'application.

1.3 Conclusion

Dans ce chapitre nous avons présenté le contexte dans lequel se placent nos travaux, à savoir les systèmes distribués. Ainsi nous avons présenté ses propriétés inhérentes, détaillé ses modèles de communication et le fonctionnement de quelques algorithmes classiques. Ensuite, nous avons décrit les notions de pannes et la tolérance aux pannes ainsi que les modèles d'architectures des systèmes distribués.

Dans la deuxième partie de ce chapitre, nous avons abordé l'étude des systèmes pair-à-pair qui nous intéressent particulièrement dans nos travaux de recherche. De ce fait, nous avons d'abord défini et présenté les caractéristiques de ces systèmes. Par la suite, nous avons décrit les différentes architectures des systèmes pair-à-pair.

Les systèmes pair-à-pair offrent de nombreux avantages grâce à leurs propriétés fonda-

mentales et inhérentes. C'est pour ces raisons qu'ils sont utilisés dans plusieurs domaines. Ainsi, le prochain chapitre est consacré à l'étude des services dans différents environnements (ou plateformes d'exécution) ; notamment dans les grilles pair-à-pair.

Chapitre 2

Les services dans les systèmes distribués

Résumé. *Dans ce chapitre, nous étudions les services ainsi que leurs environnements d'exécution. Ainsi, dans un premier temps, nous présentons les généralités sur les services. Après avoir défini la notion de service, nous présentons l'architecture orientée service. Ensuite, nous décrivons les Services Web puis les grilles de services. Dans un second temps, nous étudions les solutions de gestion de services dans un environnement de grille. Par la suite, nous exposons notre problématique de recherche.*

Sommaire

2.1 Généralités sur les services 32
2.1.1 Notion de service 32
2.1.2 Approche conceptuelle orientée service (SOA) 33
2.2 Services Web 36
2.2.1 Contexte 36
2.2.2 Définitions de la notion de Service Web 37
2.2.3 Les technologies des Services Web 38
2.3 Services de grilles informatiques 44
2.3.1 Concept de grille informatique 44
2.3.2 Architecture d'une grille 47
2.3.3 Topologies de grilles 49
2.3.4 Les services de grille 50
2.4 Gestion de services dans un environnement de grille informatique à large échelle 52
2.4.1 Objectifs et motivations 53
2.4.2 Mécanismes de découverte de services dans les grilles 54
2.5 Conclusion 61

2.1 Généralités sur les services

2.1.1 Notion de service

La notion de service n'est pas seulement appliquée à l'informatique. En effet, dans la vie de tous les jours, un fournisseur offre un service à un client le consommant dans une relation de confiance établie entre les deux parties. Toutefois, il est difficile de donner une définition précise d'un service car son utilisation trouve des échos dans plusieurs domaines. Par exemple, dans le domaine des télécommunications, un opérateur fournit un service d'accès à Internet, un service de messagerie SMS, etc.

Dans le domaine de l'informatique, un hébergeur fournit un service de stockage de données, un système d'exploitation fournit plusieurs services d'abstractions du matériel. Nous distinguons également le modèle des Services Web qui, avec l'affirmation du Web grâce à des technologies accessibles à tous les acteurs, a le plus alimenté le débat autour des services. À cet effet, la Section 2.2 sera entièrement consacrée à l'étude des Services Web. D'autres part, des environnements tels que les systèmes pair-à-pair, les *Cloud* [AFG+10] ou encore les grilles informatiques [FKT01], offrent également l'accès à des services de natures diverses.

Dans le cadre des systèmes pair-à-pair, la notion de service peut prendre plusieurs formes. Le partage de fichiers constitue une des premières applications offertes dans les environnements P2P tels que par exemple, Napster [SGG01], Bittorrent [Coh02, Coh03], etc. Par la suite, avec les besoins diversifiés du monde de l'informatique (communauté scientifique, industriels, etc.) et voulant tirer profit des propriétés fondamentales et inhérentes des systèmes pair-à-pair telles que l'auto-organisation, la tolérance aux pannes, le passage à l'échelle, le changement dynamique de topologie, etc., la notion de service s'est étendue avec succès dans d'autres domaines spécifiques tels que le calcul scientifique et le stockage de données avec par exemple OceanStore [KBC+00], ou encore la téléphonie via Internet avec notamment Skype [BS06].

Vers 2008 est apparu le concept du *Cloud* [AFG+10, MG11] désignant un modèle de service mutualisé, accessible par le réseau, et permettant à des organisations de déployer rapidement et à la demande des ressources informatiques. Ce paradigme propose trois modèles de services définissant le type de ressources et de flexibilité que l'utilisateur possède. On distingue ainsi, le service *Infrastructure as a Service* modèle dans lequel un fournisseur met à disposition de ses utilisateurs des machines virtuelles, louées pour une certaine période de temps renouvelable tacitement. L'utilisateur a donc la maîtrise de l'ensemble de la configuration logicielle des machines virtuelles qu'il loue. Il peut installer les services qu'il souhaite, sans restriction, sur ces machines. Il y a aussi le modèle *Platform as a Service*. Ici, le système d'exploitation et les outils d'infrastructure sont sous la respon-

sabilité du fournisseur. Le consommateur a le contrôle des applications et peut ajouter ses propres outils. La situation est analogue à celle de l'hébergement Web où le consommateur loue l'exploitation de serveurs sur lesquels les outils nécessaires sont préalablement placés et contrôlés par le fournisseur. Enfin, il y a le *Software as a Service*. Dans ce type de service, des applications sont mises à la disposition des consommateurs. Ces applications peuvent être manipulées à l'aide d'un navigateur web ou installées de façon locative sur un PC, et le consommateur n'a pas à se soucier d'effectuer des mises-à-jour et d'assurer la disponibilité du service.

Dans le cadre des grilles informatiques, la notion de service se manifeste sous forme de ressources de calcul et de stockage ou encore des applications mises à dispositions des utilisateurs. La Section 2.3 est consacrée à la description du modèle de services de grille puisque nous nous intéressons dans cette thèse à la gestion de services dans un tel environnement.

Définition du concept de service

Nous définissons la notion de service comme suit :

Définition 2.1. ***Service***
Un service est une entité qui peut être un objet, une donnée, une ressource, une application ou un composant logiciel ayant une fonction bien définie. Il possède une interface et est accessible directement par les utilisateurs via des requêtes. Ainsi, à partir de son interface, l'utilisateur peut accéder à une ou plusieurs ressources. Ces ressources peuvent êtres localisées sur un même serveur ou distribuées géographiquement entre plusieurs serveurs du réseau. De plus, un service peut également interagir avec d'autres services à travers le réseau dans le but de créer des services composés.

En substance, le service est une action exécutée par un composant « fournisseur » (ou serveur) à l'attention d'un composant « consommateur » (ou client aussi appelé utilisateur). En général, l'utilisateur s'intéresse uniquement au résultat produit du service sans avoir le besoin ni le souci de savoir comment ce dernier est obtenu. L'Architecture Orientée Services (SOA) suit ce même principe.

2.1.2 Approche conceptuelle orientée service (SOA)

Conceptualisée par le *Gartner Group*[1], la notion d'architecture axée sur les services notée SOA (de l'anglais *Service Oriented Architecture*) a été présentée comme une technologie promise à un bel avenir pour le développement des applications d'entreprise

1. *http ://www.gartner.com/technology/home.jsp*

[MB08]. En effet, les architectures orientées services ont émergé à la suite de CORBA [Vin97] comme des architectures de références pour la conception, le développement et l'intégration des Systèmes d'Information (SI) de nouvelle génération qui deviennent de plus en plus complexes et hétérogènes [Dav09].

En guise d'exemples, on peut citer :

- La SNCF [2] qui a mis en place une architecture de type SOA pour son système de réservation (recherche d'horaire, demande de tarif, réservation, etc.) qui prend en charge à la fois les terminaux des guichets des agences et gares, et les sollicitations de son site web de commande en ligne *www.voyages-sncf.com*.
- Le Groupe Air France-KLM [3] qui a lui aussi décidé en juillet 2008 de choisir une architecture orientée service pour son système d'information dans le but de rendre ce dernier à la fois évolutif et réactif.

Ce paradigme architectural présente ainsi un intérêt particulier pour les technologies de l'information et le domaine de l'entreprise. En effet, les SOA encouragent la rationalisation des processus et l'adaptation des systèmes informatiques aux évolutions du marché.

Les architectures reposent sur la réorganisation des applications en ensembles fonctionnels appelés *services* et l'exposition des informations nécessaires sur ces services pour qu'ils soient facilement utilisés par les clients. Chaque ressource informatique est considérée comme un service. L'objectif est donc de décomposer les différentes fonctionnalités d'un SI en un ensemble de fonctions basiques offertes par des services et de décrire finement un schéma d'interaction entre ces services. Ces applications–services peuvent être exécutées sur des plateformes hétérogènes dans un environnement distribué, et fournissent des fonctionnalités à d'autres entités (clients) [CCMN04].

Le W3C (*World Wide Web Consortium*), chargé du développement des standards sur le Web et leur évolutivité, définit la SOA comme suit :

Définition 2.2. ***Architecture Orientée Services (SOA)***
La SOA est une collection de composants logiciels, appelés services, pouvant être invoqués et dont les descriptions d'interfaces peuvent être publiées et découvertes [HB04].

La Figure 2.1 décrit l'organisation et les modalités d'utilisation de fonctionnalités distribuées pouvant être sous le contrôle de différents participants [Erl08, MLM+06].

La SOA implique trois types d'acteurs :

1. Les fournisseurs de services : un fournisseur décrit ses services, les rend publics ou autorise un accès contrôlé et peut les publier dans un ou plusieurs annuaires.

2. *http ://www.sncf.com/*
3. *http ://www.airfranceklm.com/*

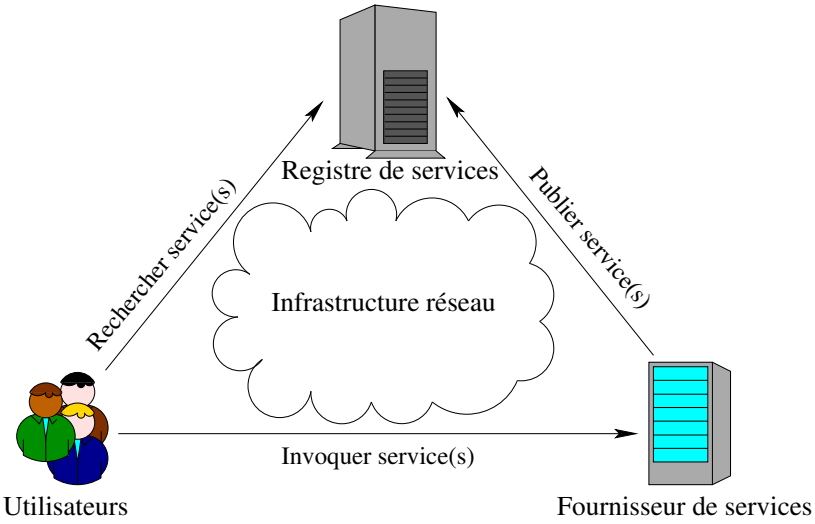


FIGURE 2.1 – Modèle d'interaction SOA

2. Les annuaires ou registres de services : un annuaire expose les descriptions des services indexés afin de permettre aux utilisateurs de connaitre pour chacun d'eux, ses fonctionnalités et sa localisation.
3. Les consommateurs ou utilisateurs de services : un utilisateur découvre le ou les services répondant à ses besoins en interrogeant un ou des annuaires. Une fois que le ou les services recherchés sont trouvés, l'utilisateur les invoque à travers les interfaces publiées par leurs fournisseurs respectifs.

L'élément clé donc de la mise en œuvre de la SOA est de rendre accessible, via un réseau (par exemple Internet), des services afin que les utilisateurs puissent y accéder et les réutiliser dans leurs applications. À cet effet, T. Erl propose dans [Erl08], un ensemble de principes de conception sous-jacents à la SOA. Nous énumérons ci-dessous ces principes.

Contrat de service. Ceci définit un accord entre le fournisseur et le consommateur, composé des éléments suivants :

- une représentation technique du service (i.e., son interface) comme les opérations, leurs paramètres d'entrée et de sortie et les contraintes sur les entrées ;
- une description informelle des opérations sous formes de règles et contraintes d'utilisation du service (par exemple, volume des données échangées) ;
- un niveau de service (*Service Level Agreement*) qui indique la qualité de service (QoS) requise (par exemple, temps de réponse maximum attendu, plages horaires d'accessibilité, etc.). Dans ce sens, une étude sur les contraintes de QoS pour l'exécution efficace d'une composition de Services Web a été mené dans [GNG+11, GNGD13].

Faible couplage entre les services. Les services doivent maintenir une relation minimisant les dépendances entre eux.

Autonomie des services. Les services contrôlent la logique d'exécution qu'ils encapsulent. Plus ce contrôle est fort, plus l'exécution d'un service est prédictible.

Abstraction de service. En dehors des différentes descriptions dans le contrat de service, les services cachent leur logique au monde extérieur. Le contrat de service ne doit contenir que les informations essentielles à son invocation.

Absence d'état des services. C'est-à-dire que l'échange qui se produit entre le client et le service doit comporter toutes les ressources requises pour traiter la requête du client et une fois cet échange terminé le service ne conserve pas de données sur celui-ci.

Services réutilisables. La logique est divisée en différents services avec comme objectif de promouvoir la réutilisation. Les services peuvent être ainsi partagés parmi différents domaines.

Services découvrables. La description des services est complétée par un ensemble de méta-données permettant leur découverte et leur interprétation de manière efficace et appropriée.

Services composables. C'est-à-dire que les services peuvent être composés pour offrir de nouveaux comportements plus complets ou satisfaire de nouveaux besoins précis.

La SOA est ainsi une réponse très efficace aux problématiques que rencontrent les systèmes informatiques en termes de réutilisabilité, d'interopérabilité et de réduction de couplage entre les différents systèmes qui implémentent leurs SI. Les SOA ont été popularisées avec l'apparition de standards comme les Services Web (cf. Section 2.2) dans le commerce électronique.

2.2 Services Web

2.2.1 Contexte

Les systèmes d'information ont évolué d'un contexte centralisé et monolithique à des environnements distribués et hétérogènes. Dans ce nouveau contexte, les intergiciels ont pris une place importante afin d'assurer l'interopérabilité entre différents systèmes d'information. Étant donné que les intergiciels sont devenus primordiaux à la vie d'une entreprise, plusieurs constructeurs et organismes de normalisation ont proposé leur propre intergiciel sans consensus universel. Ainsi, plusieurs solutions de mise en œuvre sont apparues et dont les plus répandues sont CORBA de l'OMG (*Object Management Group*) [4], DCOM

4. http ://www.omg.org/

de Microsoft [5] et RMI de Sun [6].

CORBA (*Common Object Request Broker Architecture*). C'est une norme de communication utilisée pour l'échange entre objets logiciels hétérogènes. Ces objets peuvent être écrits dans des langages de programmation distincts, exécutés dans des processus séparés, voire déployés sur des machines distinctes.

DCOM (*Distributed Component Object Model*). C'est une technologie de composants introduite par Microsoft pour le développement de composants logiciels réutilisables, orientés objet et indépendants du langage de programmation. Cependant, l'interopérabilité n'est supportée qu'entre plates-formes Windows.

RMI (*Remote Method Invocation*). Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cependant, l'interopérabilité n'est supportée qu'entre programmes Java.

Précisons que ces intergiciels traditionnels présentent un certains nombreux de limites. En effet, la même technologie doit être déployée de chaque coté pour assurer la communication entre entités. De plus, ils sont fragiles car si le partenaire fait évoluer son application, le code client peut ne plus fonctionner. En outre, les protocoles ne passent généralement pas les pare-feux. Ces différentes approches sont ainsi incapables de passer à l'échelle et elles restent donc, le plus souvent, confinées à l'intérieur des entreprises.

D'autre part, Internet est arrivé dans les entreprises avec de nouvelles exigences pour lesquelles les intergiciels traditionnels ne pouvaient pas répondre de manière satisfaisante. Afin d'assurer une interopérabilité universelle, les grands constructeurs et les organismes de normalisation tels que Microsoft, IBM, Sun, SAP, etc. se sont mis d'accord sur un ensemble de technologies, formant aujourd'hui les Services Web.

Parallèlement à l'apparition des Services Web, les techniques et langages de modélisation ont pris une place importante avec notamment la naissance d'UML (*Unified Modeling Language*) comme langage standard. Par conséquent, l'idée que l'interopérabilité ne devait pas être résolue au niveau du code, mais à un niveau d'abstraction plus élevé commençait à émerger.

2.2.2 Définitions de la notion de Service Web

Plusieurs définitions des Services Web ont été proposées dans la littérature [ACKM04, CDK+02, Dan03, G+04]. Cette prolixité montre que la notion de Service Web a besoin d'être éclaircie, et motive des travaux de recherche. Ainsi, :

5. https ://www.microsoft.com/com/default.mspx
6. https ://www.oracle.com/sun/index.html

Curbera et al. [CDK+02] définissent un Service Web comme une application accessible à partir du Web. Ce Service Web utilise les protocoles Internet pour communiquer et un langage standard pour décrire son interface.

Dans [Dan03], Jérôme Daniel définit un Services Web comme une application modulaire, indépendante, faiblement couplée et qui peut être découverte et invoquée dynamiquement via Internet ou un intranet par d'autres services. Il est défini par un ensemble de standards qui permet aux applications de faire appel à des fonctionnalités à distance en simplifiant ainsi l'échange de données et de dialoguer à travers le réseau, indépendamment de leur plate-forme d'exécution et de leur langage d'implémentation.

Selon le W3C [G+04] : *A Web Service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols.*

En résumé, nous définissons ce concept comme suit :

Définition 2.3. ***Service Web***
Un Service Web est une application ou un composant logiciel identifié par un URI (Uniform Resource Identifier)[7] *et répondant à un besoin utilisateur. Il peut être découvert grâce à sa description et invoqué via le réseau Internet ou un intranet.*

Comme pour le W3C, l'un des principaux facteurs de succès des Services Web réside dans l'utilisation d'une pile de protocoles s'appuyant sur des normes ouvertes. De plus, les services Web partagent une architecture commune qui a pour fondement le concept d'architecture orientée service. Une SOA doit se concentrer sur la façon dont les services sont décrits et organisés pour supporter leur découverte et leur utilisation. Pour cela, des technologies telles que WSDL [CMRW07], UDDI [CHvR+04] et SOAP [GHM+03] pour respectivement la description, la découverte et l'invocation de services, ont été proposées. Ces technologies s'appuient sur XML pour garantir l'interopérabilité.

Soulignons que d'autres technologies comme XML-RPC [LJDW01] et REST [Fie00] sont aussi utilisées dans le cadre des Services Web.

2.2.3 Les technologies des Services Web

Dans cette section, nous décrivons les technologies généralement utilisées pour mettre en œuvre des Services Web.

7. Un URI est une courte chaîne de caractères identifiant une ressource sur un réseau.

XML (*eXtensible Markup Language*)

Standardisée par le W3C en 1998, la technologie XML [BPSM+98] est aujourd'hui largement reconnue, acceptée et utilisée par de nombreuses entreprises comme format universel d'échange de données. Reposant sur un système de balises au sein d'un document, cette technologie peut être employée pour exprimer n'importe quel type d'information. XML a ainsi transformé l'univers d'Internet. On la retrouve par exemple en tant qu'élément de sauvegarde de documents ou de bases de données ou encore comme format d'échange de données [Dan03]. Son rôle est de représenter des données de manière structurée.

Le fait que XML soit simple, facilement lisible par l'homme, permettant de travailler sur des systèmes hétérogènes, le rend parfaitement adapté aux échanges entre plates-formes. C'est une spécification permettant de créer de nouveaux langages donc un métalangage. Ainsi, plusieurs langages dérivent du XML ou sont eux-mêmes décrits en XML. Parmi ceux-ci on peut citer :

- XLink [8] pour lier des documents.
- XPath [9] pour adresser des documents XML.
- WSDL, UDDI, SOAP pour créer et utiliser des Services Web.

WSDL (*Web Services Description Language*)

WSDL [CMRW07] permet de décrire le Service Web en précisant les méthodes disponibles, les formats des messages d'entrée et de sortie, le protocole de transport utilisé, et comment y accéder (la localisation du service). La version 2.0 de WSDL a été approuvée et standardisée en juin 2007 par le W3C.

WSDL est une interface qui cache le détail de l'implémentation du service, permettant une utilisation indépendante de la plate-forme et du langage utilisé. Il est modulaire car constitué de plusieurs parties permettant la plus grande abstraction possible dans la définition des services. La fragmentation des définitions permet ainsi une séparation des descriptions abstraites et concrètes donnant aussi la possibilité à une réutilisation de la partie abstraite (cf. Figure 2.2).

La structure d'un message WSDL est décrite dans la Figure 2.2.

« ***description*** ». Racine d'un document WSDL, cet élément est utilisé afin de déclarer les espaces de noms utilisés tout au long du document.

« ***types*** ». Cet élément décrit les types de messages que le service envoie et reçoit lors de l'appel d'une des méthodes. Les espaces de noms nécessaires à la définition de la structure de données sont inclus à ce niveau du document.

8. http ://www.w3.org/TR/xlink/
9. http ://xmlfr.org/w3c/TR/xpath/

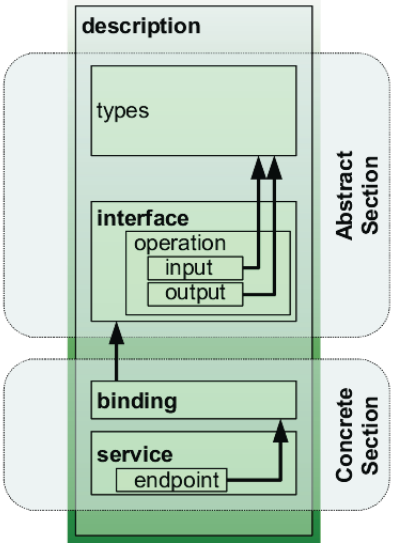


FIGURE 2.2 – Structure d'un document WSDL

« ***interface*** ». Cet élément décrit l'ensemble des fonctionnalités dites *operation* fournies par le Service Web. Un élément *operation* représente une interaction entre le client et le service. Les messages que le service reçoit (*input*) et ceux que le service envoie au client (*output*) sont définis au niveau de cet élément. De plus, l'élément *operation* possède un attribut *pattern* qui définit la séquence selon laquelle les messages sont transmis.

« ***binding*** ». Il décrit comment accéder au service. La liaison décrit la façon dont un ensemble d'opérations abstraites est mis en œuvre pour un protocole particulier (HTTP par exemple).

« ***service*** ». Cet élément définit la localisation du Service Web. Pour chaque interface décrite, un élément *service* lui est associé. L'élément nommé *endpoint* définit un port d'accès en référençant l'élément *binding* associé et en déclarant le lien localisant le service.

UDDI (*Universal Description, Discovery and Integration*)

UDDI [CHvR+04] est un annuaire de services particulièrement destiné aux services Web. Spécifié par le consortium OASIS (*Organization for the Advancement of Structured Information Standards*) [10], sa dernière version (UDDI 3.0.2) date de 2004.

La structure d'un annuaire UDDI est décrite dans la Figure 2.2.

« ***BusinessEntity*** ». Ce composant contient de l'information descriptive sur le fournisseur de services et sur les services proposés. La description des services contenue dans cette entité est de haut niveau donc aucune information technique n'est décrite ici.

10. https ://www.oasis-open.org/

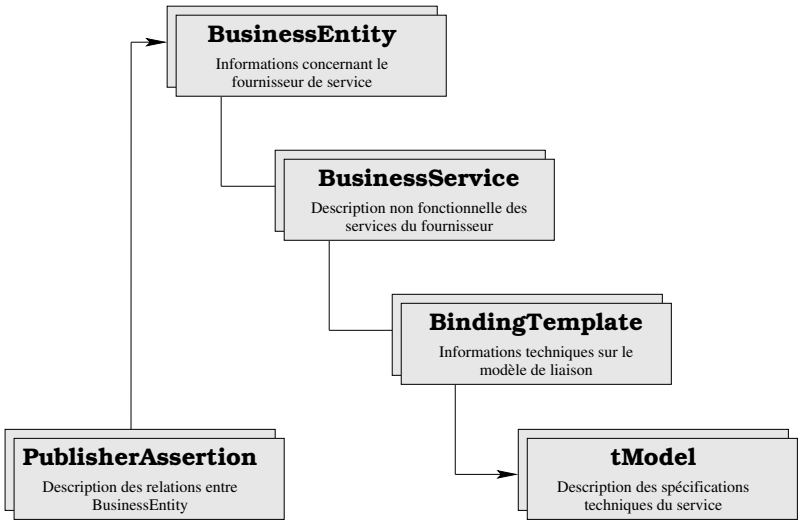


FIGURE 2.3 – Structure d'un annuaire UDDI

« ***BusinessService*** ». Ce composant représente les services proposés par un fournisseur. Ce dernier peut rassembler dans cette entité un ensemble de services répondant aux mêmes objectifs dans une même catégorie. Par exemple, une catégorie tourisme peut contenir un service météorologique, un service localisant les sites touristiques, etc.

« ***BindingTemplate*** ». Ce module décrit les points d'accès aux Services Web et le moyen d'y accéder (les différents protocoles à utiliser) afin d'invoquer les services.

« ***tModel*** ». Ce composant permet d'associer un service à sa description WSDL. Le client potentiel peut ainsi avoir connaissance des conventions d'utilisation du service.

« ***PublisherAssertion*** ». Ce composant représente un ensemble de règles contractuelles d'invocation de services sous forme de protocoles entre deux partenaires (fournisseur et client).

À partir des informations répertoriées dans un UDDI, un futur client peut connaître par l'intermédiaire de celui-ci les fournisseurs d'un service, les services proposés par un fournisseur donné, les moyens d'invoquer un service.

Remarque 2.1. *A l'origine, il existait des registres UDDI dits publics (tels que ceux de Microsoft ou IBM) pour lesquels n'importe quel utilisateur pouvait devenir, soit fournisseur, soit client de Services Web. L'universalité de ces registres devait amener UDDI à devenir le standard de publication des Services Web. Malgré celle-ci, UDDI n'a jamais atteint son but. Par conséquent, la maintenance des registres publics a été suspendue. Le réel succès de l'UDDI se situe ainsi au niveau des registres privés. En effet, de nombreuses organisations utilisent les spécifications de UDDI afin d'implémenter leur propre registre de Services Web.*

SOAP (*Simple Object Access Protocol*)

SOAP [GHM+03] est une initiative conjointe de Microsoft et IBM. La version 1.2 de SOAP est une recommandation du W3C depuis juin 2003.

SOAP définit un ensemble de règles pour structurer des messages dans le but d'exécuter des dialogues requête-réponse de type RPC (*Remote Procedure Call*). En fait, le RPC est un principe d'appel de procédure type client/serveur s'exécutant sur une machine distante dans un environnement d'applications distribuées. Il n'est pas lié à un système d'exploitation ni à un langage de programmation, donc, théoriquement, les clients et serveurs de ces dialogues peuvent tourner sur n'importe quelle plate-forme et être écrits dans n'importe quel langage du moment qu'ils puissent formuler et comprendre des messages SOAP.

La structure d'un message SOAP est décrite dans la Figure 2.4.

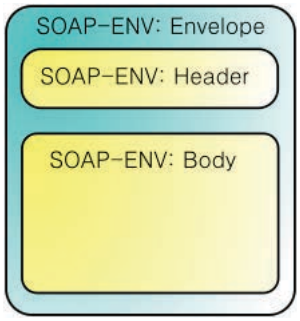


FIGURE 2.4 – Structure d'un message SOAP

« ***Envelope*** ». Cet élément est optionnel. Lorsqu'il est définit, il spécifie le style d'encodage et également les *namespaces* (espaces de noms).

« ***Header*** ». Les en-têtes SOAP sont optionnelles et sont typiquement utilisées pour transmettre des données d'authentification, de gestion de session, de transactions, de certificats de sécurité, etc.

« ***Body*** ». Cet élément encapsule la balise de méthode qui porte le nom de la méthode elle-même dans le cas d'un message de type requête et le nom suivi du mot « *Response* » dans le cas d'un message de réponse. La balise de la méthode reçoit typiquement le *namespace* correspondant au nom du service pour assurer l'unicité.

REST (Representational State Transfer)

Introduit en 2000 par Roy Fielding [Fie00], REST (*Representational State Transfert*) est un style architectural qui exploite essentiellement les technologies et les protocoles du Web.

Les Services Web qui suivent les principes de l'architecture REST sont souvent appelés *RESTful*. Ce style architectural estime en effet qu'il n'est, dans bien des cas, pas nécessaire de faire appel aux couches d'abstraction proposées par SOAP et XML-RPC, et que les méthodes de HTTP, combinées avec de bonnes URIs (*Universal Resource Identifier*), suffisent amplement dans la majorité des cas.

Les services *RESTful* ont permis d'apporter de réels gains en termes d'usage et de performance. En effet, l'absence de contrat WSDL spécifique pour chaque service a permis de réduire le couplage entre le client et le fournisseur de service et minimiser les différences d'interface et de sémantique au sein de ressources hétérogènes. En outre, contrairement à SOAP qui induit un typage fort au travers du WSDL, une ressource REST n'est pas typée, et peut être représentée via tout type de format d'échange de données. Face à ces avantages, plusieurs fournisseurs de services n'ont pas hésité à abandonner l'architecture SOAP au profit d'une architecture REST.

L'idée du *RESTful* est d'appliquer les principes de REST dans le développement de Services Web présentant les caractéristiques suivantes :

- *Resource-centric* : les entités conceptuelles et les fonctionnalités sont modélisées comme des ressources et identifiées par des URIs. Une URI qui est à la fois le nom et l'adresse d'une ressource. Donc pour qu'une ressource soit connue, il faut qu'elle soit accessible par son URI.
- L'interface uniforme : les ressources sont accessibles et manipulées via les opérations standardisées (GET, POST, PUT, DELETE, etc.) dans le protocole HTTP. GET est une opération pour prendre la représentation des ressources. POST, PUT et DELETE sont respectivement utilisées pour créer, mette à jour, et supprimer les ressources.
- Sans état : les composants du système communiquent via ces interfaces d'opération (précédemment décrites) et échangent les représentations de ces ressources. Dans un environnement de REST, chaque requête d'un client vers un serveur doit contenir toute l'information nécessaire pour permettre au serveur de comprendre la requête, sans avoir à dépendre d'un contexte conservé sur le serveur.

Vers une spécification JSON-WSP

Soulignons que nous assistons à l'emergence d'un nouveau protocole nommé JSON-WSP (*JavaScript Object Notation Web Service Protocole*) pour la mise en œuvre de Services Web. JSON-WSP [11] utilise JSON (*JavaScript Object Notation*), un format léger d'échange de données, pour la description et l'invocation de services.

11. https ://en.wikipedia.org/wiki/JSON-WSP

Tout comme le langage de description WSDL pour SOAP ou IDL pour CORBA, JSON-WSP propose un format pour décrire les types et les méthodes qui sont utilisés dans un service donné. Il décrit également les relations inter-types (c'est-à-dire types imbriqués) et définit quels types sont attendus comme arguments de la méthode et quels types l'utilisateur peut attendre à recevoir en tant que valeurs de retour des méthodes. Enfin la description donne la possibilité d'ajouter de la documentation sur les méthodes et paramètres de retour d'un service.

JSON-WSP utilise le protocole HTTP avec la méthode POST pour assurer les communications entre les clients et le serveur par échanges d'objets JSON.

2.3 Services de grilles informatiques

La grille informatique est un autre environnement qui propose une architecture orientée service. Dans cette section, nous présentons le concept de grille informatique. Par la suite, nous explorons les grilles de services.

2.3.1 Concept de grille informatique

L'évolution du Web a permis le développement de nombreuses ressources informationnelles, matérielles et logicielles distribuées, telles que les bases de données, les bases de connaissances, les espaces de stockage, les processeurs, et les divers types d'applications et d'utilitaires. Cette évolution a apporté une modification profonde dans la manière d'utiliser les ressources informatiques. En effet, face aux besoins croissants en termes de puissance de calcul et de capacité de stockage (par exemple, pour les applications d'études des changements climatiques [12], d'études de l'ADN [13], de sciences pour l'ingénierie de pointe [14], de traitement de données dans le domaine de la physique des particules [LSG+95], de surveillance et de modélisation de la pollution environnementale, d'études océanographiques [15], etc.), la communauté informatique s'est intéressée aux architectures distribuées à large échelle, afin d'offrir des solutions pour le stockage de données et le calcul réparti à un plus grand nombre d'applications et d'utilisateurs.

C'est dans ce contexte qu'est née la notion de *grille informatique*. Ce concept apparu à la fin des années 90 a été défini et formalisé par Ian Foster et Carl Kesselman dans [FK99]. Par la suite, une série de workshops et de forums dédiés aux rencontres de la communauté

12. http ://www.cosmologyathome.org
13. http ://csgrid.org/csg/dna/
14. www.lsst.org
15. https ://www.nodc.noaa.gov

scientifique internationale autour de cette technologie émergente était le point de départ de la révolution informatique du 21ème siècle avec un catalyseur important : *Internet* !

Très rapidement, dans le but d'offrir toujours plus de puissance de calcul, il est apparu qu'il était également possible d'utiliser conjointement plusieurs grilles pour résoudre un problème. Cette évolution technologique a permis de rendre opérationnelle les grilles informatiques telles que EGEE [BHK+06] en Europe ou encore TeraGrid [Cat02] aux États-Unis. La Figure 2.5 présente quelques grilles populaires à travers le monde.

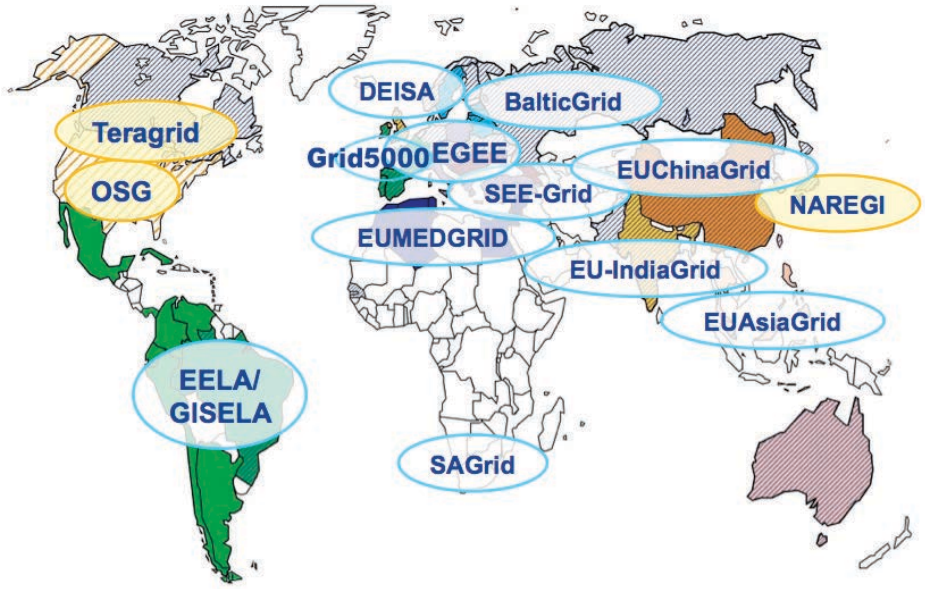


FIGURE 2.5 – Quelques grilles populaires à travers le monde

L'objectif d'une grille informatique est ainsi de concevoir une architecture informatique permettant de mettre à disposition des utilisateurs un ensemble hétérogène de ressources dont ils ont besoin au moyen d'une interface simplifiée. Toute ressource, qualifiée de service (service de stockage, service de calcul, logiciel de traitement d'une tâche spécifique, etc.), peut faire partie d'une grille à condition qu'elle soit mise à disposition via une interface d'accès standard. Cette grande flexibilité permet d'inclure au sein d'une même grille des machines avec des processeurs, des quantités de mémoire et des systèmes d'exploitation différents. Non seulement les machines peuvent être différentes mais également des réseaux très divers peuvent faire partie d'une grille comme Internet et des réseaux longue distance privés, mais aussi toutes sortes de réseaux locaux haute performance.

La complexité du réseau et des logiciels de gestion du système doit être invisible pour l'utilisateur. Ainsi, il doit accéder aux ressources de façon transparente. En effet, la vision originelle d'une grille informatique (*Grid computing*, traduit littéralement *grille de calcul*) est inspirée d'une analogie avec le réseau électrique (*Power grid*, terme lui même

inspiré des installations du réseau électrique américain). L'abonnement à un fournisseur d'électricité permet à tout consommateur d'accéder à l'électricité sans se soucier où et comment elle est produite (barrage, centrale nucléaire, éolienne, etc.).

Définitions et terminologie

Nous définissons dans cette section, la notion de grille informatique ainsi que les éléments qui la caractérisent.

Définition 2.4. ***Grille informatique***
Une grille informatique [FK99, FKT01] désigne une infrastructure constituée d'un ensemble de ressources informatiques (de stockage, de calcul, de réseau, etc.) appartenant à des entités administratives différentes (appelés ***sites****), établies dans le but de résoudre un même problème. Ces ressources, interconnectées par un réseau, sont caractérisées par leur hétérogénéité et leur distribution géographique.*

Définition 2.5. ***Site***
Un site est un ensemble de ressources informatiques localisées géographiquement dans une même organisation (campus universitaire, centre de calcul, entreprise, etc.) et qui forme un domaine d'administration autonome, uniforme et coordonné.

Remarque 2.2. *Tout au long de ce manuscrit, nous utilisons indifféremment les expressions de grille informatique, grille de calcul ou simplement grille.*

Le but des grilles informatiques est de favoriser la collaboration à très large échelle entre plusieurs partenaires, menant un projet commun et nécessitant des calculs intensifs et/ou sur des volumes de données importants. De telles collaborations sont supportées par le partage de ressources hétérogènes et dynamiques au sein d'organisations virtuelles regroupant ces différents partenaires.

Notion d'organisations virtuelles (VO)

Mettre en œuvre une grille de calcul, c'est vouloir partager des ressources. Mais tous les utilisateurs ou toutes les organisations n'ont pas les mêmes besoins, ni les mêmes préoccupations ; ils n'utiliseront donc pas nécessairement l'outil de la même façon.

Les VO regroupent ainsi les utilisateurs de grille en fonction de leurs objectifs, la grille servant de contexte d'échanges de service. En effet, l'usage des grilles était à ses débuts l'apanage du calcul intensif, mais son usage est désormais étendu à n'importe quel type de service. Les VO permettent ainsi à des groupes d'organismes et/ou individuels de partager des ressources de façon contrôlée, de sorte que tous les membres puissent collaborer pour réaliser un but commun.

Par exemple, le *DataGrid* est conçu pour trois communautés scientifiques : la Recherche sur les Particules, les Sciences de la Vie et l'Observation de la Terre. Les données des uns n'intéressent pas forcément les autres. Chaque domaine a ses propres contraintes de sécurité et fait appel à des ressources de différentes natures.

Ainsi, on définit des VO qui peuvent prendre la forme de fournisseurs d'applications, de fournisseurs de données stockées, mais également de consommateurs de ressources. La durée de vie des VO peut être variable, tout comme leur composition et les buts qu'elles poursuivent. Une organisation peut donc participer à une ou plusieurs VO en partageant une partie ou toutes ses ressources.

2.3.2 Architecture d'une grille

L'architecture d'une grille [FKT01], comme illustrée dans la Figure 2.6, est organisée en couches. Une couche est une abstraction représentant un ensemble de services.

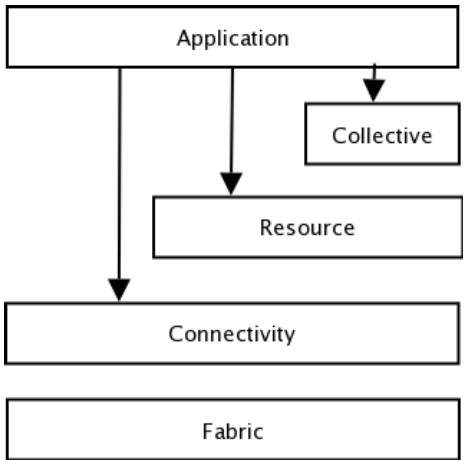


FIGURE 2.6 – Architecture d'une grille informatique

Fabric layer. C'est la couche de plus bas niveau, elle est en relation directe avec le matériel afin de mettre à disposition les ressources partagées. Les ressources fournies par cette couche sont d'un point de vue physique des ressources telles que des processeurs pour le calcul, des systèmes de stockage, des catalogues ou annuaires, des ressources réseau, etc. Une ressource peut-être également une entité logique comme un système de fichiers distribué, ou un serveur virtuel dans le cas d'un cluster d'ordinateurs. Lorsqu'une demande d'accès à une ressource est formulée, par le biais d'une opération de partage d'un niveau supérieur, un composant logiciel du niveau *Fabric* est invoqué. Le rôle des composants de la couche *Fabric* est donc d'offrir au niveau supérieur des fonctions permettant d'exploiter les ressources logiques et physiques de la grille.

Connectivity layer. Cette couche implémente les principaux protocoles de communication et d'authentification nécessaires pour des transactions s'effectuant dans le réseau de la grille.

Les protocoles de communication permettent l'échange des données entre les ressources de la couche *Fabric*.

Les protocoles d'authentification basés sur les services de communication fournissent des mécanismes sécurisés de cryptographie pour vérifier l'identité des utilisateurs et des ressources. En ce qui concerne les aspects de sécurité de cette couche, plusieurs des normes de sécurité développées dans le cadre de la suite de protocole d'Internet sont applicables dans ce système.

Resource layer. Cette couche utilise les services des couches connectivité et fabrique pour collecter des informations sur les caractéristiques des ressources, les surveiller et les contrôler. Soulignons que la couche ressource ne se préoccupe pas des ressources et de leurs interactions d'un point de vue global. Ceci incombe à la couche collective. Donc, elle ne s'intéresse qu'aux caractéristiques essentielles des ressources et à la façon dont elles se comportent. Au niveau de cette couche, deux types de protocoles sont à distinguer :

- Les protocoles d'information qui sont définis pour obtenir des informations sur la structure et l'état d'une ressource. Parmi ces informations on peut citer la configuration de la ressource, sa charge courante, la politique d'utilisation telle que le coût par exemple.
- Les protocoles de gestion qui sont définis pour négocier l'accès à une ressource partagée. On peut ainsi spécifier, par exemple, les ressources requises et les opérations à exécuter, comme la création de processus ou d'accès aux données. Les protocoles de gestion sont responsables de l'instanciation des rapports de partage. Ils servent de point d'application des politiques locales, s'assurant que les opérations demandées aux protocoles soient conformes à la politique sous laquelle la ressource doit être partagée.

Cette couche a en charge le monitoring (surveillance et remontée d'alarmes) des opérations. Elle contrôle l'exécution des traitements et signale les erreurs rencontrées aux couches de niveaux supérieurs qui souhaiteraient en être informées.

Collective layer. Cette couche se charge des interactions entre les ressources. Elle gère l'ordonnancement et la co-allocation des ressources en cas de demande des utilisateurs, faisant appel à plusieurs ressources simultanément. C'est elle qui choisit sur quelle ressource de calcul faire exécuter un traitement en fonction des coûts estimés. Elle a en charge la surveillance des services et elle doit assumer la détection des pannes. En outre, elle joue le

rôle d'annuaire qui est une base de données répertoriant les ressources et leurs différentes caractéristiques. Quand une application va devoir utiliser une ressource, un courtier de ressource (*Resource Broker*) va chercher dans l'annuaire celle correspondant le mieux aux exigences requises par la tâche. Une fois que l'annuaire aura transmis la localisation de la ressource, la tâche pourra être exécutée.

Application layer. C'est la couche la plus haute du modèle, elle correspond aux logiciels qui utilisent la grille pour fournir aux utilisateurs ce dont ils ont besoin, qu'il s'agisse de calcul, ou de données. Les applications déployées utilisent des services de chacune des couches de l'architecture. Les couches *Collective* et *Ressource* sont par exemple sollicitées pour la recherche des ressources. Une fois la ressource identifiée, et après s'être authentifié au travers de la couche *Connectivity*, les applications utilisent les services du niveau *Fabric* pour y accéder.

2.3.3 Topologies de grilles

Dans cette section, nous présentons les différentes topologies de grilles.

Du point de vue topologique, les auteurs de [FBA+03] définissent trois types de grilles informatiques : les *intragrilles*, les *extragrilles* et les *intergrilles*.

L'intragrille. Elle est constituée d'un ensemble de services et de ressources relativement homogènes, qui appartiennent au même organisme. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion performant et haut-débit et d'un domaine de sécurité unique et contrôlé par les administrateurs de l'organisme.

L'extragrille. Elle est constituée d'un modèle en agrégeant plusieurs *intragrilles*. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion relativement hétérogène, de plusieurs domaines de sécurité distincts, et d'un ensemble relativement dynamique de ressources. Les échanges B2B (*Business-to-Business*) entre entreprises ou universités partenaires sont un exemple d'utilisation.

L'intergrille. Elle consiste à agréger les grilles de multiples organisations, en une seule grille. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion très hétérogène, de plusieurs domaines de sécurité distincts et ayant parfois des politiques de sécurité différentes voire même contradictoires, et d'un ensemble très dynamique de ressources. Les *intergrilles* sont souvent mises en œuvre lors de grands projets industriels (conception d'un avion par un consortium aéronautique par exemple), scientifiques (modélisation de protéines par exemple) où plusieurs organisations seront amenées à participer.

2.3.4 Les services de grille

Comme nous l'avons vu dans la section précédente, une grille de calcul se caractérise par des VO et des ressources (calcul, stockage, logicielles, etc.) dispersées et hétérogènes. L'état des ressources matérielles et logicielles est de nature variable. De même, les VO et la connectivité réseau sont également variables au cours du temps. Il apparaît clairement dans un tel contexte que l'utilisation de standards est un point particulièrement critique pour la réussite de la mise en place d'une infrastructure de grille, notamment pour faire face aux problèmes d'hétérogénéité du matériel et des logiciels, et aussi favoriser la réutilisation.

La spécification de services de grille (*Grid Service Specification*) [TCF+02] a pour objectif de normaliser les services composant les grilles, afin de garantir l'interopérabilité de systèmes hétérogènes pour le partage et l'accès à des ressources de calcul et de stockage distribuées. La spécification définit qu'un *Service Grid* est un Service Web respectant un ensemble de conventions (interfaces et comportement) adaptées aux contraintes de l'environnement de grille. Cette spécification est le résultat de recherches établies par le GGF (*Global Grid Forum* devenu depuis Septembre 2006 l'*Open Grid Forum*[16]) et ayant abouti aux standardisations de l'OGSA (*Open Grid Service Architecture*) [FKS+06] et de l'OGSI (*Open Grid Service Infrastructure*) [TCF+03].

Soulignons que ce même groupe de travail est à l'origine de la spécification Grid-RPC [SNM+02], qui est un modèle de programmation par appel de procédures à distance dans le contexte des grilles de calcul. Comparé au modèle RPC classique, le modèle Grid-RPC inclut également la possibilité d'effectuer des tâches parallèles à gros grain de manière asynchrone. L'API Grid-RPC fournit des mécanismes standardisés et portables ainsi qu'une programmation simplifiée pour implémenter le RPC dans les grilles. Différentes implémentations de cette spécification ont été proposées dans les projets DIET [CD06], NetSolve [SYAD05] ou encore Ninf-G [TNS+03].

L'évolution des normes OGSA et OGSI a eu comme résultat significatif, la proposition de la norme WSRF (*Web Services Resource Framework*) [CFF+04] qui permet d'uniformiser les mécanismes pour accéder à des ressources à état avec des Services Web/Grid.

Dans les sections qui suivent, nous présentons ces différents standards.

La norme OGSA (*Open Grid Services Architecture*)

La norme OGSA (*Open Grid Service Architecture*) [FKS+06], est une série de spécifications techniques par lesquelles on définit une infrastructure pour intégrer et gérer les services à l'intérieur d'une organisation dynamique, virtuelle et distribuée.

16. https ://www.ogf.org

Elle a été adoptée en 2002, lors de la conférence à Toronto, par le GGF (actuellement l'OGF pour *Open Grid Forum*) sur la base d'une extension des normes utilisées pour les Services Web (SOAP et WSDL). En effet, Ian Foster et al. [FKNT02b], sont partis de l'idée selon laquelle, une fois les ressources informatiques virtualisées, il est envisageable de mutualiser tout ou une partie de ces ressources pour générer des services.

La norme OGSA vise ainsi à définir des mécanismes pour virtualiser les ressources et les restituer sous forme de services afin de pouvoir les assembler et les désassembler en fonction des besoins des utilisateurs.

OGSA décrit un grille de services comme étant un Service Web classique se conformant à un certain nombre de spécifications dont principalement :

Le nommage. Une instance de service a un nom unique, utilisé pour l'identification.

Les données de services. Elles sont associées à chaque instance de grille de service. Des opérations pour contrôler et modifier les valeurs des données sont disponibles.

La durée de vie. Gestion de la durée de vie des services.

Les notifications. Interfaces pour enregistrer des inscriptions et délivrer des notifications si certains événements surviennent (en cas de modification des données d'un service par exemple).

Deux caractéristiques essentielles fondent la singularité d'OGSA. D'une part, OGSA est un modèle d'architecture orientée services par opposition à l'architecture orientée système, tel que le modèle client/serveur. D'autre part, OGSA prévoit la gestion de l'état dans les services, ce qui n'est pas prévu à l'origine pour les Services Web.

Définition 2.6. ***Service à état***
Un service dit à état, est un élément qui modélise un état physique (base de données, fichier, serveur, etc.) ou logique (contrat, accord, etc.) qui est persistant et qui est modifié par des interactions. Il est caractérisé par sa capacité à garder en mémoire les résultats d'actions antérieures comme par exemple, un indicateur de présence lorsque des personnes se connectent sur un service de messagerie instantanée [CTT05].

La norme OGSI (*Open Grid Services Infrastructure*)

La norme OGSI (*Open Grid Service Infrastructure*) [TCF+03], représente la convergence des Services Web avec la technologie des grilles informatiques. Elle définit le mécanisme pour gérer les instances des services de grille.

Normalisée en juin 2003, OGSI constitue l'infrastructure de base pour la définition des services de grille composant l'architecture OGSA [FKS+06].

L'objectif de la norme OGSI est de faciliter le développement de Services Web adaptés aux contraintes spécifiques à un environnement de grille. Ainsi, les services de grille sont spécifiés afin de garantir l'interopérabilité de systèmes hétérogènes, pour le partage et l'accès à des ressources de calcul et de stockage distribués. Un certain nombre d'opérations standard sont implémentés, dont certaines sont obligatoires (gestion du cycle de vie par exemple), et d'autres facultatives (la notification par exemple).

Cependant, au cours de son évolution, le *GGF* annonça en 2004 le WSRF (*Web service Resource framework*) [CFF+04] comme successeur de l'OGSI. Ainsi, les concepts OGSI ont été repris avec une nouvelle terminologie issue du groupe de normalisation de WSRF.

La norme WSRF définit une infrastructure générique de modélisation des ressources dynamiques et d'accès à ces dernières à l'aide des Services Web, afin de faciliter la définition et l'implémentation d'un service, ainsi que l'intégration et la gestion de plusieurs services. Cette norme définit la convergence entre les technologies de grille et celle de Services Web. Elle permet ainsi d'uniformiser les mécanismes d'accès à des ressources à état avec des Services Web/Grid. La Figure 2.7 décrit ce processus de convergence.

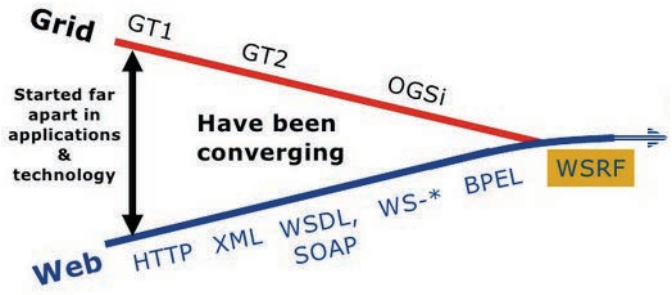


FIGURE 2.7 – Convergence Grille et Service Web

Le WSRF permet de définir, d'inspecter et de contrôler l'état du service et contribue ainsi à l'utilisation de la grille à très grande échelle.

2.4 Gestion de services dans un environnement de grille informatique à large échelle

Dans cette section, nous étudions les mécanismes de gestions de services dans un environnement grille informatique à large échelle.

2.4.1 Objectifs et motivations

La gestion de ressources réparties et en particulier, la découverte appropriée de ressources constitue un des défis essentiels dans un environnement de grille [HCE10, MSC+12, NRNH14, TTP+07]. En effet, la grille étant une agrégation de ressources dispersées et disponibles dans plusieurs organisations, rechercher et localiser un service résolvant un besoin spécifique devient un enjeu majeur. Ainsi, plusieurs mécanismes de découverte de services dans un environnement de grille ont été proposés dans la littérature.

Les auteurs de [TTP+07] définissent la découverte de ressources comme « le service permettant la localisation de ressources à travers plusieurs domaines administratifs suivant un ensemble d'attributs ».

On peut donc dire que la découverte de services est un processus qui consiste à rechercher dans un système un ou des services correspondants à une description et à retourner un ensemble d'adresses de services qui correspond à cette description. La description est souvent constituée d'un ensemble de méta-données (mots-clefs).

Pour répondre à ces exigences, les grilles n'ont pas cessé d'évoluer en termes d'architectures qui guident le développement de tous les composants d'une application. En effet, les systèmes de grilles traditionnelles présentent des architectures qualifiées de centralisées ou hiérarchiques. Par la suite, Foster et Iamnitchi suggèrent que les grilles peuvent fortement tirer profit des technologies pair-à-pair [IFN02]. En effet, celles-ci offrent de nombreux avantages grâce à leurs propriétés fondamentales et inhérentes telles que l'auto-organisation, la tolérance aux pannes, le passage à l'échelle, le changement dynamique de topologie, etc. Notons toutefois que les modèles de grilles centralisés comme hiérarchiques continuent à être utilisés [BWH06, KS07, KKRG10, NRNH14].

Nous proposons ainsi de classifier les solutions de découverte de services en fonction du degré de centralisation de l'architecture de grille. En fait, le degré de centralisation d'une architecture impacte grandement sur la tolérance aux pannes, ainsi que de la possibilité pour l'application de passer à l'échelle.

On peut ainsi distinguer d'une part, les approches basées sur le modèle traditionnel client/serveur. Les solutions basées sur ce modèle sont soit centralisées [BWH06, FFK+97, KS07, KKRG10] ou hiérarchiques [CD06, CFFK01, Fos05, LMH+09].

On distingue d'autre part, les approches basées sur le modèle pair-à-pair. Les solutions basées sur ce modèle sont soit purement décentralisées ou hybrides [AA+12, BJ15, BG03, MTV05]. De même, les solutions décentralisées peuvent être non-structurées [BMH10, IF01, IF04, RJTB08] ou structurées [JA14, RFH+01, SMK+01, TTZ07].

La Figure 2.8 illustre ces différentes mécanismes de découverte de services.

Dans ce qui suit, nous nous proposons d'étudier les approches proposées dans la littérature pour la découverte de services dans un environnement de grille. A l'issue de

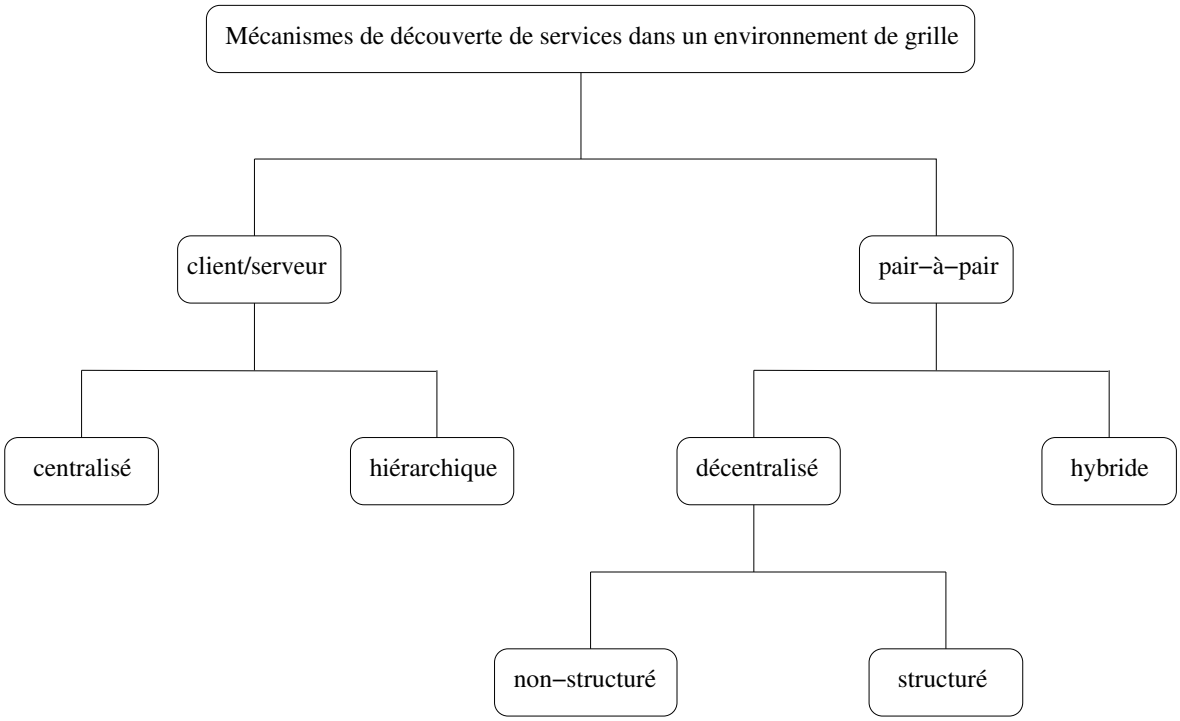


FIGURE 2.8 – Mécanismes de découverte de services dans un environnement de grille

cette étude, nous faisons la synthèse des travaux des approches proposées afin de dégager nos objectifs de recherche.

2.4.2 Mécanismes de découverte de services dans les grilles

Cette section est divisée en deux parties. Dans un premier temps, nous verrons les approches de découverte de services basées sur le modèle client/serveur. Dans un second temps, nous verrons celles basées sur le modèle pair-à-pair.

Approches basées sur le modèle client/serveur

Les systèmes de grilles traditionnelles de première génération sont basés sur ce modèle d'architecture. En effet, la topologie de l'architecture réseau des grilles de cette génération fut inspirée directement de celle des clusters qui est centralisée. Ainsi, les mécanismes de la découverte de services suivent une approche orientée client/serveur.

a) Approches centralisées

Dans les mécanismes centralisés, un "*serveur de coordination*" assure les fonctions de découverte de services. C'est aussi lui qui est en charge de l'ordonnancement et de la

soumission des tâches à traiter.

Lorsqu'une entité demande un service, elle formule sa requête et l'envoie au serveur de coordination. Cette dernière recherche les ressources appropriées afin de les allouer à l'entité ayant effectuée la demande.

Plusieurs mécanismes de la découverte de services basés sur ce modèle d'architecture ont été proposés. Parmi ceux-ci, on peut citer [BWH06, FFK+97, KS07, KKRG10].

Le défaut majeur de cette approche est le goulot d'étranglement et point unique de défaillance que représente le serveur de coordination, qui risque la saturation et compromet l'ensemble du système en cas de panne.

b) Approches hiérarchiques

Dans les mécanismes hiérarchiques, des serveurs sont interconnectés suivant une hiérarchie prédéfinie par les administrateurs de la grille. Chaque serveur déclare ses ressources (services) auprès d'un agent, souvent appelé *Registry* ou RMS (*Resource Management System*) ou encore courtier (*broker*). Le client s'adresse alors à l'agent pour connaître les serveurs pouvant satisfaire ses besoins. Il peut ensuite directement contacter un serveur pertinent afin d'exécuter sa tâche. Les implémentations les plus complètes de ce modèle d'architecture sont Globus [Fos05] et DIET [CD06].

Globus [Fos05] est considéré comme l'outil le plus répandu pour le développement des applications tournant sur un système de grille. Il fournit les fonctionnalités et les services de base nécessaires à la construction de grille de calcul tels que la sécurité, la gestion des ressources ou la communication. Par exemple, le composant GRAM (*Grid Resource Allocation Manager*) [CFK+98] est proposé pour la soumission des tâches et leur exécution ainsi que la surveillance des exécutions.

Soulignons que le projet Globus, comme tout projet actif *open source*, a subit plusieurs stades d'évolution. En effet, sa version GT3 (*Globus Toolkit* 3) est conforme à la norme OGSA qui vise à combiner la technologie des Services Web à celle du *Grid Computing*.

Plusieurs solutions fondées sur Globus et héritant de son modèle d'architecture ont été proposées dans la littérature. Parmi celles-ci, on peut citer [CFFK01, FTL+02, HML09, KS07, RDM06].

Comme nous l'avons précisé plus haut, DIET (*Distributed Interactive Engineering Toolbox*) [CD06] repose sur la spécification Grid-RPC [SNM+02].

L'architecture de DIET est décrit dans la Figure 2.9. Dans DIET, la requête de recherche d'un client est reçue par un MA (*Master Agent*) qui la transmet à toute son arborescence d'agents locaux (*Local Agents*) dans le but de trouver le meilleur serveur de calcul appelé SeD (*Server Daemon*). Dans le cas où la requête ne peut pas être résolue dans son arborescence, le *master agent* transmet celle-ci vers d'autre MAs.

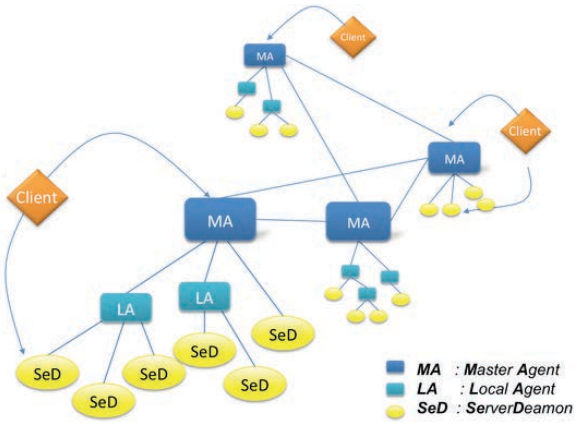


FIGURE 2.9 – Architecture de DIET

Les approches hiérarchiques permettent de mieux partager la charge du travail ce qui permet un meilleur passage à l'échelle en comparaison avec les approches centralisées. Toutefois, un serveur peut devenir un goulet d'étranglement si un grand nombre de requêtes lui est destiné. En outre, la panne d'un serveur peut rendre inaccessible les services d'une partie du système ; et c'est encore pire lorsqu'il s'agit d'un serveur racine.

Notons qu'il existe des travaux qui tentent de minimiser voire masquer les pannes de nœuds sensibles au niveau des structures hiérarchiques. C'est le cas notamment des travaux décrits dans [FCT15] qui visent à ajouter des mécanismes d'auto-adaptation à l'intergiciel DIET [CD06]. En effet, ces auteurs spécifient que lorsqu'un *Local Agent* (voir Figure 2.9) détecte qu'il n'a pas de père (de *Master Agent*), et qu'il a l'information qu'il est l'unique agent (*Local Agent*) dans le déploiement, alors il crée un *Master Agent* comme père. De cette manière, en présence de perturbations, le système converge toujours après un temps fini, dans un état stable.

Approches basées sur le modèle pair-à-pair

Les grilles et les systèmes pair-à-pair partagent plusieurs caractéristiques et peuvent ainsi être utilement intégrés. En effet, Foster et al. [FI03] affirment 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. Ainsi, le succès des systèmes P2P dû à leurs propriétés fondamentales, fait qu'ils soient largement utilisés dans un environnement de grille [HM15, KNS14, KL12, Tor12, ZHS14].

Dans ce qui suit, nous présentons d'abord les mécanismes de découverte basés sur les systèmes P2P décentralisés non-structurés. Ensuite, nous explorons les mécanismes de découverte basés sur les systèmes P2P décentralisés structurés. Enfin, nous passons en revue les mécanismes de découverte basés sur les systèmes P2P hybrides.

a) Approches décentralisées et non-structurées

Dans les mécanismes de découverte basés sur les systèmes P2P décentralisés non-structurés, les nœuds de la grille sont organisés de manière non-déterministe et ont tous les mêmes responsabilités. Ainsi, il n'y a pas de contrôle précis sur la topologie du réseau de la grille. Chaque nœud a son propre catalogue local. Il n'existe donc pas la notion d'index centralisé. De plus, il n'y a aucune restriction sur la manière de décrire un service désiré. L'approche de recherche par mots-clefs reste la plus utilisée.

Les systèmes P2P décentralisés non-structurés traditionnels (Gnutella 0.4 [Sol01] par exemple) utilisent la technique de découverte basée sur l'inondation. La recherche par inondation propose de retransmettre récursivement la requête de recherche à tous les voisins d'un nœud (sauf celui dont il a reçu la requête), jusqu'à la localisation du service ou l'expiration TTL qui traduit le nombre de retransmissions possibles d'une requête. Le TTL est utilisé pour éviter les recherches en boucle et que les messages circulent indéfiniment dans le réseau. Dans [IF01] par exemple, Iamnitchi et Ian Foster proposent une approche de localisation de ressources totalement décentralisée, et afin de limiter le nombre de messages propagés lors de la découverte de services, un TTL est ainsi utilisé. Ceci ne garantit toutefois pas des recherches fiables.

Les auteurs de [BMH10] proposent d'améliorer la technique d'inondation avec une diffusion selective basée sur des informations de cache des nœuds et limitée par un TTL.

D'autres solutions d'améliorations ont été proposées dont certaines basées sur les marches aléatoires [JM08, IF04], d'autres basées sur les marches aléatoires biaisés [RJTB08, Tor12] ou encore basées sur les indices de routage [MMO07].

La recherche par marche aléatoire consiste à retransmettre récursivement la requête de recherche à un unique voisin choisi aléatoirement. Le TTL est également utilisé pour limiter le nombre de sauts. La marche aléatoire est biaisée lorsque la requête de recherche est retransmise en choisissant le voisin destinataire de manière déterministe. C'est le cas par exemple des travaux présentés dans [JM08] où les auteurs proposent un mécanisme qui améliore la découverte classique basée sur des caches (proposée dans [IF04]) en permettant la coopération entre les caches. Toutefois, cette méthode nécessite une politique de mise-à-jour des caches sinon les métadonnées risquent d'être obsolètes.

Dans tous les cas, le choix du TTL n'est pas facile à déterminer. Les performances de la recherche dans un tel environnement sont donc tributaires du TTL qui possède une valeur bornée. En effet, si le TTL est grand, cela peut surcharger le réseau ; et s'il est petit, la recherche d'un service peut échouer même si la ressource demandée se trouve dans le système.

En résumé, les mécanismes non-structurés permettent de partager la charge du travail entre tous les noeuds du système vu qu'il n'y a aucun contrôle centralisé dans le

système. Ils sont ainsi bien adaptés face à la dynamicité du système vu que tous les nœuds possèdent les mêmes responsabilités. De plus, l'absence de contraintes liées à une structure quelconque, permet de relâcher certaines tâches complexes, comme le maintien d'index ou de structures de données réparties.

Cependant, les mécanismes de recherche, tel que l'inondation, sont très coûteux en termes de trafic et ainsi de charge qu'ils font subir au réseau. Ainsi, le passage à l'échelle n'est pas garanti dans un tel système. D'autre part, l'incomplétude des résultats de recherche peut être élevée, du fait que certains nœuds hébergeant des ressources significatives peuvent ne pas être atteints parce qu'ils sont tout simplement éloignés (en termes de nombre de sauts) de l'origine de la requête. Donc une recherche peut ne pas être fructueuse même si la ressource demandée se trouve dans le système.

b) Approches décentralisées et structurées

Dans les mécanismes de découverte basés sur les systèmes P2P décentralisés structurés, les nœuds de la grille maintiennent une structure virtuelle rigide (issue de la théorie de graphes) et basée sur les DHTs [MSC+12, NRNH14, Tor12, TTP+07].

Les protocoles les plus utilisés basés sur les systèmes P2P décentralisés structurés sont Chord [SMK+01], Pastry [RD01], CAN [RFH+01] et Kademlia [MM02].

Étant donné que l'expression de recherches sur des mots-clefs n'est pas possible avec les DHTs, les solutions étendent le protocole P2P sous-jacent pour satisfaire de tels besoins.

Par exemple, dans [CMG05], les auteurs se basent sur le protocole Pastry [RD01], pour proposer une méthode de découverte de services de calcul (par exemple, vitesse ou charge CPU, taille RAM, etc.), capable de traiter des requêtes à intervalle et à multi-attributs.

Les auteurs de [TTZ07] proposent une solution basée sur le protocole Chord [SMK+01] pour le traitement des requêtes à intervalle et à multi-attributs. Les identificateurs d'une DHT sont générés en concaténant le type de ressource avec l'attribut de la ressource. Ces auteurs ont choisi d'insérer seulement des attributs statiques (par exemple, type de système d'exploitation, vitesse CPU) dans la DHT. En effet, ils soutiennent que l'insertion d'attributs dynamiques (par exemple RAM libre) provoquerait des problèmes de performances surtout en présence des mises à jour fréquentes des ressources dynamiques.

Dans [JA14], les auteurs utilisent le protocole Chord [SMK+01] pour proposer une solution de découverte de service. Ces auteurs proposent de modifier la DHT de Chord en ajoutant de nouvelles entrées correspondant aux informations sur la localisation des nœuds récemment visités. Et lors de la réception d'une requête de recherche de service, un nœud recevant la requête vérifie d'abord dans son index RVN-id (*Recent Visited Node*) ; et c'est lorsque la clé recherchée n'est pas trouvée dans cet index que l'algorithme de recherche de Chord est invoqué.

Dans [CDT06], Caron et al. proposent une structure d'indexation pair-à-pair nommée DLPT (Distributed Lexical Placement Table) et basé sur un arbre de plus long préfixes pour la découverte de services. Comme une DHT, la DLPT stocke des références d'objets sous la forme de couples (clef, valeur) déclarés par les serveurs. Afin d'offrir des mécanismes de recherche multi-critères, ces auteurs proposent une structure multi-DLPT en construisant une DLPT par type d'information. De plus, des mécanismes de cache ont été proposés afin d'éviter les goulots d'étranglement sur les nœuds logiques stockant des clefs populaires. D'autres travaux ont été menés dans le but d'améliorer les performances de la DLPT. En effet, dans [CDT08], les mêmes auteurs ont proposé des mécanismes pour la réduction du coût de maintenance de la structure et aussi l'équilibrage de charge des pairs hébergeant cette DLPT. Caron et al. proposent ainsi un protocole auto-stabilisant à passage de messages afin de rendre cette structure tolérante aux pannes. Soulignons que l'intergiciel SBAM (Spades BAsed Middleware), développé dans le cadre de la mise en œuvre de la plateforme SPADES (Servicing Petascale Architecture and DistributEd System) [THCC11], constitue une implémentation de la DLPT.

En résumé, les mécanismes de découverte basés sur les systèmes pair-à-pair structurés sont plus rapides et plus efficaces que ceux basés sur les systèmes non-structurés. En effet, la recherche de service s'opère en temps logarithmique pour la plupart des systèmes pair-à-pair structurés [HCE10, LCP+05]. En outre, la nature distribuée de la table de hachage entre les différents pairs confère à ce type de système une certaine robustesse face aux pannes. Ainsi, le passage à l'échelle peut être garanti dans un tel système.

Cependant, les DHT de base des protocoles P2P supportent seulement une recherche exacte. En effet, la recherche dans une DHT s'effectue simplement par *matching* (correspondance) entre la clé de la requête et l'index dans la table. Par conséquent, pour offrir des mécanismes d'expressions de recherche sur des mots-clés, une implémentation de telles fonctionnalités est nécessaire.

c) Approches hybrides

Dans les mécanismes de découverte basés sur les systèmes P2P hybrides, les nœuds de la grilles n'ont pas les mêmes responsabilités. On distingue en effet, des nœuds appelés *super-nœuds* qui vont jouer le rôle d'annuaire en indexant les méta-données des nœuds rattachés à eux et appelés *nœuds ordinaires* ou *nœuds simples*.

Ce modèle présente l'avantage majeur de s'adapter plus facilement dans un environnement de grille. En effet, une grille étant constituée d'un ensemble de VOs, chaque *super-nœud* sera responsable d'une VO [AFdSY+04, DMMRS15, MTV05, PMB+05, TLL12]. De ce fait, la maintenance des métadonnées de services d'une VO est gérée par le *super-nœud* responsable de celle-ci.

Dans [MTV05], les *super-nœuds* sont organisés dans une couche pair-à-pair non-structurée et communiquent entre eux selon une approche empirique. Une requête de découverte est d'abord soumise localement dans une VO. Si le service recherché n'est pas localisé, alors la requête de découverte est propagée vers un ensemble de VOs, choisit suivant l'approche empirique.

En résumé, les mécanismes de découverte basés sur ces systèmes présentent comme avantage majeur, la réduction du trafic des requêtes et du temps de recherche. En effet, vu que les *nœuds ordinaires* ne sont pas concernés lors des processus de recherche, cette architecture offre des performances meilleures que celles de l'architecture pair-à-pair non-structurées. En outre, ces systèmes sont relativement tolérants aux pannes car l'indisponibilité d'un *super-nœud* n'affecte que son groupe et pas tout le système.

Toutefois, ce type de système est plus complexe à mettre en œuvre. De plus, le choix optimal des *super-nœuds* n'est pas trivial, plusieurs critères sont à définir selon les besoins de l'application.

Synthèse

Le Tableau 2.1 ci-dessous fait une synthèse des différentes approches de découverte de services en fonction du degré de centralisation de l'architecture de grille.

Architecture	Mécanisme	Avantages	Limites
Client-Serveur	Centralisé	+ Résultats de recherche fiables	– Goulot d'étranglement – Ne passe à l'échelle – Pas tolérant aux pannes
	Hiérarchique	+ Equilibrage de charge	– Passage à l'échelle difficile – Pas tolérant aux pannes
P2P	Non-structuré	+ Autonomie des nœuds + Tolérant à la dynamique du système + Flexibilité dans la recherche	– Inondation très couteux – Passage à l'échelle limité – Résultats de recherche non fiables
	Structuré	+ Temps de recherche logarithmique + Tolérant à la dynamique du système + Passage à l'échelle garanti	– Rigidité des requêtes avec les DHT – Coût de maintenance élevé – Caractéristiques des nœuds ignorées
	Super-pair	+ Equilibrage de charge + Tolérant à la dynamique des nœuds ordinaires	– Pérennité des super-pairs – Passage à l'échelle limité – Complexe à mettre en œuvre

TABLE 2.1 – Synthèse des différentes approches de découverte de services dans un environnement de grille

D'une manière générale, on peut dire que les mécanismes de la découverte de services basés sur le modèle client/serveur sont sensibles à la dynamique du système et généralement inaptes à passer à l'échelle. Les mécanismes basés sur le modèle pair-à-pair offrent de meilleures performances.

Néanmoins, nous remarquons qu'aucune des approches P2P (non-structuré, structuré ou hybride) ne présente une solution pleinement satisfaisante. En effet, un seul type de système pair-à-pair ne peut pas accomplir tous les besoins et exigences en termes d'hétérogénéité, de dynamicité, etc. d'un système de grilles. Nous pensons donc que différents systèmes P2P seront mieux adaptés que d'autres selon l'application. En d'autres termes, en fonction de l'environnement de grille (peu ou très dynamique, large échelle ou échelle moyenne, etc.), certaines architectures sont plus adaptées que d'autres.

Par exemple, une architecture P2P hybride peut s'adapter facilement à un environnement de grille organisé en VO puisque que ces deux modèles présentent une topologie semblable. Toutefois, pour assurer la communication entre *super-nœuds*, ces solutions se basent généralement sur les approches non-structurées ou structurées qui présentent des avantages, mais aussi des limites, telles que celles que nous venons d'élucider. De plus, le choix optimal des *super-nœuds* n'est pas trivial, plusieurs critères sont à définir selon les besoins de l'application.

Dans ce contexte, nous nous proposons dans le cadre de nos travaux de recherche d'élaborer une solution efficace de gestion dynamique de services, dans un environnement de grille pair-à-pair dynamique et à large échelle.

Afin d'apporter des solutions aux limites précédemment citées, nous faisons dans le cadre de la modélisation le choix de ne pas nous aligner sur une architecture typique d'exécution. Nous proposons ainsi un modèle générique ; c'est-à-dire applicable sur toute architecture pair-à-pair.

2.5 Conclusion

Dans ce chapitre nous avons étudié les services ainsi que leurs environnements d'exécution. Pour ce faire, nous avons tout d'abord présenté les généralités sur les services. Après avoir défini la notion de service, nous avons décrit l'architecture orienté services. Ensuite, nous avons présenté les Services Web puis les Services de grilles. Enfin, nous avons étudié les solutions de gestion de services dans un environnement de grille. Ce qui nous a permit de dégager notre problématique de recherche.

Nous présentons dans le prochain chapitre notre modèle pour de gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle.

Chapitre 3

Spécifications de services dans un environnement de grille P2P à large échelle

Résumé. *Dans ce chapitre, nous présentons notre solution, nommée P2P4GS (Peer-To-Peer For Grid Services), pour la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle.*

Nous commençons d'abord par exposer nos motivations ainsi que nos objectifs. Ensuite, nous présentons les spécifications de P2P4GS. Pour ce faire, nous définissons les principaux concepts de bases sur lesquels s'appuient la spécification. Par la suite, nous décrivons notre modèle d'architecture. Enfin, nous présentons le principe d'exécution de P2P4GS ainsi que son comportement face à la dynamicité du système.

Les travaux que nous présentons dans ce chapitre sont publiés dans des revues et conférences avec actes et comités de sélection : [GFN12a, GFN12b, GFRN14a, GFNR14, GFRN14b, GFRN15, GFRN16b, GFRN16a].

Sommaire

3.1 Motivations et objectifs 64
3.2 Spécifications de P2P4GS 65
3.2.1 Concepts de bases 65
3.2.2 Modèle d'architecture 68
3.3 Approche de structuration du système en communautés . . . 72
3.3.1 Présentation de la solution de structuration 72
3.3.2 Algorithme de structuration 75
3.3.3 Mécanismes d'adaptation à la dynamique du système 78
3.4 Synthèse 80

3.1 Motivations et objectifs

La gestion de ressources réparties dans plusieurs organisations (centres de calcul, entreprises, etc.) est une tâche complexe. De plus, les ressources d'une grille sont de nature très hétérogènes et dynamiques. D'autre part, la découverte de ressources constitue un des défis essentiels dans un tel environnement. En effet, la grille étant constituée d'une agrégation de plusieurs ressources dispersées géographiquement, rechercher et localiser un service résolvant un besoin spécifique devient un enjeu majeur.

Pour répondre à ces exigences, les grilles n'ont pas cessé d'évoluer en termes d'architectures qui guident le développement de tous les composants d'une application.

Nous avons présenté dans les chapitres précédents les raisons qui ont motivées la communauté scientifique à coupler les systèmes de grilles avec la technologie pair-à-pair afin d'assurer une meilleure évolutivité.

Nous avons vu que les systèmes pair-à-pair proposent plusieurs modèles d'architectures et dont les plus utilisés dans le cadre des grilles sont le modèle d'architecture décentralisée non-structuré, le modèle d'architecture décentralisée structuré et le modèle d'architecture hybride. Chacun de ces modèles bénéficie de plusieurs des avantages qu'offrent les systèmes pair-à-pair. En effet, suivant l'environnement où ces modèles s'exécutent (peu ou très dynamique, homogène ou hétérogène, large échelle ou échelle moyenne, etc.), certaines architectures sont plus adaptées que d'autres.

Cependant, les mécanismes de recherche proposés par ces modèles présentent des limites dans un contexte de grille. En effet, dans le cadre d'un modèle d'architecture décentralisée non-structurée, l'inconvénient majeur est l'incomplétude des résultats de recherche. Donc, une recherche peut ne pas être fructueuse même si la ressource demandée se trouve dans le système.

Dans le cadre d'un modèle d'architecture décentralisée structurée, les DHTs ne proposent pas une recherche par mot-clefs. Une adaptation de la DHT sera ainsi nécessaire pour mieux répondre aux exigences des utilisateurs. De plus, les nœuds de ce type de système ont tous les mêmes responsabilités alors qu'ils ne présentent pas les mêmes caractéristiques en termes de fiabilité, de stabilité ou encore de capacité. Afin de fournir une meilleure qualité de service, il serait envisageable de tirer profit des diversités caractéristiques des différents nœuds d'un système.

Dans ce contexte, nous proposons un modèle qui, en plus d'exploiter les propriétés inhérentes et fondamentales qu'offrent les systèmes pair-à-pair, tendra d'apporter des solutions aux limites précédemment citées.

La solution que nous proposons, nommée P2P4GS (*Peer-To-Peer For Grid Services*), présente ainsi l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme

d'exécution de services. En fait, la couche de gestion de la grille pair-à-pair est séparée de celle de localisation et d'invocation de services. Nous faisons ainsi 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.

Le modèle que nous proposons ainsi est générique ; c'est-à-dire applicable sur toute architecture pair-à-pair. Pour garantir cette propriété de généricité, étant donné que les systèmes distribués à large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous proposons de structurer l'environnement de grille pair-à-pair en communautés virtuelles (aussi appelées groupes virtuels ou *clusters*). Cette structuration présente deux caractéristiques intéressantes, à savoir la limitation des communications et le passage à l'échelle. En effet, comme le soulignent les auteurs de [Bas99, MSC+12, SM04], une structuration efficace d'un réseau permet de garder les performances satisfaisantes même avec l'augmentation de la taille du système.

Afin de bien orchestrer les communications au sein des différentes communautés virtuelles, ainsi que dans le système, nous proposons de définir différents rôles pour les nœuds. Nous définissons ainsi des nœuds appelés proxys qui vont assurer collectivement la gestion des ressources partagées dans le système.

Soulignons par ailleurs que le système P2P4GS permet aussi bien le déploiement, la localisation et l'invocation de services, tout en respectant le paradigme des systèmes pair-à-pair. Le déploiement, comme l'invocation, sont totalement délégués à la plateforme et l'accès aux ressources se fait de manière transparente pour l'utilisateur.

La suite de ce chapitre est organisée de la manière suivante. Dans la Section 3.2, nous présentons les spécifications de P2P4GS. Pour ce faire, nous définissons d'abord les principaux concepts de base sur lesquels s'appuient le P2P4GS. Puis, nous décrivons notre modèle d'architecture. Par la suite, nous présentons dans la Section 3.3, le principe d'exécution de P2P4GS ainsi que son comportement face à la dynamique du système. Enfin, la Section 4.3 fait une synthèse de l'ensemble des travaux présentés dans ce chapitre.

3.2 Spécifications de P2P4GS

3.2.1 Concepts de bases

Dans cette section, nous définissons les principaux concepts de bases sur lesquels s'appuient notre modèle. Nous décrivons ainsi les composants principaux du système à savoir le réseau, les nœuds qui composent le réseau et la notion de service.

Le réseau de communication

Le réseau de communication que nous considérons est le réseau *overlay*, qui est mis à notre disposition par la plate-forme pair-à-pair sous-jacente. Donc c'est un réseau logique (virtuel) de recouvrement construit au-dessus du réseau physique de communications, généralement le réseau IP (voir Figure 3.1).

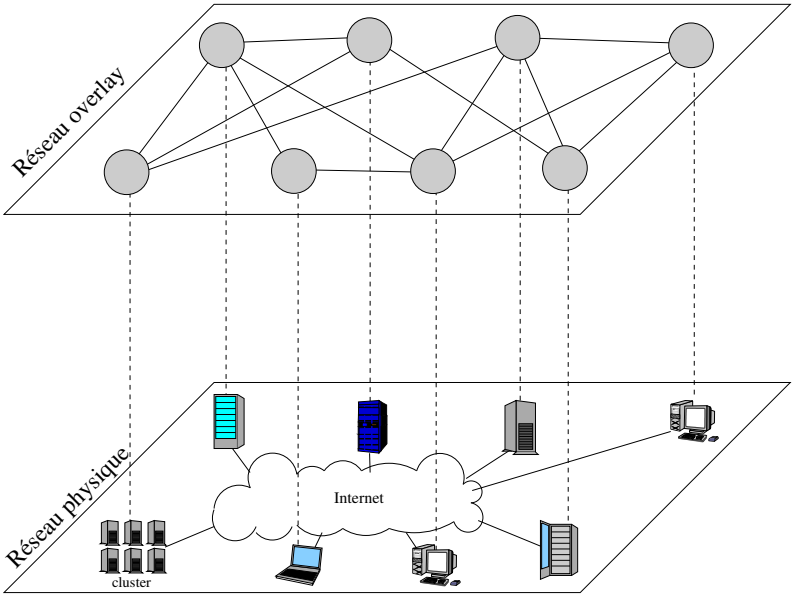


FIGURE 3.1 – Architecture du réseau de recouvrement

Le réseau recouvrant présente des propriétés de transparence vis-à-vis des nœuds du système. En effet, les nœuds qui peuvent être de types différents sur le plan matériel sont vus comme des pairs dans le réseau *overlay*.

Afin d'être exploitable dans notre spécification, on modélise ce réseau virtuel sous forme d'un graphe non-orienté et connexe noté : $G = (V, E)$ où V représente l'ensemble des nœuds du système et E l'ensemble des liens de communication pair-à-pair établis entre les différents nœuds.

Remarque 3.1. *Les solutions que nous proposons tolèrent la non-connexité du système durant un temps relativement court à la suite de changements topologiques. Au-delà d'un temps borné, un nœud non-atteignable est considéré comme déconnecté.*

Ces définitions permettent la prise en compte des différentes spécificités du réseau *overlay*. En effet, tout en faisant abstraction de la complexité du réseau physique sous-jacent, grâce aux propriétés fondamentales des systèmes pair-à-pair, deux nœuds peuvent communiquer soit directement comme étant des voisins réels, soit via des nœuds relais du

réseau recouvrant à l'aide d'un routage de proche en proche. Cette communication peut ainsi passer par un ou plusieurs liens physiques du réseau de communication sous-jacent.

Les nœuds du système

L'environnement de grille pair-à-pair est constitué par un ensemble de nœuds coopérant pour l'accomplissement de la tâche globale du système. Ces nœuds sont virtuellement vus comme des pairs et peuvent être de types différents sur le plan matériel. Un nœud peut être une machine puissante (un supercalculateur par exemple), un ordinateur de bureau ou une station de travail, ou un site tel qu'un centre de calcul ou un cluster.

Chaque nœud détient un identifiant unique dans le réseau *overlay*. Cet identifiant est fourni par le protocole pair-à-pair sous-jacent.

Les nœuds sont en charge de la gestion locale du réseau et assurent collectivement les tâches de localisation et d'invocation de service. 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.

Pour assurer ces fonctions, chaque nœud détient son propre "*Registre de services*", qui répertorie l'ensemble des services déployés localement. En outre, comme nous le verrons par la suite, en fonction du rôle du nœud, ce registre répertorie aussi les services localisés dans le système.

Concept de service

Un service est une action exécutée par un nœud du système (vu du côté demandeur de service comme serveur) à l'attention d'un utilisateur (un autre nœud du système ou une machine externe au système) aussi appelé client.

Le service peut être un objet, une donnée, une ressource, une application ou un composant logiciel ayant une fonction bien définie. Il possède une interface qui est accessible par les utilisateurs via des requêtes et est caractérisé par :

- les ressources nécessaires à son exécution (CPU, RAM, etc.) ;
- le format et les données nécessaires lors de l'invocation du service ;
- le format et les contraintes des données résultats.

L'utilisateur peut ainsi accéder à une ou plusieurs ressources. Ces ressources peuvent êtres localisées sur un même nœud ou distribuées géographiquement entre plusieurs nœuds du système.

3.2.2 Modèle d'architecture

Après avoir décrit nos motivations pour un modèle destiné aux applications de grille pair-à-pair, et après avoir spécifié des différents éléments du système P2P4GS, nous présentons son architecture dans cette section.

Nous proposons un modèle d'architecture constitué de quatre couches d'abstraction. Ces couches superposées mettent en évidence les différents mécanismes sous-jacents à l'environnement de grille pair-à-pair ainsi que les interactions entre les différentes entités du système. Au niveau de chaque couche, un certain nombre de tâches requises pour le fonctionnement global du système sont réalisées et fournies aux couches supérieures.

La Figure 3.2 présente l'architecture du système P2P4GS.

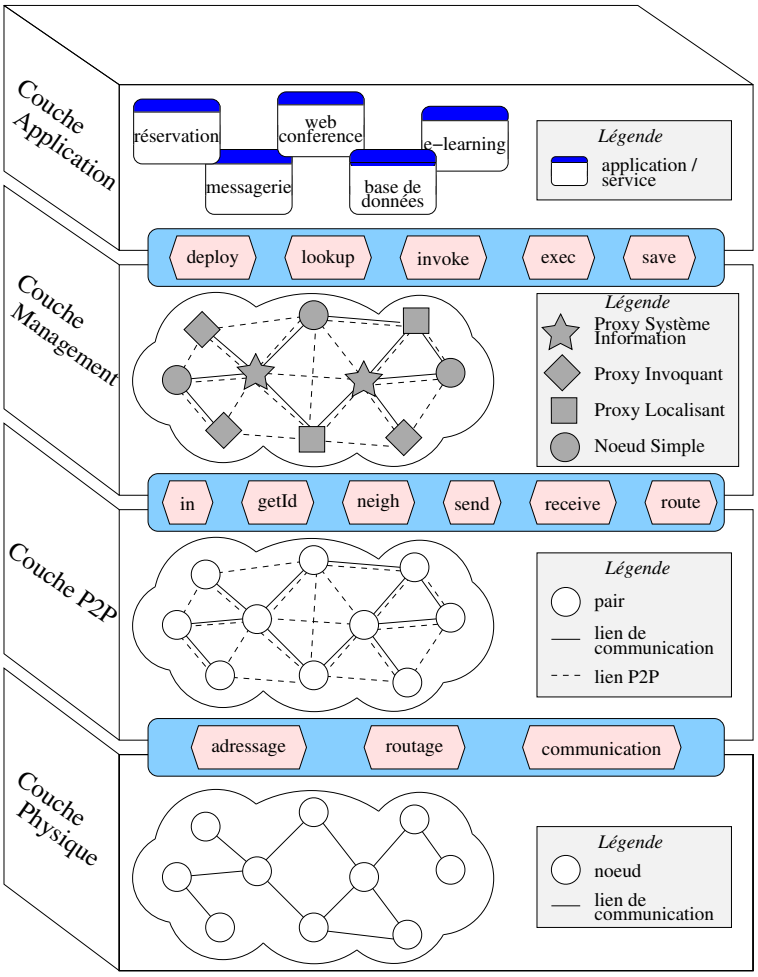


FIGURE 3.2 – Architecture du système P2P4GS

Dans ce qui suit, nous décrivons les différentes couches de l'architecture.

La couche physique

C'est la couche la plus basse et elle représente le réseau physique de communication qui est généralement Internet. Ce réseau est généralement Internet. Les nœuds d'une grille peuvent toutefois être interconnectés à travers des réseaux haut-débit dédiés. C'est le cas par exemple de la grille Grid'5000 [BCC+06] dont ses entités sont interconnectées via une infrastructure réseau dédiée à 10 Gb/s fournie par RENATER [1].

Nous modélisons cette couche sous la forme d'un graphe orienté et connexe noté : $G_1 = (V, E_1)$ où V représente l'ensemble des nœuds (ordinateurs, supercalculateurs, clusters, etc.) du système et E_1 représente l'ensemble des liens physiques (les bus, les câbles ou les connexions sans fil) entre les différentes entités du réseau physique de communication.

Ces définitions permettent ainsi la prise en considération des différents éléments :

- d'implémentation et de configuration réseau comme le NAT ;
- de sécurité comme des pare-feux qui limitent la possibilité de communication bi-directionnelle des nœuds. Par exemple, un nœud qui est situé derrière un pare-feu, peut être inaccessible pour une partie des nœuds du réseau de communications.

Cette couche fournit plusieurs services aux couches supérieures dont les fonctions d'adressage, de routage et de communication.

La couche P2P

Cette couche correspond à l'intergiciel pair-à-pair utilisé. Afin d'être exploitable dans notre spécification, cette couche est modélisée sous la forme d'un graphe non-orienté et connexe noté : $G_2 = (V, E_2)$ où V correspond toujours à l'ensemble des nœuds (appelés aussi pairs) du système et E_2 représente l'ensemble des liens virtuels de communication entre les nœuds et établis par le protocole pair-à-pair.

Ces définitions permettent la prise en compte des différentes spécificités du réseau *overlay* pair-à-pair. En effet, grâce aux propriétés inhérentes des systèmes pair-à-pair, tout en faisant abstraction de la complexité du réseau physique de communication sous-jacent, deux nœuds se trouvant derrière des pare-feux peuvent être voisins et ainsi communiquer dans le réseau recouvrant.

Le réseau P2P possède en effet des mécanismes de nommage, d'adressage, de communication, de routage, etc. Ces mécanismes sont en général distincts du réseau de communications utilisé.

Cette couche fournit ainsi des fonctionnalités de communication et de maintenance de la topologie du réseau. On suppose que le système pair-à-pair offre les primitives basiques de communication suivantes :

1. https ://www.renater.fr/

- **in()** cette primitive est exécutée par tout nouveau nœud connecté au système ;
- **getId()** pour récupérer son identifiant fourni par la couche P2P ;
- **neigh()** pour récupérer la liste de tous ses voisins dans l'*overlay* P2P ;
- **send(id, message)** pour envoyer un message à un nœud d'identifiant *id* du système ;
- **receive()** pour reçevoir un message en provenance d'un nœud du système ;
- **route(id, message)** pour router le message vers une destination.

On peut remarquer que la primitive *out()* devant permettre à un nœud qui quitte le système de prévenir ses voisins n'est pas prise en compte. En effet, cette fonctionnalité n'est pratiquement pas implémentée au niveau des protocoles pair-à-pair. Ainsi, ce sont des mécanismes de détection de pannes qui sont généralement mis en œuvre pour la gestion de la déconnexion de nœud.

En conséquence, tout système pair-à-pair assurant ces fonctionnalités basiques pourra être exploité par notre spécification.

La couche management

Cette couche constitue le cœur de la spécification. En effet, c'est au niveau de cette couche que toutes les opérations de gestion d'un service sont définies. Ces opérations vont du déploiement d'un service jusqu'à son exécution, tout en passant par sa publication, sa recherche et son invocation.

Tout comme la couche P2P, cette couche est modélisée sous la forme d'un graphe non-orienté et connexe noté : $G_3 = (V, E_3)$ où V correspond toujours à l'ensemble des nœuds du système et E_3 représente l'ensemble des liens virtuels de communication entre ces différents nœuds.

Cette couche hérite des propriétés inhérentes offertes par le réseau *overlay* pair-à-pair. Afin de structurer le réseau en communautés virtuelles, les primitives de communication offertes par la couche P2P sont exploitées. Nous les classifions comme suit :

La signalisation. La primitive *in*() permet de détecter la connexion d'un nouveau nœud dans le système. Cette fonction permet au nœud de définir son statut (type de nœud) après échange d'informations avec ses voisins.

Le nommage. La primitive *getId*() permet de récupérer l'identifiant du nouveau nœud connecté. Cet identifiant (unique) est fourni par le protocole P2P.

Le voisinage. La primitive *neigh*() permet de récupérer la liste des voisins d'un nœud.

La communication. Les primitives *send*(), *receive*() et *route*() permettent à un nœud de communiquer avec d'autres nœuds du système.

Ces différentes fonctions permettent ainsi, sur la base d'un algorithme de structuration, de définir les différents rôles (ou statuts) des nœuds du système afin de former les communautés virtuelles (aussi appelées *clusters*).

Vu la nature dynamique d'un environnement de grille pair-à-pair, des mécanismes de gestion des services sont nécessaires pour conserver la consistance d'une telle organisation. Ainsi, au sein de chaque communauté virtuelle, un nœud spécifique appelé PSI (*Proxy Système d'Information*) joue le rôle d'annuaire ou registre de services. Un PSI connaît ainsi la localisation de l'ensemble des services partagés de sa communauté. C'est le PSI qui assure la gestion de sa communauté et notamment des nœuds membres de celle-ci aussi appelés NS (*Nœuds Simples*).

La gestion des services, et précisément du cycle de vie des services, inclut plusieurs tâches complexes telles que la gestion du déploiement, la gestion de la localisation et la gestion de l'invocation ainsi que de l'exécution. En vue de ne pas surcharger les PSI, nous proposons de répartir certaines tâches sur d'autres types de nœuds distingués. Ainsi, les tâches d'invocation et d'exécution de services sont déléguées à des nœuds appelés PI (*Proxys Invoquant*). Étant donné que les services n'ont pas les mêmes contraintes d'exécution en termes de CPU, RAM, plateforme d'exécution, etc., un nœud est dit proxy invoquant pour un service S_i donné si et seulement si :

i) il connaît sa localisation ;

ii) il respecte ses contraintes d'exécution, c'est-à-dire s'il possède la partie cliente (*stub*) du service S_i.

Lorsqu'un nœud connait la localisation d'un service S_i (condition *i)*) mais ne satisfait pas la condition *ii)*, alors il sera nommé PL (*Proxy Localisant*) pour ce service.

Le rôle des proxys invoquant et localisant est de décharger leur PSI et aussi permettre une découverte ou une invocation ultérieure rapide. En effet, un nœud simple devient proxy invoquant ou localisant à la suite d'une première sollicitation d'un utilisateur externe. En fait, puisqu'un nœud simple ne participe pas au processus de découverte d'un service dans la communauté, seul un utilisateur externe, dont le fonctionnement du système est totalement transparent, peut passer par ce nœud (soit parce qu'il connait son adresse, soit il est géographiquement plus proche, soit il est connu pour sa fiabilité, etc.), considéré ainsi comme "*point d'entrée*" pour solliciter un service au système. Comme il est probable que l'utilisateur revienne dans le système pour solliciter le même service, le proxy interrogé pourra répondre plus rapidement.

Remarque 3.2. *Afin d'éviter une surcharge en mémoire des nœuds proxys, nous proposons de supprimer la connaissance sur la localisation d'un service à la fin d'un compte-à-rebours que nous notons $\mathcal{T}_{Live}$. De ce fait, un service qui est sollicité assez rarement dans*

le système ne va pas être mémorisé de manière indéfinie. Par contre, un nœud reste proxy invoquant ou proxy localisant pour un service assez fréquemment sollicité, du moment où son $\mathcal{T}_{Live}$ sera réinitialisé après chaque nouvelle demande.

Les nœuds de chaque communauté virtuelle vont ainsi coopérer pour assurer la gestion locale des ressources afin d'offrir un ensemble de primitives à la couche supérieure (couche application). Ces fonctions sont *deploy*, *lookup*, *invoke*, *exec* et *save* pour respectivement le déploiement, la localisation, l'invocation, l'exécution et la mémorisation d'un service. Nous verrons l'étude des opérations de ces différentes primitives dans le prochain chapitre.

La couche application

C'est la couche la plus haute. Elle sert d'interface aux utilisateurs pour l'accès aux services qu'offre l'environnement de grille pair-à-pair. En effet, les primitives de la couche sous-jacente sont exploitées par les différentes plate-formes avec lesquelles elles interagissent afin d'offrir des services à la couche application. Soulignons que l'utilisateur accède de manière transparente aux services de la grille pair-à-pair.

3.3 Approche de structuration du système en communautés

Dans cette section, nous présentons d'abord l'approche de structuration que nous proposons. Par la suite, nous décrivons son principe d'exécution ainsi que les mécanismes d'adaption à la dynamique du système.

3.3.1 Présentation de la solution de structuration

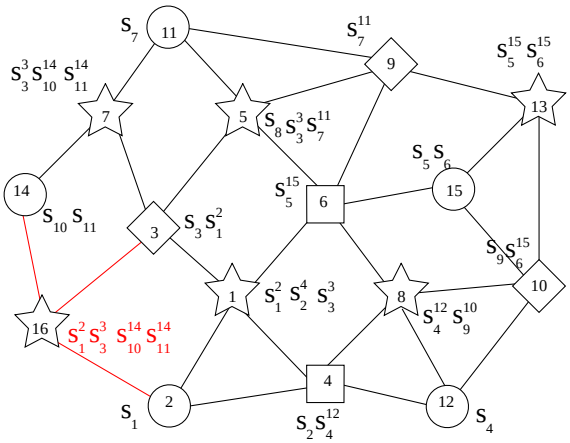
L'approche de structuration proposée est complètement distribuée. Elle se base uniquement sur le voisinage des nœuds pour l'élection des nœuds PSI et ainsi la formation des différentes communautés virtuelles (*clusters*). Son originalité est qu'elle se rapproche le plus possible des conditions réelles. En effet, dans un environnement à grande échelle où les nœuds sont géographiquement dispersés, le réseau ne peut pas se former de manière spontanée. Ainsi, nous considérons que le réseau n'est pas créé à l'avance. Par conséquent, nous proposons d'élire les PSI au fur et à mesure de la connexion des nœuds en se basant sur leurs voisinages.

Afin de doter le système des mécanismes de contrôle sur la distribution des *clusters*, nous introduisons le critère de degré minimal de connexion dans le processus d'élection des nœuds PSI. Ainsi pour être candidat potentiel à l'élection de PSI, un nœud doit avoir

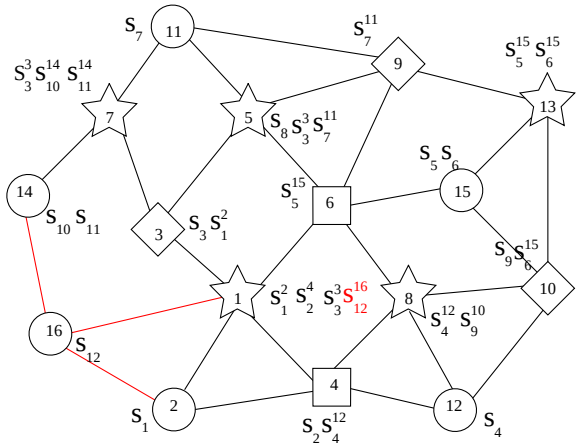
un nombre minimal de voisins que nous notons $\Delta_{RequiredMinDegree}$. De ce fait, ce sont les nœuds les plus stables et les plus distingués, vraisemblablement les nœuds qui ont une grande réputation, qui auront le plus de chance d'être PSI. Une fois cette condition vérifiée, un nœud devient PSI s'il n'a pas de PSI dans son voisinage. Par contre, si un nœud a au moins un PSI dans son voisinage, alors il devient NS. La Figure 3.3 donne une illustration de ces deux cas de situation. Pour simplifier, le $\Delta_{RequiredMinDegree}$ est fixé à 3.

Lorsqu'un NS se lie avec un PSI, il lui envoie la liste de ses services indexés ainsi que la liste de ses autres voisins PSI (s'il en a évidemment). Ces informations permettront au PSI d'alimenter son registre de services ainsi que sa table de routage.

Soulignons que dans l'approche de structuration proposée, un NS peut avoir plusieurs PSI dans son voisinage. L'intérêt principal de cette technique est la redondance (réplication) des catalogues de service. Ce qui améliore la rapidité de la recherche et augmente le degré de tolérance de pannes des PSI.



(a) Le nœud 16 a comme voisins, les nœuds 2, 3 et 14, qui ont tous un statut NS. Puisqu'il a le degré minimal requis, il devient PSI et informe ses voisins. Ces derniers lui envoient la liste de leurs services hébergés ainsi que la liste des autres PSI auxquels ils sont voisins. Le PSI met alors à jour son registre de services et sa table de routage.



(b) Le nœud 16 a comme voisins, les nœuds 1, 2 et 14. Puisqu'il a au moins un voisin PSI (nœud 1), il devient alors NS. Par la suite, étant donné qu'il n'a pas d'autres voisins PSI à ce stade de structuration, il envoie à son PSI juste la liste de ses services hébergés (dans cet exemple, S_{12}). Le PSI (nœud 1) met alors à jour son registre de services.

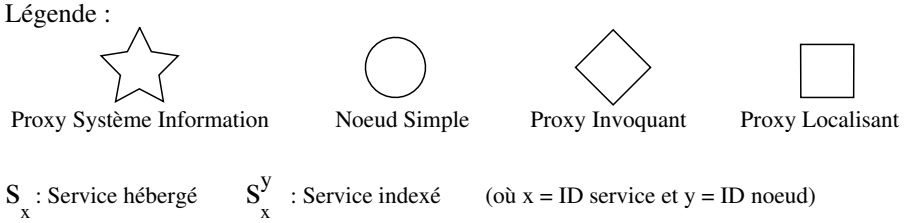


FIGURE 3.3 – Connexion d'un nœud dans le système

D'autre part, en vue de répondre à la problématique de découverte de services, nous proposons de construire un arbre couvrant constitué uniquement des PSI ainsi formés. En effet, lorsqu'un le nouveau PSI met à jour sa table de routage à partir des informations reçues de ses différents voisins NS, il choisit dans cette table le PSI qui a numériquement le plus petit identifiant et se lie avec lui dans l'arbre couvrant. La particularité de l'approche proposée est que la construction de l'arbre couvrant se fait dans la même phase que la structuration du système. Ce qui permet de minimiser le coût des communications en termes de messages. Les requêtes de recherche sont ainsi acheminées le long de l'arbre couvrant qui permet une recherche exhaustive puisque tous les services partagés dans le système sont indexés au niveau des différents PSI constituant cet arbre.

La Figure 3.4 illustre un exemple de réseau structuré. L'arbre couvrant est matérialisé par les liens en pointillés bleus.

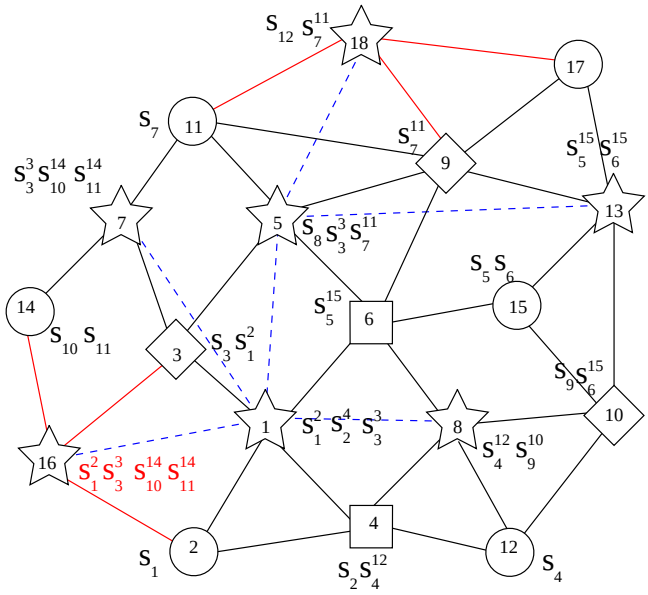


FIGURE 3.4 – Evolution de la structuration et choix d'un PSI passerelle

Dans cet exemple, le PSI 16 a dans sa table de routage, les PSI 1, 5 et 7. Ainsi, il choisit comme PSI passerelle le nœud 1 et se lie avec lui dans l'arbre couvrant. De même, le PSI 18 a dans sa table de routage, les PSI 5, 7 et 13. Il choisit et se lie ainsi avec le PSI 5 dans l'arbre couvrant.

Remarque 3.3. *Précisons que les statuts PI (Proxy Invoquant) et PL (Proxy Localisant) n'interviennent pas dans le processus de structuration ; mais plutôt dans le processus de localisation de service. En effet, comme nous l'avons souligné dans la section précédente, c'est suite à une opération de localisation d'un service qu'un nœud, en fonction de ses ressources, devient temporairement PI ou PL pour ce service.*

3.3.2 Algorithme de structuration

Dans cette section, nous décrivons l'algorithme de structuration du système P2P4GS. Pour cela, nous présentons dans le Tableau 3.1 les principales notations utilisées.

Paramètres
– $\Delta_{RequiredMinDegree}$: degré minimal requis.
– $\mathcal{T}_{timeout}$: temps d'attente pour effectuer un traitement.
Variables
– id_u : identifiant du nœud u.
– $statut_u$: statut du nœud u; avec $statut_u \in \{UNDEF, NS, PSI\}$.
– $neighTable_u$: table de voisinage du nœud u. Elle reçoit des couples (id, $statut$).
– updateNeighStatus(id_v, $statut_v$) : mettre à jour le statut de son voisin v.
– $numberOfResponses$: compteur initialisé à 0.
– $psiList_u$: liste des voisins PSI du nœud u.
– $serviceList_u$: liste des services hébergés par le nœud u.
– $serviceRegistry_u$: registre de services du nœud u.
– $routingTable_u$: table de routage du nœud u.
– $gatewayTable_u$: table des PSI passerelles du nœud u.
Structure des messages
– queryStatus(id, $statut$) : demander le statut de son voisin.
– responseStatus(id, $statut$) : répondre à un message $queryStatus$.
– updateStatus(id, $statut$) : envoyer son statut à son voisin.
– clusterManagement($serviceList$, $psiList$) : envoyer à son PSI la liste des services indexés ainsi que la liste de ses autres voisins PSI.
– masterRouteManagement(id) : notifier au nœud destinataire (PSI) qu'il est un nœud passerelle.

TABLE 3.1 – Paramètres, variables et structure des messages de P2P4GS

Notre approche de structuration se base uniquement sur le voisinage pour élire les PSI au fur et à mesure de la connexion des nœuds. Ainsi, lorsqu'un nœud se connecte dans le système, il établit son voisinage qui dépend du protocole P2P implémenté et qui évolue en cours d'exécution du système.

Initialement, le statut d'un nouveau nœud connecté est *UNDEF*.

Lorsqu'un nœud u de statut PSI ou NS établit un nouveau voisinage avec un nœud v de statut quelconque, il lui envoie son statut à travers un message *updateStatus*. Le nœud v met alors à jour le statut de son voisin ($updateNeighStatus$).

Si le nœud u de statut *UNDEF* a au moins un voisin PSI, alors il devient NS et informe ainsi ses voisins de son nouveau statut. Dans ce cas, s'il a des services hébergés, alors il envoie la liste de leur index (message *clusterManagement*) à l'ensemble de ses voisins PSI (Algorithme 1). Si par contre le nœud u de statut *UNDEF* a un nombre de voisins qui atteint le degré minimal requis et qu'il n'a pas de PSI dans son voisinage, alors il devient PSI (Algorithme 1). Par la suite, il envoie à ses voisins un message *updateStatus* pour leurs informer de son nouveau statut.

Algorithme 1: À la réception d'un message responseStatus(id_v, $status_v$) du nœud v

```
1  updateNeighStatus(id_v, status_v) ; /* Mise à jour du statut de v                */
2  numberOfResponses ⟵ numberOfResponses + 1;
3  if statut_u = UNDEF ∧ numberOfResponses = Δ_RequiredMinDegree then
4  |  if ∃ k ∈ neighTable / statut_k = PSI then
5  |  |  statut_u ⟵ NS;
6  |  |  forall the k ∈ neighTable / statut_k = PSI do
7  |  |  |  send(k, clusterManagement(ServiceList_u, psiList_u));
8  |  |  end
9  |  else
10 |  |  statut_u ⟵ PSI;
11 |  end
12 |  forall the k ∈ neighTable do
13 |  |  send(k, updateStatus(id_u, status_u)) ; /* Envoyer à k mon statut       */
14 |  end
15 else
16 |  if statut_u = NS ∧ statut_v = PSI then
17 |  |  send(v, clusterManagement(ServiceList_u, psiList_u));
18 |  end
19 end
```

Lorsqu'un nœud u reçoit un message *updateStatus* (Algorithme 2), il met à jour le statut de l'émetteur. Après cette phase, le nœud u doit vérifier sa cohérence. Ainsi, si son état est *UNDEF* et que le statut de l'émetteur est PSI, alors il devient obligatoirement NS et informe ainsi ses voisins de son nouveau statut. D'autre part, si le statut du nœud u est (ou devient) NS alors, s'il a des services hébergés et qu'il a d'autres voisins PSI (le cas où son statut était déjà NS avant la réception de message), il envoie au PSI un message *clusterManagement* contenant ces informations.

Une fois ses voisins informés, le nouveau PSI déclenche un compte-à-rebours ($\mathcal{T}_{timeout}$) de réception des messages *clusterManagement* provenant des nœuds de son voisinage. Ainsi, à la réception de chaque message *clusterManagement* (Algorithme 3), le PSI met à jour son registre de services (*serviceRegistry*) et sa table de routage (*routingTable*).

Algorithme 2: À la réception d'un message updateStatus(id_v, $status_v$) du nœud v

```
1  updateNeighStatus(id_v, status_v) ; /* Mise à jour du statut de v */
2  if statut_u ≠ PSI ∧ statut_v = PSI then
3  |  if statut_u = UNDEF then
4  |  |  statut_u ⟵ NS;
5  |  |  forall the k ∈ neighTable do
6  |  |  |  send(k, updateStatus(id_u, status_u)) ; /* Envoyer à k mon statut */
7  |  |  end
8  |  end
9  |  if statut_u = NS then
10 |  |  send(v, clusterManagement(ServiceList_u, psiList_u));
11 |  end
12 end
```

Lorsque le $\mathcal{T}_{timeout}$ expire, le nœud u PSI définit sa passerelle dans l'arbre couvrant sur la base des informations contenues dans sa table de routage. Pour ce faire, il choisit dans sa table de routage le PSI qui a numériquement le plus petit identifiant et l'ajoute dans sa table de passerelles (*gatewayTable*). Par la suite, il envoie à ce PSI un message *masterRouteManagement* afin de lui notifier ce choix.

À la réception d'un message *masterRouteManagement* sur un PSI v, il exécute l'algorithme 4. Il ajoute alors l'identifiant du PSI source comme une nouvelle entrée dans sa table de passerelles (*gatewayTable*).

Algorithme 3: À la réception d'un clusterManagement($serviceList_v$, $neighList_v$) ou à l'expiration de $\mathcal{T}_{timeout}$

```
1  if ¬(T_timeout expire) then
2  |  forall the S_i ∈ serviceList_v do
3  |  |  serviceRegistry ⟵ serviceRegistry ∪ {(S_i, id_v)};
4  |  end
5  |  forall the q ∈ neighList_v / statut_q = PSI do
6  |  |  if ∄ k = (id_k) ∈ routingTable / id_k = id_q then
7  |  |  |  routingTable ⟵ routingTable ∪ {id_q};
8  |  |  end
9  |  end
10 else
11 |  if |gatewayTable| = 0 ∧ |routingTable| > 0 then
12 |  |  gatewayTable ⟵ gatewayTable ∪ {id_k / id_k = min(routingTable)};
13 |  |  send(k, masterRouteManagement(id_u));
14 |  end
15 end
```

Algorithme 4: À la réception d'un message masterRouteManagement(id_v) du nœud v

```
1 if ∄ k = (id_k) ∈ gatewayTable / id_k = id_v then
2 |   gatewayTable ⟵ gatewayTable ∪ {id_v};
3 end
```

3.3.3 Mécanismes d'adaptation à la dynamique du système

Au cours de l'évolution d'un système distribué, des modifications topologiques peuvent se produire à tout moment. Une modification topologique se traduit par l'apparition (connexion) ou la disparition (déconnexion) de nœuds au niveau du système. Il est par conséquent nécessaire de mettre en place des mécanismes d'adaptation afin d'assurer l'évolutivité du système.

Notre approche de structuration propose d'élire les PSI au fur et à mesure de la connexion des nœuds en se basant sur leurs voisinages. Le système P2P4GS s'adapte donc à la connexion de nœuds. Les mécanismes que nous mettons en place vont ainsi s'intéresser à la déconnexion de nœuds.

Les déconnexions de nœuds peuvent être volontaires dans le cas par exemple d'un service accompli ou bien involontaires quant il s'agit de nœuds défaillants ou mobiles. En outre, les canaux de communication peuvent également être non fonctionnels pour une période donnée. Par conséquent, nous considérons qu'un nœud est en panne, lorsqu'il n'est plus joignable après un certain temps prédéfini et configurable. Ce temps est généralement appelé *délai de garde* et nous le notons $\mathcal{T}_g$.

Dans ce qui suit, nous présentons d'abord notre modèle de détection de pannes. Par la suite, nous décrivons le modèle de tolérance aux pannes.

Modèle de détection de pannes

Une technique de gestion de pannes comprend généralement un mécanisme chargé de les détecter et de les gérer d'une manière transparente. Les pannes peuvent être détectées par des messages périodiques de type *ping-pong* ou *heartbeat* [CT96, LDPC10, HN15].

Nous utilisons deux mécanismes pour la détection de pannes dans le système : un mécanisme proactif pour la détection de pannes de nœuds PSI et un mécanisme réactif pour la détection de pannes de nœuds NS.

Dans le cas de la détection de pannes de nœuds PSI, nous utilisons des messages de type *heartbeat*. Ainsi, les PSI envoient périodiquement des messages *heartbeats* à leurs voisins afin de signaler leur présence. Au niveau des nœuds NS, un délai de garde ($\mathcal{T}_g$) est associé au message *heartbeat* en attente de chacun de leurs PSI.

Le principe du détecteur *heartbeat* est le suivant : chaque PSI émet périodiquement un message *heartbeat* vers l'ensemble de ses voisins. À la réception d'un tel message, le voisin u réinitialise son temps de garde. Ainsi, si le voisin u ne reçoit pas un message *heartbeat* en provenance du PSI v, jusqu'à l'expiration du temps de garde, alors u va considérer ce PSI comme défaillant.

Dans le cas de la détection de pannes de nœuds NS, nous utilisons un mécanisme réactif. En effet, vu la nature dynamique des environnements de grille pair-à-pair, les nœuds simples, aussi qualifiés de nœuds feuilles, ont généralement un caractère volatile. Nous admettons ainsi que maintenir des messages contrôle des nœuds NS peut dégrader les performances du système. De ce fait, un PSI détecte la déconnexion de son voisin NS que si son service est demandé.

Modèle de tolérance aux pannes

Nous décrivons dans cette section, le comportement du système P2P4GS lorsqu'une panne d'un nœud PSI ou NS est détectée dans le système.

Rappelons que les statuts PI et PL n'interviennent pas dans le processus de structuration ; mais plutôt dans le processus de localisation de service. Un nœud PSI ou NS peut être temporairement PI ou PL pour un quelconque service.

a) Panne d'un PSI

Lorsqu'un PSI disparait, alors chacun de ses voisins u mettra à jour sa table de voisinage après le délai de garde associé à ce PSI. Par la suite, chaque voisin u vérifie s'il a au moins un PSI dans sa table, auquel cas il ne fait rien. Si par contre un voisin u n'a plus de voisin PSI et qu'il possède un nombre de voisins qui atteint le degré minimal requis ($\Delta_{RequiredMinDegree}$), il devient candidat potentiel à l'élection de PSI. En conséquence, c'est le nœud qui a le plus fort degré parmi les candidats potentiels qui sera élu PSI.

Le processus d'élection est décrit comme suit (voir Algorithme 5) : chaque nœud u candidat potentiel envoie son degré (nombre de voisins) à l'ensemble de ses voisins ; puis déclenche son $\mathcal{T}_{timeout}$. Lorsque u reçoit en réponse un degré inférieur, il reste candidat. Si par contre, il reçoit un degré supérieur alors, il est battu.

Le nœud qui a le plus fort degré se déclare PSI et informe ses voisins selon le même processus décrit dans la section précédente.

Remarque 3.4. *Comme nous venons de le préciser dans la section précédente, un NS peut avoir plusieurs PSI dans son voisinage. Il y a ainsi une redondance des catalogues de service. Ce qui augmente par conséquence le degré de tolérance de pannes des PSI.*

Algorithme 5: À la réception d'un message responseElection(id_v, $degree_v$) ou à l'expiration du *timeout*

```
1 if T_timeout expire then
2 |   if etat_u = CandidatPotentiel then
3 |   |   statut_u ⟵ PSI;
4 |   end
5 else
6 |   if degree_v > degre_u then
7 |   |   etat_u ⟵ Battu;
8 |   end
9 end
```

b) Panne d'un NS

Un PSI détecte la déconnexion de son voisin NS que si un service de ce dernier est demandé. Dans ce cas, un *timeout* ($\mathcal{T}_{Live}$) est déclenché. Lorsque ce dernier expire sans que le nœud soit joignable, alors le PSI supprime les informations sur ce nœud ainsi que celles de ses services.

Étant donné que la recherche se fait par mots-clefs, l'utilisateur peut avoir le choix parmi plusieurs autres sources, en cas d'indisponibilité d'un service. Afin d'assurer une meilleure disponibilité d'un service, nous envisageons dans nos travaux futurs de le répliquer selon un facteur qui dépendra de sa réputation.

3.4 Synthèse

Dans ce chapitre, nous avons présenté notre modèle, nommé P2P4GS (*Peer-To-Peer For Grid Services*), pour la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle. Ce modèle présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme d'exécution de services. Il permet aussi bien le déploiement, la localisation et l'invocation de services tout en respectant le paradigme des réseaux P2P.

De plus, il est générique c'est-à-dire applicable sur toute architecture pair-à-pair. Pour garantir cette propriété, étant donné que les systèmes distribués à large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous avons proposé de structurer le système de grille pair-à-pair en communautés virtuelles (*clusters*). Notre approche de structuration est complètement distribuée et se base uniquement sur le voisinage des nœuds pour l'élection des PSI et la formation des différentes communautés virtuelles. Cette structuration présente deux caractéristiques intéressantes, à savoir la limitation des communications et le passage à l'échelle.

En outre, afin de permettre une recherche efficace dans l'environnement de grille pair-à-pair, un arbre couvrant constitué uniquement des PSI est maintenu. La particularité de cette approche est que la construction de l'arbre couvrant se fait en même temps que la structuration du système. Ce qui permet de minimiser le coût des communications en termes de messages.

Vers l'unification de systèmes de grilles pair-à-pair

Outre les caractéristiques qu'offre le système P2P4GS, nous comptons exploiter une autre propriété qu'elle pourra garantir : l'unification de plusieurs systèmes de grilles pair-à-pair isolés. En effet, vu la structure modulaire de son architecture, sa propriété de "généricité" et son couplage faible avec le protocole P2P sous-jacent pour former le réseau de recouvrement, il suffira d'adapter la spécification en ajoutant quelques modules ou fonctionnalités comme :

- un mécanisme de nommage qui est propre à la spécification en vue d'assurer l'unicité des identifiants des nœuds issus des différents réseaux ;
- un mécanisme de maintien de liens virtuels entre des nœuds se trouvant dans des réseaux pair-à-pair différents.

La gestion des ressources est un mécanisme essentiel dans tout système distribué et plus particulièrement pour un environnement de grille.

Le prochain chapitre sera ainsi consacré à l'étude de la gestion de ressources dans un environnement de grille pair-à-pair.

Chapitre 4

Gestion des services dans P2P4GS

Résumé. *Dans le chapitre précédent, nous avons défini les spécifications de P2P4GS. On a ainsi décrit son modèle d'architecture ainsi que l'approche de structuration du système P2P4GS afin qu'il puisse supporter toute architecture pair-à-pair.*

Nous présentons dans ce chapitre les mécanismes de gestion de services qu'offre le système P2P4GS. Pour ce faire, dans un premier temps, nous décrivons les motivations et présentons les objectifs quant à la gestion de ressources dans un tel environnement. Dans un second temps, nous définissons les différentes primitives pour la gestion de services dans P2P4GS. Ainsi, les différentes opérations qui entrent en jeu dans le cycle de vie d'un service, à savoir le déploiement, l'enregistrement (ou publication), la localisation, l'invocation et l'exécution, seront décrites dans cette partie.

Les travaux que nous présentons dans ce chapitre sont publiés dans des revues et conférences avec actes et comités de sélection : [GFRN13, GFNR14, GFRN14a, GFRN16b, GFRN16a]

Sommaire

4.1 Motivations 84
4.2 Primitives de gestion de services 85
4.2.1 Formalisme de notations 85
4.2.2 Service de déploiement : `deploy` 86
4.2.3 Enregistrement d'un service : `save` 90
4.2.4 Service de découverte : `lookup` 91
4.2.5 Invocation et exécution de service : `invoke` et `exec` 94
4.3 Synthèse 95

4.1 Motivations

Comme nous l'avons précisé dans le chapitre précédent, la gestion de ressources est un mécanisme essentiel dans tout système distribué, et plus particulièrement pour un environnement de grille pair-à-pair où les ressources mises en commun sont souvent de nature très hétérogènes et dynamiques. La découverte de ressources constitue notamment un des défis essentiels dans un tel environnement. En effet, la grille étant une agrégation de ressources géographiquement dispersées et disponibles dans plusieurs organisations (Universités, entreprises, etc.), sous formes de centres de calcul, rechercher et localiser un service résolvant un besoin spécifique devient un enjeu majeur.

Outre les caractéristiques concernant la diversité et la dynamique des ressources, les systèmes de grilles présentent également des caractéristiques de large échelle. L'accès aux ressources réparties doit se faire de manière transparente du point de vue utilisateur.

En conséquence, il est nécessaire de fournir des mécanismes efficaces de gestion des ressources de la grille pour permettre leur utilisation rationnelle. C'est pour répondre à ces problématiques que nous avons proposé le système P2P4GS pour la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle.

L'intergiciel ainsi proposé est générique et supporte l'hétérogénéité des ressources (puissances de calcul diverses des nœuds et réseau d'interconnexion hétérogène). Pour parvenir à cela, nous avons défini un ensemble de spécifications sur lesquelles s'appuie le système P2P4GS. Ainsi, après avoir défini les principaux composants, ainsi que le modèle d'architecture, nous avons proposé une approche de structuration du système de grille pair-à-pair en communautés virtuelles en vue de garantir les propriétés de généricité, d'auto-adaptativité, d'hétérogénéité et de passage à l'échelle.

D'autre part, dans le processus de structuration, un arbre couvrant constitué uniquement des PSI ainsi formés est maintenu. L'arbre couvrant est construit dans le but d'orchestrer les communications entre communautés virtuelles (clusters) mais aussi pour l'utiliser pendant le processus de découverte de services. Ainsi, les requêtes de recherche vont être acheminées le long de cet arbre. Ceci permet une recherche exhaustive du moment où tous les services partagés dans le système sont indexés au niveau des différents PSI constituant l'arbre couvrant.

Nous présentons dans ce chapitre les mécanismes de gestion de services qu'offre le système P2P4GS.

Dans la Section 4.2, nous décrivons les différentes primitives et opérations de gestion de services. Les services de déploiement, d'enregistrement (publication), de localisation, d'invocation et d'exécution seront ainsi présentés dans cette partie. Enfin, une synthèse de l'ensemble des travaux présentés dans ce chapitre sera faite dans la Section 4.3.

4.2 Primitives de gestion de services

Dans cette section, nous présentons les mécanismes de gestion de services qu'offre la plateforme P2P4GS. Nous définissons d'abord le formalisme de notations que nous utilisons. Ensuite, nous décrivons les différentes opérations qui régissent le cycle de vie d'un service, à savoir le déploiement, l'enregistrement (ou publication), la localisation, l'invocation et l'exécution.

4.2.1 Formalisme de notations

Le Tableau 4.1 résume les principales notations que nous utilisons dans nos différents algorithmes présentés dans ce chapitre.

Paramètres
– $\mathcal{T}_{TimeoutLookup}$: temps de latence d'une recherche. La recherche s'arrête à l'expiration de ce temps. – TTL : nombre de retransmissions possible d'une requête.
Variables

– id_u : identifiant du nœud u.
– $statut_u$: statut du nœud u ; avec $statut_u \in \{UNDEF, NS, PSI\}$.
– $neighTable_u$: table de voisinage du nœud u. Elle reçoit des couples (id, $statut$).
– $serviceRegistry_u$: registre de services (annuaire) du nœud u.
– $gatewayTable_u$: table des PSI passerelles du nœud u.
– $nodeRequester$: utilisateur (externe ou nœud du système) initiant une demande de service (de déploiement ou de découverte).
– $entryPoint$: nœud point d'entrée au système par utilisateur externe.
– $distance$: nombre de sauts qui sépare une source ($nodeRequester$) et une destination (nœud responsable de la ressource).
– $constraints$: contraintes de déploiement.
– $accessParameter$: paramètres d'accès à une instance.
– $serviceQuery$: ensemble de méta-données décrivant le service recherché.
– $requestList$: gestion des requêtes sur le nœud en cours et définit par l'ensemble :
  – $keyQuery$: clé associée au service recherché ;
  – $nodeRequester$: utilisateur (nœud) demandant le service.

– $serviceFindList$: liste de services découverts répondant à la demande utilisateur.
– $resultLookup$: résultat(s) renvoyé(s) à l'utilisateur après un processus de localisation d'un service. Retourne :
  – un ensemble constitué du couple ($ServiceFindList$, $distance$) ;
  – "0 résultat(s) !" si aucun service ne satisfait la requête de l'utilisateur.
– $isSatisfied(constraints)$: retourne true si les contraintes de déploiement sont satisfaites. Sinon false.

Structure des messages
– resourceDiscovery($keyQuery$, $constraints$, $entryPoint$, TTL) : requête de découverte de ressource.
– deployCandidate($keyQuery$) : réponse d'un nœud candidat au déploiement.
– discoveryRequest($keyQuery$, $serviceQuery$, $entryPoint$, $distance$) : requête de découverte de service.
– discoveryResponse($keyQuery$, $serviceFindList$, $distance$) : réponse à la requête de recherche de service.
– executionRequest($keyQuery$, $serviceQuery$) : requête de demande d'exécution de service.
– executionResult($keyQuery$, $result$) : réponse d'un message *executionRequest*.

Table 4.1 – Paramètres, variables et structure des messages de gestion de P2P4GS

4.2.2 Service de déploiement : `deploy`

Ce service permet à un utilisateur interne ou externe du système d'allouer des ressources de la grille afin de déployer et exécuter son application.

Le déploiement dans un environnement dynamique et à grande échelle, tels que les système grilles pair-à-pair, est un processus complexe qui nécessite d'avoir un service de découverte de ressource appropriées. En effet, vu le caractère dispersé des ressources d'un tel environnement, l'administrateur de déploiement ne connait pas forcement à l'avance le nœud cible capable de supporter son déploiement. Donc la plateforme de déploiement doit être capable de fournir un moyen de spécification de contraintes matérielles et logicielles pour le déploiement de service.

Nous résumons le processus de déploiement d'un service (application, logiciel, etc.) en trois phases :

1. description des contraintes de déploiement ;
2. découverte et sélection de ressources (nœuds) satisfaisant les contraintes ;
3. déploiement effectif du service.

Dans ce qui suit, nous décrivons ces différentes phases.

Spécification des contraintes de déploiement

Dans la première étape qui correspond à la description des contraintes de déploiement, l'utilisateur spécifie simplement les ressources nécessaires à l'exécution de son service. Donc aucune description du service à déployer et aucun langage de description ne seront nécessaires dans ce modèle de déploiement.

Dans ce travail, nous proposons un ensemble de contraintes matérielles et logicielles qui sont exprimées de la manière suivante :

- `OSNeed` : pour l'expression de contraintes sur le système d'exploitation ;
- `CPUNeed` : pour l'expression de contraintes sur la puissance du processeur (en nombre de cœurs) ;
- `RAMNeed` : pour l'expression de contraintes sur la quantité de mémoire (en Go) ;
- `DISKNeed` : pour l'expression de contraintes sur la capacité de stockage (en Go) ;
- `DURATIONNeed` : pour l'expression de contraintes pour la durée de la réservation (en Jours). Les valeurs possibles vont de 1 (pour 24H) à 7 (pour une semaine).

Découverte et sélection de ressources

À la réception de la description des contraintes de déploiement, une phase de détection de nœuds de la plate-forme en mesure d'héberger et d'exécuter le service est déclenchée. Cette phase consiste à sélectionner, suivant une stratégie particulière de placement, un nœud candidat à l'hébergement du service.

Nous distinguons trois stratégies principales de placement :

- la **stratégie premier nœud** qui consiste à déployer le nouveau service sur le premier nœud détecté capable de supporter son exécution ;
- la **stratégie équilibrée** qui consiste à collecter les statistiques d'un ensemble de nœuds candidats et à tenter d'équilibrer la charge entre eux ;
- la **stratégie aléatoire** qui consiste à choisir aléatoirement parmi un ensemble de nœuds candidats, un nœud sur lequel sera déployé le nouveau service.

La stratégie **premier nœud** limite la connaissance nécessaire et réduit aussi les coûts de communications. Contrairement à cette stratégie, les stratégies **équilibrée** et **aléatoire** impliquent la connaissance de l'ensemble des nœuds candidats au déploiement. Ce qui est une tâche assez coûteuse, surtout dans un environnement à grande échelle pouvant être constitué d'un nombre élevé de nœuds. De plus, la stratégie **équilibrée** n'est pas aussi fiable vu qu'elle repose sur des informations dynamiques (charge CPU, mémoire libre, etc) qui évoluent dans le temps (par exemple entre la phase de sélection du nœud et celle de déploiement effective).

Pour toute ces raisons, nous nous intéressons dans nos travaux à la **stratégie premier nœud** vu sa simplicité.

Le principe d'exécution d'algorithme de découverte de ressources basé sur la **stratégie premier nœud** se déroule comme suit :

Tout d'abord, le nœud ayant reçu la requête de spécification de contraintes appelé nœud *point d'entrée* vérifie s'il est capable de supporter l'exécution du service (cf. Algorithme 6). Dans le cas favorable, le nœud *point d'entrée* créé une instance répondant aux spécifications de l'utilisateur ; puis envoie les paramètres d'accès à cette instance.

Si par contre les contraintes spécifiées ne peuvent pas être satisfaites, le nœud *point d'entrée* transmet la requête à ses voisins en générant, au préalable, une clé d'identification de la requête et en récupérant le paramètre de configuration TTL, qui représente le nombre de retransmissions possible de la requête.

Algorithme 6: À la réception de la description des contraintes de déploiement depuis le nœud *nodeRequester*

```
1 if isSatisfied(constraints) then
      /* Envoyer à l'utilisateur les paramètres d'accès à l'instance créée */
2     send(nodeRequester, accessParameter);
3 else
      /* Génération d'une clé d'identification de la requête               */
      /* Ajout de la clé keyQuery dans la pile de requête :                */
4     requestList ⟵ requestList ∪ {(keyQuery, nodeRequester)};
5     set timer T_TimeoutLookup;
      /* Envoie de la requête de découverte de ressources aux voisins      */
6     forall the k ∈ neighTable do
7         send(k, resourceDiscovery(keyQuery, constraints, entryPoint, TTL));
8     end
9 end
```

À la réception sur un nœud u d'une requête de découverte de ressource identifiée par une clé (*keyQuery*), ce dernier exécute l'Algorithme 7. Si la requête identifiée par cette clé est déjà reçue alors, elle est ignorée. Dans le cas contraire, la requête est traitée.

- Si le nœud u est capable de supporter l'exécution du service, il devient ainsi candidat potentiel pour le déploiement. Il envoie dans ce cas un message *deployCandidate* au nœud *point d'entrée*.
- Si par contre les contraintes spécifiées ne peuvent pas être satisfaites, alors la requête est retransmise aux voisins, sauf celui d'où elle a été reçue. La requête est ignorée dans le cas où le nœud n'a pas de voisins à qui retransmettre la requête ou si la valeur du TTL est nulle.

Algorithme 7: À la réception d'un message serviceDiscovery(*keyReq*, *constraints*, *entryPoint*, TTL) depuis le nœud v

```
1  if ∄ k = (keyQuery_k) ∈ requestList_i / keyQuery_k = keyQuery then
2      requestList_i ⟵ requestList_i ∪ {(keyQuery, nodeRequester)};
3      if isSatisfied(constraints) then
           /* Envoyer au point d'entrée les paramètres d'accès à l'instance  */
4          send(entryPoint, deployCandidate(keyQuery));
5      else
6          TTL ⟵ TTL − 1;
7          if |neighTable| > 1 ∧ TTL > 0 then
               /* Retransmettre la requête aux voisins sauf la source        */
8              forall the k ∈ neighTable / id_k ≠ id_v do
9                  send(k, resourceDiscovery(keyQuery, constraints, entryPoint, TTL));
10             end
11         end
12     end
13 end
```

Étant donné que nous utilisons la stratégie premier nœud, la première réception au *point d'entrée* d'un message *deployCandidate* provenant d'un nœud v arrête le processus de découverte de ressources. Dans ce cas, le nœud *point d'entrée* envoie au nœud v élu un message de notification. Ce dernier créé alors une instance répondant aux spécifications de l'utilisateur ; puis envoie au *point d'entrée* les paramètres d'accès à cette instance.

Par contre, si le *point d'entrée* ne reçoit aucun message *deployCandidate* jusqu'à l'expiration du temps de latence ($\mathcal{T}_{TimeoutLookup}$), alors il considère que le déploiement selon les contraintes spécifiées est impossible. La requête de l'utilisateur pour la sélection de ressources échoue alors.

Déploiement effectif du service

Une fois la phase de sélection de ressources achevée avec succès et qui se traduit alors par la réception des paramètres d'accès à l'instance créée, l'utilisateur peut alors procéder au déploiement effectif de son service.

Le déploiement inclut trois phases, à savoir l'installation, la configuration et l'exécution.

Le processus d'installation consiste à transférer sur l'instance les paquets du service, c'est-à-dire les unités élémentaires de déploiement qui contiennent toutes les ressources (bibliothèques, exécutables, fichiers de configuration, etc.) de l'application.

La configuration est l'étape qui suit l'installation. Dans cette étape, les paramètres de l'application déployée sont fixés en fonction des propriétés de la machine hôte (c'est-à-dire

l'instance). Lors de cette phase, l'installation de paquets nécessaire au bon fonctionnement de l'application peut aussi être requise.

Une fois cette étape de configuration achevée, l'application est prête à fonctionner et peut alors être exécutée. À la fin de l'exécution, l'utilisateur récupère ses résultats. À l'expiration de la durée de réservation (`DURATIONPref`), l'instance est supprimée et les ressources allouées libérées.

Nous venons de voir comment un utilisateur peut allouer des ressources partagées de types CPU, RAM, etc. afin de déployer et exécuter son propre application.

Pour permettre aux utilisateurs l'accès à d'autres types de services (logiciel, application, etc.) nous verrons dans la section qui suit comment les partager avec le mécanisme de publication.

4.2.3 Enregistrement d'un service : `save`

Il est possible d'exploiter une grille pair-à-pair par l'intermédiaire de recherche sans mémoire de services à chaque invocation. Mais comme nous l'avons vu dans notre approche, nous exploitons la notion de registre de services afin augmenter les performances de la plate-forme P2P4GS.

L'enregistrement d'un service consiste à mémoriser les informations sur ce service dans un registre de services. Cette opération se produit soit lors de la publication d'un service déployé ou soit lors de la découverte d'un service.

Nous allons dans ce qui suit, étudier ces deux cas de possibilités.

Enregistrement après déploiement d'un service

Après le déploiement d'un nouveau service sur un nœud, si le responsable décide de partager ce service, il le publie à partir de l'interface de publication. Cette interface appelle la fonction `save` en lui passant en paramètre le nom du service ainsi que ses métadonnées. La fonction génère ainsi un identifiant pour le service et enregistre le service dans le registre local de services nommé *serviceRegistry* selon la structure suivante :

SERVICE_ID	SERVICE_NAME	SERVICE_META_DATA	SERVICE_OWNER

De plus, si le nœud a le statut `NS`, alors il envoie les informations sur le service à ses différents voisins PSI qui mettront à jour leur registre de services.

Remarque 4.1. *Pour des soucis de cohérence des informations sur les services partagés, seuls les gestionnaires ou administrateurs de nœuds de la grille pair-à-pair auront accès à cette fonctionnalité.*

Enregistrement après découverte d'un service

La découverte d'un service consiste à rechercher un service dans le système. Lorsque cette recherche aboutit avec succès, le nœud du système ayant initié le processus de découverte enregistre (`save`) les informations sur ce service dans son *serviceRegistry*. Ainsi si la découverte est suivie d'une invocation du service, le nœud devient Proxy Invoquant (PI) pour ce service. Dans le cas contraire, il gardera le statut de Proxy Localisant (PL) pour le service.

L'avantage de l'enregistrement de la connaissance après découverte d'un service est double. En effet, ce mécanisme permet d'une part de décharger les PSI. D'autre part, il améliore les performances du système en permettant une localisation et/ou une invocation ultérieure plus rapide.

Rappelons que dans les deux cas (PI ou PL), pour un service donné, un nœud est proxy invoquant ou localisant pour une durée temporaire qui dépend de la réputation de ce service en question. Donc si le service n'est pas demandé jusqu'à la fin du *timeout* $\mathcal{T}_{Live}$, le nœud retire les informations sur le service de son *serviceRegistry*. Cette approche permet ainsi de garder les indexes des services désirés dans le système et aussi de ne pas saturer inutilement les *serviceRegistry*.

La publication de service le rend localisable par les utilisateurs de la grille. Nous verrons ce mécanisme dans la prochaine section.

4.2.4 Service de découverte : `lookup`

La localisation (ou découverte) est la première étape de la chaîne d'exécution de service. Toutefois, rechercher et localiser un service résolvant un besoin spécifique dans un environnement de grille pair-à-pair où les ressources sont géographiquement dispersées, est un enjeu majeur.

C'est pour répondre à cette problématique que nous avons proposé de construire un arbre couvrant durant la phase de structuration du système P2P4GS. Cet arbre couvrant constitué uniquement des PSI du système sera utilisé pendant le processus de découverte de services. Ainsi, les requêtes de recherche vont être acheminées le long de cet arbre. Ceci permet une recherche exhaustive puisque tous les services partagés dans le système sont indexés au niveau des différents PSI constituant l'arbre couvrant.

Dans ce qui suit, nous décrivons le processus de découverte de services dans P2P4GS.

Approche de découverte de services

Le principe d'exécution d'algorithme de découverte de services se déroule comme suit :

Lorsqu'un utilisateur souhaite rechercher un service au sein du système P2P4GS, il se connecte à un nœud au système qui constituera son *point d'entrée*. Ensuite, il formule sa requête en spécifiant les mots clés décrivant le service recherché.

Le nœud *point d'entrée* exécute alors l'Algorithme 8. Une recherche locale est d'abord effectuée avec l'appel de la fonction `lookup`. Cette fonction retourne sous forme de liste, l'ensemble des services répondant à l'expression du besoin de l'utilisateur. Dans ce cas, le nœud point d'entrée retourne le résultat à l'utilisateur. La distance en terme de nombre de sauts est ainsi égale à 1.

Si par contre aucune entrée de la table *serviceRegistry* du nœud *point d'entrée* ne correspond (*matche*) avec l'expression de la requête, alors la fonction `lookup` retourne -1. Dans ce cas, une recherche globale est alors déclenchée.

Pour initier une recherche globale, le nœud génère une clé de recherche (*keyQuery*) afin d'identifier la requête. En effet, plusieurs requêtes peuvent être simultanément traitées dans le système. Par la suite, en fonction de son statut, le nœud *point d'entrée* envoie la requête de localisation soit à ses voisins PSI (s'il est NS) ou ses PSI passerelles dans l'arbre couvrant (s'il est PSI).

Algorithme 8: À la réception d'une requête de recherche *serviceQuery* depuis le nœud *nodeRequester*

```
1  distance ⟵ 1;
2  if lookup(serviceQuery) ≠ -1 then
       /* Service localement indexé                                    */
3      send(nodeRequester, resultLookup(serviceFindList, distance));
4  else
       /* Génération d'une clé d'identification de la requête          */
       /* Ajout de la clé keyQuery dans la pile de requête             */
5      requestList ⟵ requestList ∪ {(keyQuery, nodeRequester)};
6      set timer T_TimeoutLookup;
7      if statut_entryPoint = NS then
           /* Transmettre la requête aux voisins PSI                   */
8          forall the k ∈ neighTable / statut_k = PSI do
9              send(k, discoveryRequest(keyQuery, serviceQuery, entryPoint, distance));
10         end
11     else if statut_entryPoint = PSI then
           /* Transmettre la requête aux passerelles PSI               */
12         forall the k ∈ gatewayTable do
13             send(k, discoveryRequest(keyQuery, serviceQuery, entryPoint, distance));
14         end
15     end
16 end
```

Remarque 4.2. *Lorsque le nœud point d'entrée n'a pas de PSI dans son voisinage (c'est-à-dire qu'il a toujours son statut UNDEF) alors, le message de recherche est envoyé aux différents nœuds de son voisinage.*

À la réception sur un nœud u d'une requête de découverte de service identifiée par une clé (*keyQuery*), ce dernier exécute l'Algorithme 9. Tout d'abord, le nœud vérifie si la requête identifiée par cette clé est déjà reçue. Si c'est le cas alors, la requête est ignorée. Dans le cas contraire, grâce à sa fonction locale `lookup`, le nœud recherche localement tout service qui correspond avec l'expression de la requête. Ainsi :

- Si le service est indexé, il retourne au nœud point d'entrée de liste des services répondant satisfaisant à l'expression de la requête de recherche.
- Si par contre le service n'est pas indexé localement, alors la requête est retransmise aux voisins PSI passerelles, sauf celui depuis lequel elle a été reçue. La requête est ignorée dans le cas où le nœud n'a pas de voisin PSI passerelle à qui retransmettre la requête.

Algorithme 9: À la réception d'un message discoveryRequest(*keyReq*, *serviceQuery*, *entryPoint*, *distance*) depuis le nœud v

```
1  if ∄ k = (keyQuery_k) ∈ requestList_i / keyQuery_k = keyQuery then
2      requestList_i ⟵ requestList_i ∪ {(keyQuery, nodeRequester)};
3      distance ⟵ distance + 1;
4      if lookup(serviceQuery) ≠ -1 then
           /* Service localement indexé                                  */
5          send(entryPoint, discoveryResponse(keyQuery, ServiceFindList, distance));
6      else
7          if |gatewayTable| > 1 then
               /* Retransmettre requête aux PSI passerelles sauf la source */
8              forall the k ∈ gatewayTable / id_k ≠ id_v do
9                  send(k, discoveryRequest(keyQuery, serviceQuery, entryPoint,
                   distance));
10             end
11         end
12     end
13 end
```

Durant le processus de découverte, les éventuels résultats reçus par le nœud *point d'entrée* sont agrégés (Algorithme 10). À l'expiration du temps de latence $\mathcal{T}_{TimeoutLookup}$, le nœud *point d'entrée* retourne à l'utilisateur :

- « 0 résultat(s) ! » si aucun résultat n'est reçu lors de la recherche ;

Algorithme 10: À la réception d'un message discoveryResponse(*keyQuery*, *serviceFindList*, *distance*) ou à l'expiration du temps de recherche

```
1  if T_Lookup expire then
2  |  if |resultLookup| > 0 then
   |  |  /* Envoyer à l'utilisateur les résultats reçus                    */
3  |  |  forall the i ∈ resultLookup do
4  |  |  |  send(nodeRequester, resultLookup(serviceFindList_i, distance_i));
5  |  |  end
6  |  else
7  |  |  send(nodeRequester, "0 résultat(s) !");
8  |  end
9  else
   |  /* Agrégation des résultats                                          */
10 |  resultLookup ⟵ resultLookup ∪ {(serviceFindList_i, distance_i)};
11 end
```

- la liste des services qui *matchent* avec la requête si la recherche est fructueuse.

Si la recherche est fructueuse, l'utilisateur pourra alors invoquer le service qui répond le mieux à ses besoins. La section qui suit détaille les mécanismes d'invocation et d'exécution de service.

4.2.5 Invocation et exécution de service : `invoke` et `exec`

L'invocation d'un service (`invoke`) consiste à appeler un service distant afin de procéder à son exécution. Étant donné que les services n'ont pas les mêmes contraintes d'exécution, un nœud peut être proxy invoquant pour un service et ne pas l'être pour un autre service. Ainsi, lorsqu'un nœud du système souhaite l'exécution d'un service :

- s'il est proxy invoquant pour ce service, il peut alors procéder à son invocation ;
- dans le cas contraire, il envoie un message *executionRequest* au nœud responsable du service pour demander une exécution en local. Lorsque cette requête d'exécution parvient au nœud responsable du service, ce dernier l'exécute (`exec`). Après exécution, le résultat ainsi produit est retourné (message *executionResult*) au nœud ayant initié la demande l'exécution.

Une demande d'exécution d'un service peut aussi être initiée par un utilisateur (ou nœud) extérieur de la grille. Dans ce cas :

- si le nœud *point d'entrée* est responsable du service demandé, alors il l'exécute et le résultat retourné est mis à la disposition de l'utilisateur ;

- si par contre, le nœud *point d'entrée* n'est pas responsable du service demandé, soit il l'invoque directement s'il est proxy invoquant pour ce service (exemple de la Figure 4.1) ; soit il soumet la requête (message *executionRequest*) au nœud responsable du service pour une exécution en local (exemple de la Figure 4.2). Dans les deux cas, le résultat de l'exécution est retourné au nœud *point d'entrée* et ce dernier le met à la disposition de l'utilisateur.

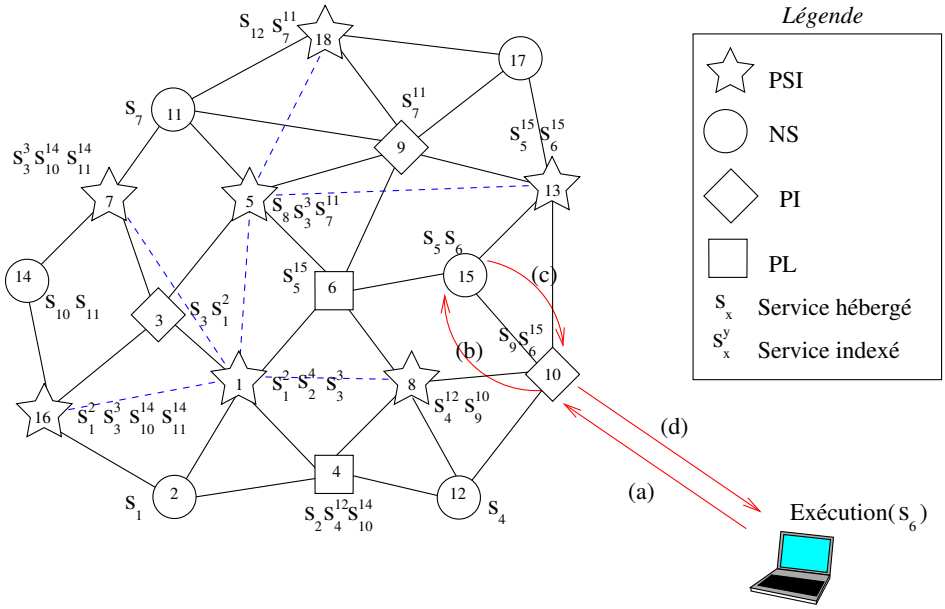


FIGURE 4.1 – Scénario d'exécution du service S_6

Soulignons que l'exécution du service s'effectue de manière transparente pour l'utilisateur. Ce dernier ne connait donc pas l'emplacement du nœud chargé de l'exécution du service sollicité. Il interagit seulement avec son nœud *point d'entrée* pour la soumission de l'exécution d'un service et la récupération des résultats produits.

4.3 Synthèse

Dans ce chapitre nous avons proposé des mécanismes de gestion de services dans un système de grilles de services pair-à-pair. En effet, la gestion des ressources constitue un réél défi dans tout système distribué, en particulier dans un environnement de grille pair-à-pair où les ressources mises en commun sont dispersées géographiquement et souvent de nature dynamiques et très hétérogènes.

Nous avons ainsi décrit dans un premier temps les motivations liées à la gestion de ressources dans un tel environnement. Dans un second temps, nous avons décrit les différents

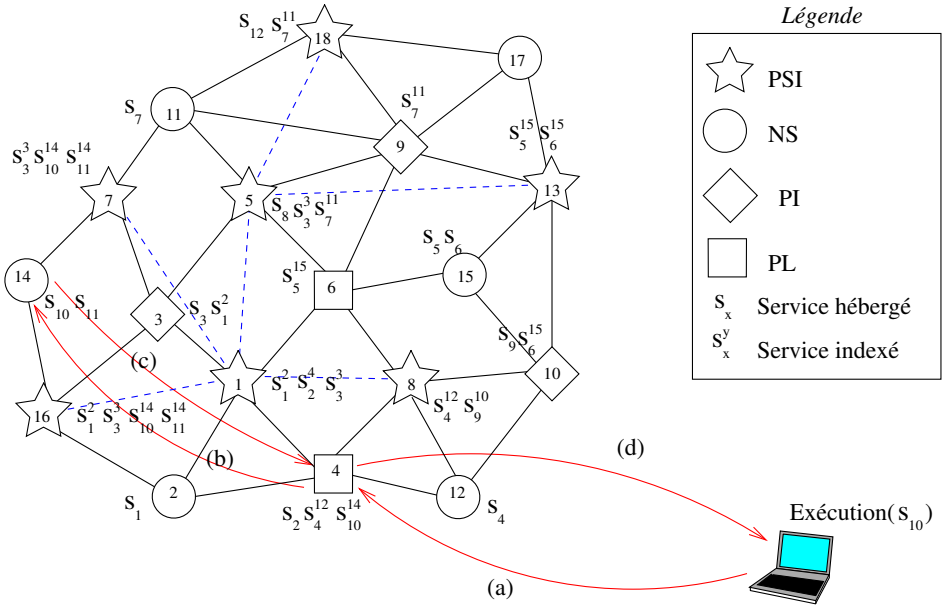


FIGURE 4.2 – Scénario d'exécution du service S_{10}

mécanismes de gestion de services offerts par la plateforme P2P4GS. Ces mécanismes couvrent le cycle de vie d'un service et comprend ainsi la gestion du déploiement, la gestion de l'enregistrement (publication et mémorisation), la gestion de la localisation et la gestion de l'invocation ainsi que de l'exécution.

Dans le prochain chapitre, nous analysons expérimentalement nos différentes solutions proposées.

Chapitre 5

Expérimentation et évaluation de la spécification P2P4GS

Résumé. *Dans ce chapitre, nous présentons une campagne de simulations en vue d'analyser les performances de l'intergiciel P2P4GS.*

Nous commençons d'abord par décrire notre environnement de simulation ainsi que les protocoles implémentés à savoir Gia, Pastry et Kademlia. Ensuite nous définissons les différentes métriques d'évaluation de performances de notre système. Par la suite, nous présentons les résultats obtenus lors de l'évaluation du nombre de PSI formés avec notre première approche de structuration. Ensuite, nous étudions l'impact du degré minimal requis ($\Delta_{RequiredMinDegree}$) sur les performances du système. Pour ce faire, nous déterminons en fonction du degré minimal requis, le pourcentage de PSI formés et le coût des communications en termes de messages. Enfin, nous analysons l'impact des pannes sur la découverte de services.

Les travaux que nous présentons dans ce chapitre sont publiés dans des conférences avec actes et comités de sélection : [GFRN14b, GFRN15, GFRN16b, GFRN16a]

Sommaire

5.1 Environnement de simulation 98
5.2 Description des overlays implémentés 99
5.2.1 Le protocole Gia 100
5.2.2 Le protocole Pastry 101
5.2.3 Le protocole Kademlia 102
5.3 Mesures de performances 104
5.3.1 Métriques de performance et paramètres de simulations 104
5.3.2 Performances de la première approche de structuration 105
5.3.3 Impact du degré minimal requis sur les performances du système 106

5.3.4 Impact des pannes sur la découverte de services 114
5.4 Synthèse 115

5.1 Environnement de simulation

Dans cette section, nous présentons notre environnement de simulation.

D'emblée, une remarque que l'on pourrait soulever est l'absence d'un simulateur pair-à-pair standard. Ce qui n'est pas le cas pour la simulation de réseau bas niveau dont la référence est NS (*Network Simulator*) [HLR+08].

Il existe plusieurs simulateurs de réseaux pair-à-pair disponibles [BFS+13] ; parmi ceux-ci, on peut citer P2PSim [GKL+03], Peersim [MJ09], SimGrid [Cas01], Dasor [BFR08], PlanetSim [GPM+05], Oversim [BHK07].

Nous avons choisit le simulateur Oversim pour plusieurs raisons.

En effet, la simulation d'un réseau dynamique requière un module de "*bootstrapping*" pour la gestion des connexions. Or, la plupart des simulateurs n'intègre pas ce module. Les auteurs de [BFS+13] précisent d'ailleurs que seuls Planetsim et Oversim ont implémenté cette fonctionnalité. D'autre part, Planetsim ne supporte pas le *churn* [BFS+13].

En outre, plusieurs protocoles P2P (structurés comme non structurés) sont implémentés dans OverSim. Ce qui permet d'avoir un large choix sur les scénarios de simulations.

Oversim est développé à l'Institut de la télématique de Karlsruhe en Allemagne. C'est un Framework basé sur le simulateur OMNeT++ [1] qui est *open source*, à événements discrets et écrit en C++.

Ce simulateur est totalement programmable, paramétrable et hautement modulaire. Son fonctionnement repose entièrement sur l'utilisation de modules qui communiquent entre eux par échanges de messages. Ces modules sont organisés hiérarchiquement. Les modules de base sont appelés les *simple modules* et sont regroupés en modules composés appelés *compound modules*. La Figure 5.1 illustre la relation entre *simple modules* et *compound modules*. Notons que les *compound modules* peuvent eux mêmes être regroupés en *compound modules*. Le nombre de niveaux hiérarchiques n'est donc pas limité.

L'architecture est construite de telle sorte que les *simple modules* sont à la fois les émetteurs et destinataires des messages. Les *compound modules* se contentent de relayer les messages aux *simple modules* de façon transparente. On peut attribuer différents paramètres aux connections reliant les modules : des délais de propagation, des débits de données, des taux d'erreur, etc.

1. http ://www.omnetpp.org/

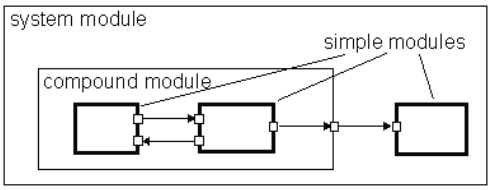


FIGURE 5.1 – Structure des modules du simulateur OMNeT++

Les messages sont transmis par le biais de portes (*gates*) qui sont les interfaces d'émission et de réception des modules.

L'ensemble de la structure du réseau et des interconnexions est décrit à l'aide d'un langage de définition (déclaratif) propre appelé NED (pour *NEtwork Description*). Ainsi, la description de la topologie est stockée dans un fichier d'extension « .ned », qui est essentiellement constitué de la déclaration des *simple modules*, qui spécifient les gates ainsi que les paramètres de connexion ; mais aussi de la définition des *compound modules* (interfaces externes) et du réseau de manière générale. Soulignons que le langage NED est aussi compatible avec le standard XML en permettant, grâce à son API NEDXML, l'importation et l'exportation de fichiers au format XML.

Nos simulations sont effectuées dans Grid'5000 [2]. Nous avons utilisé les nœuds de Saint Rémi du Centre de Calcul de Champagne-Ardenne ROMEO [3] et les nœuds de Sophia [4].

Afin d'atteindre nos objectifs, nous avons implémenté notre solution sur des systèmes pair-à-pair fonctionnant de manière totalement différente, à savoir :

- Gia [CRB+03] qui construit un overlay non structuré ;
- Pastry [RD01] qui construit un overlay structuré en anneau ;
- Kademlia [MM02] qui construit un overlay structuré en hypercube bien que modélisé souvent sous la forme d'un arbre.

Précisons qu'avec ces trois applications choisies, nous faisons le tour des structures existantes.

5.2 Description des overlays implémentés

Dans cette section, nous décrivons les protocoles P2P implémentés dans le cadre de nos travaux. Ces protocoles sont Gia, Pastry et Kademlia.

2. https ://www.grid5000.fr/
3. https ://romeo.univ-reims.fr
4. http ://www-sop.inria.fr/grid5000/

5.2.1 Le protocole Gia

Gia [CRB+03] est un réseau non-structuré fortement basé sur le protocole Gnutella [Sol01] et qui propose plusieurs améliorations de celui-ci.

Dans son fonctionnement, le protocole fournit un algorithme de sélection des voisins qui adapte précisément la connectivité d'un nœud à ses capacités, en définissant pour chaque nœud un nombre maximal de voisins (`MAX_NEIGHBORS`) auquel il peut se connecter. Afin d'accroître les performances du système, quatre mécanismes essentiels, décrits ci-dessous, sont mis en œuvre dans ce protocole.

Gia possède une topologie adaptative qui tire partie de l'hétérogénéité entre les capacités des nœuds présents dans le réseau. En effet, il introduit un paramètre de capacité local à chaque nœud, reflétant sa puissance en termes de bande passante, CPU, quantité de mémoire, etc. et assure que les nœuds à plus haut degré sont ceux à plus haute capacité.

Pour atteindre cet objectif, chaque nœud calcule indépendamment son niveau de satisfaction (noté S) en fonction de sa puissance, son voisinage actuel et de `MAX_NEIGHBORS`, qui est un paramètre de configuration. S est une valeur comprise entre 0 et 1. Une valeur S = 0, indique que le nœud n'est pas encore satisfait, et peut, par conséquent recevoir de nouvelles connexions (nouveaux voisins). Tandis que S = 1, signifie que le nœud est pleinement satisfait. De cette manière, les nœuds ayant de faibles capacités sont rapprochés d'au moins un nœud à forte capacité. Ces nœuds à haute capacité sont ainsi vus comme des *Super-nœuds*. Toutefois notons qu'il n'y a aucune distinction, du point de vue architectural, entre un nœud de faible capacité et un nœud de forte capacité et que, par conséquent la topologie n'est pas à deux niveaux.

En plus de cela, Gia intègre un mécanisme de contrôle de flux actif. Afin d'éviter que les nœuds ne soient surchargés de requêtes, le protocole met en œuvre un mécanisme de jetons. En effet, chaque nœud attribue régulièrement des jetons périmables de contrôle de flux à ses voisins. Ainsi, un nœud ne peut envoyer une requête à son voisin que s'il a reçu préalablement un jeton de celui-ci. Lorsqu'un nœud commence à devenir surchargé de requêtes, il réduit alors la fréquence à laquelle il attribue des jetons à ses voisins.

En outre, les nœuds du réseau échangent périodiquement avec leurs voisins les index des fichiers qu'ils possèdent. Cela permet d'améliorer l'efficacité de la recherche d'information car un nœud qui reçoit une requête peut y répondre, non seulement pour lui, mais aussi pour ses voisins.

Enfin, Gia utilise le mécanisme de marche aléatoire biaisée pour la propagation des requêtes, contrairement à Gnutella qui se base sur l'inondation. Ainsi, au lieu de transmettre les requêtes à tous les voisins (*flooding*) ou certains choisis au hasard (*random walk*), les nœuds dans Gia essayent d'aiguiller les requêtes vers les nœuds ayant le plus fort degré, tout en respectant bien évidemment le mécanisme de contrôle de flux. La pro-

pagation d'une requête est limitée par un nombre de sauts ainsi qu'un nombre de réponses positives au delà duquel la réponse est retournée. Comparé aux protocoles non-structurés, Gia requiert moins de messages lors d'une recherche ce qui le rend compatible avec de très grands réseaux. Cependant, la recherche fournit des résultats moins complets, ce qui est problématique lorsqu'un pair recherche une ressource rare.

5.2.2 Le protocole Pastry

Pastry [RD01] est un réseau de recouvrement (*overlay*) auto-organisé et basé sur une table de hachage distribuée. Son architecture repose sur un modèle pair-à-pair décentralisé et structuré. Il construit une structure virtuelle en anneau et utilise une approche de routage en préfixe basé sur l'algorithme de Plaxton [PRR99].

Chaque nœud dans Pastry est muni d'un identifiant *nodeID* codé sur 128 bits et qui est le résultat d'une fonction de hachage appliquée à son adresse IP ou à sa clé publique. Cet identifiant est utilisé pour positionner le nœud dans un espace circulaire de nommage, qui va de 0 à $2^{128} - 1$. Ainsi, dans un réseau contenant N pairs (nœuds), le protocole est capable d'associer un pair dont l'identifiant est le plus proche numériquement en $\log_{2^b}$N sauts, en moyenne. Où b est un paramètre de configuration typiquement égal à 4.

Chaque nœud u maintient une table de routage contenant trois catégories d'entrées.

Le *leaf set*. Il contient l'ensemble L des voisins virtuels du nœud u considéré en termes de distance numérique de son identifiant ($nodeID_u$). Plus spécifiquement, le *leaf set* supérieur contiendra les L/2 nœuds les plus proches numériquement et supérieurs à l'identifiant du nœud u. Il en sera de même pour le *leaf set* inférieur qui contiendra les L/2 noeuds les plus proches numériquement et ayant un identifiant inférieur. L est un paramètre de configuration typiquement égal à 2^b.

Le *routing table*. Il contient la table de routage elle-même. Chaque entrée de cette table contient un bloc d'adresse qui lui est assigné. Pour former le bloc des adresses, l'identifiant de 128 bits est divisé en digits de b bits chacun, menant à un système en base b. Ainsi, chaque ligne r contient $2^b - 1$ entrées dont les r premiers chiffres sont communs à ceux du pair considéré. Ceci partitionne les adresses en plusieurs niveaux où le niveau 0 représente un préfixe commun de taille nulle entre les deux adresses, le niveau 1 un préfixe commun de taille 1 entre les deux adresses, et ainsi de suite. La table de routage contient alors l'adresse du nœud le plus proche connu pour chaque digit à chaque niveau. En conséquence, une table de routage Pastry contient $(2^b - 1) \times (\log_{2^b}\text{N})$ entrées.

Le *neighbourhood*. Il contient l'ensemble des voisins réels les plus proches du nœud u. La métrique choisie pour évaluer la distance entre pairs est le nombre de sauts IP.

Le Tableau 5.1 présente un exemple simplifié de table de routage Pastry. Dans cet exemple, le paramètre b est fixé à 2 et la longueur des identifiants est de 8 digits.

Nœud d'identifiant 10233102			
Voisins virtuels			
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Table de routage			
-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	
Voisins réels			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

TABLE 5.1 – Table de routage d'un pair d'identifiant 10233102 (extrait de Pastry [RD01])

Pour s'insérer dans l'anneau virtuel, le nouveau nœud contacte grâce au *bootstrap*, un nœud présent dans le réseau. Si ce dernier est responsable de la clé du nouveau nœud, il établit la connexion avec lui. Dans le cas contraire, il route le message vers le nœud dans sa table de routage dont l'identifiant est numériquement le plus proche de la clé.

Lorsqu'un message est routé vers un destinataire, le préfixe commun entre les nœuds intermédiaires et la destination augmente à chaque saut. Chaque nœud intermédiaire envoie au nouveau nœud, l'entrée correspondante au préfixe commun. Une fois que le nouveau nœud s'insère dans l'anneau, il met à jour sa table de routage. Ensuite, il informe les nœuds de sa table de routage de sa présence et ces derniers mettront à jour leur table de routage à leur tour.

Soulignons que le protocole Pastry a servi de réseau de base à de nombreuses applications, dont le système de stockage de données anonyme PAST [DR01], et de l'application de diffusion d'évènements Scribe [CDKR02]. Pastry a été avantagé par le développement de FreePastry [5] qui est une implémentation *open source* du protocole.

5.2.3 Le protocole Kademlia

Kademlia [MM02] est un réseau de recouvrement auto-organisé et basé sur une table de hachage distribuée. Son architecture repose sur un modèle pair-à-pair décentralisé et

5. http ://www.freepastry.org

structuré. Il construit une structure virtuelle en hypercube bien que modélisé souvent sous la forme d'un arbre.

Chaque nœud est muni d'un identifiant *nodeID* codé sur 160 bits et qui est le résultat d'une fonction de hachage appliquée à son adresse IP. Kademlia utilise la métrique OU exclusif (XOR) pour calculer la distance séparant dans le réseau logique deux identifiants (*nodeID*) de nœuds. Donc la distance d entre deux nœuds u et v est définie par la fonction XOR entre ces nœuds : $d(x, y) = x \oplus y$.

Chaque nœud u maintient une table de routage composée des ensembles nommés ***k-buckets***. Un ***bucket*** regroupe k nœuds dont les distances sont comprises entre 2^i et 2^{i+1} (avec $0 \leq i < 160$). Tous les nœuds du même ***k-bucket*** sont à la même distance du nœud u. Les ***k-buckets*** sont classés par ordre d'éloignement selon la métrique XOR. Chaque ***k-bucket*** correspond donc à un sous arbre et contient k voisins classés selon leur ancienneté dans le système.

Notons que les premiers ***k-buckets*** du nœud u contiennent les identifiants des voisins les plus proches selon la métrique XOR. Ces derniers sont nommés ***sibling*** par les auteurs de S/Kademlia [BM07].

Dans son fonctionnement, le protocole Kademlia fournit quatre primitives de types RPC (*Remote procedure call*) :

- `PING` : cette procédure interroge un noeud pour savoir s'il est connecté au réseau.
- `STORE` : cette procédure ordonne à un noeud de stocker un couple <*key*, *value*> dans le but de le récupérer ultérieurement.
- `FIND_NODE` : cette procédure retourne une liste de triplets contenant l'adresse IP, le numéro de port UDP et l'identifiant *nodeID* des nœuds connus pour être les plus proches de l'identifiant ciblé.
- `FIND_VALUE` : cette procédure retourne la donnée stockée par un nœud lors d'une recherche sur son identifiant.

Lorsqu'un nouveau nœud u rejoint le réseau Kademlia, il insère son nœud de *bootstrap* dans son ***k-bucket*** approprié. Ensuite, le nœud u effectue un "*self-lookup*" (une recherche) de son propre identifiant ($nodeID_u$) afin d'alimenter sa table de routage.

La recherche dans ce système se fait de manière itérative. En effet, lorsqu'un nœud reçoit un message de localisation d'une clé ($nodeID_u$), il retourne une liste de q nœuds connus et plus proches (selon la métrique XOR) de la clé. À la réception d'une réponse, le nœud met à jour sa table de routage avec la liste q reçue. À l'issue de cette première étape, les plus proches q contacts reçus sont interrogés et ainsi de suite.

Soulignons que Kademlia est le premier protocole déployé réellement à grande échelle.

Parmi ses premières implémentations on peut citer eDonkey [HB02] et eMule [6] avec son réseau Kad. D'autres implémentations basées sur Kademlia, dont le client BitTorrent [Coh02] de Vuze [7], ont été réalisées par la suite.

5.3 Mesures de performances

Nous analysons les performance de notre solution dans cette section. Pour ce faire, nous définissons dans un premier temps les métriques de performance ainsi que les paramètres de simulations. Ensuite, nous présentons les résultats obtenus lors de l'évaluation du pourcentage de PSI formés avec notre première approche de structuration. Puis, nous étudions l'impact du degré minimal requis ($\Delta_{RequiredMinDegree}$) sur les performances du système. Enfin, nous évaluons l'impact des pannes sur la découverte de services.

5.3.1 Métriques de performance et paramètres de simulations

Afin d'analyser les performances de notre solution, nous considérons les métriques d'évaluations suivantes :

- Pourcentage de PSI : il définit le nombre PSI formés sur le nombre total de nœuds dans le système. En d'autres termes, il correspond au pourcentage de clusters (communautés virtuelles) en fonction de la taille du réseau. Il permet ainsi de définir le dimensionnement du système et a un impact sur le degré de centralisation de l'information.
- Coût des communications : il définit le nombre de messages échangés dans le système pour l'accomplissement d'une tâche bien définie. En fonction du degré minimal requis et de la taille du système, nous déterminons :
  - d'une part, le nombre requis de messages échangés pour la formation de communautés virtuelles (clusters) ;
  - d'autre part, le diamètre de l'arbre couvrant qui représente le nombre maximal de sauts à effectuer lors de la procédure de découverte de services.
- Taux de succès des requêtes : il définit le pourcentage de requêtes réussies en fonction du pourcentage de nœuds en pannes lors d'un processus de découverte de services.

Les paramètres que nous utilisons dans nos différentes simulations sont résumés dans le Tableau 5.2. Soulignons que ce sont les valeurs par défaut proposées dans chaque protocole que nous utilisons dans ces expérimentations.

6. http ://www.emule.com
7. http ://www.vuze.com

	Paramètres	Valeurs
Paramètres Globaux	Nombre de nœuds	[500, 5000]
	$\Delta_{RequiredMinDegree}$	[4, 24]
	$\mathcal{T}_{timeout}$ décision	30 s
	Fréquence messages heartbeat	120 s
	Init Phase Creation Interval	0.1 s
Protocole Gia	Gia Level of Satisfaction	1
	Aggressiveness of Adaptation	256
	Maximum Neighbors	50
Protocole Pastry	Bits Per Digit	4
	Number of Leaves	16
	Proximity Neighbor Selection	On
Protocole Kademlia	Bits Per Digit	1
	Bucket Nodes	16
	Sibling Nodes	8

TABLE 5.2 – Paramètres de simulation pour l'évaluation de la spécification P2P4GS

5.3.2 Performances de la première approche de structuration

Dans cette section, nous faisons une analyse de performances de notre première approche de structuration.

Rappelons que dans la première approche, la structuration se base uniquement sur le voisinage initial des nœuds. En effet, lorsqu'un nœud se connecte dans le système et établit son voisinage initial, il devient PSI s'il n'a pas de voisins PSI et NS dans le cas contraire.

Pour évaluer le pourcentage de PSI, nous considérons des réseaux avec un nombre de nœuds variant entre 500 et 5000.

La Figure 5.2 représente, pour chaque protocole P2P sous-jacent (i.e. Gia, Pastry et Kademlia), le pourcentage de PSI formés en fonction de la taille du système.

Comme nous pouvons le constater, cette approche de structuration passe à l'échelle puisque le pourcentage de PSI formés reste relativement constant lorsque la taille du réseau augmente.

Toutefois, le pourcentage de PSI (et donc de *clusters*) est supérieur à 20% pour les différents protocoles P2P. Pour améliorer cette approche, nous avons introduit le critère de degré minimum requis afin de mieux contrôler la distribution des clusters.

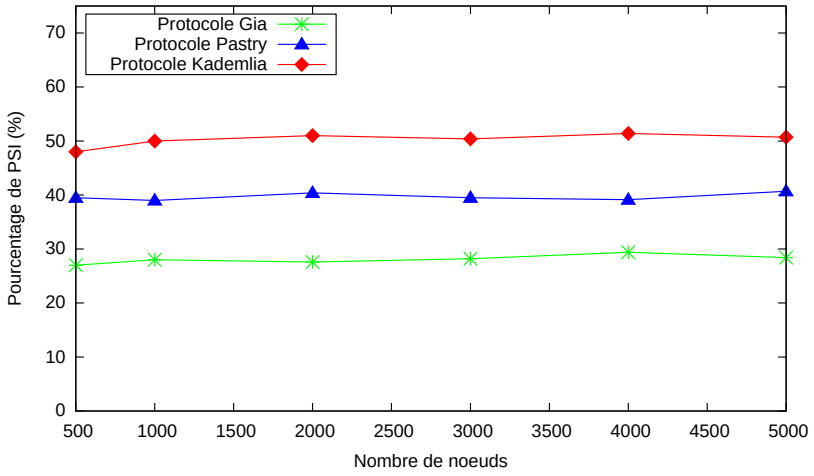


FIGURE 5.2 – Pourcentage de PSI en fonction du protocole P2P et de la taille du réseau

5.3.3 Impact du degré minimal requis sur les performances du système

Dans cette section, nous étudions l'impact du degré minimal requis ($\Delta_{RequiredMinDegree}$) sur les performances du système.

Nous allons dans un premier temps évaluer le pourcentage de PSI formés en fonction de la taille du système. Dans un second temps, nous évaluons le coût des communications en termes de nombre de messages. Pour ce faire, nous mesurons d'une part, le nombre de messages requis pour la formation des groupes virtuels. D'autre part, nous déterminons le diamètre de l'arbre couvrant qui représente le nombre maximal de sauts à effectuer lors de la procédure de découverte de services.

Pourcentage de PSI en fonction du degré minimal requis

Nous avons introduit le critère de degré minimal requis ($\Delta_{RequiredMinDegree}$) dans le but de mieux contrôler la distribution des groupes virtuels. En effet, la contrainte sur le degré minimal requis permet d'une part, d'éviter la création de clusters singletons (c'est-à-dire contenant qu'un seul nœud et dans ce cas le PSI) ou encore de clusters contenant un nombre insignifiant (très petit) de nœuds. Cela signifie que $\Delta_{RequiredMinDegree}$ ne doit pas prendre une valeur trop petite.

D'autre part, cette contrainte permet d'éviter la création d'un faible nombre de PSI. En effet, plus le nombre de PSI est petit, plus les PSI ont un grand nombre de voisins et donc plus les clusters seront denses. Ce qui impliquera une plus forte centralisation de l'information pouvant ainsi provoquer une surcharge de PSI et favoriser des goulots

d'étranglement dans le réseau. En outre, les risques d'indisponibilité en cas de panne augmentent. Cela signifie que $\Delta_{RequiredMinDegree}$ ne doit pas prendre une valeur trop grande.

Ainsi, comme le précisent des travaux dans la littérature [AVX+12], nous préconisons un pourcentage de clusters compris entre 5% et 20% pour un meilleur dimensionnement du système.

Pour évaluer le pourcentage de PSI formés en fonction du degré minimal requis, nous considérons des réseaux avec un nombre de nœuds variant entre 500 et 5000. Pour chaque taille de réseau, nous faisons varier le paramètre $\Delta_{RequiredMinDegree}$ entre 4 et 24.

Les figures 5.3, 5.4 and 5.5 représentent le pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau, suivant respectivement le protocole P2P Gia, Pastry et Kademlia.

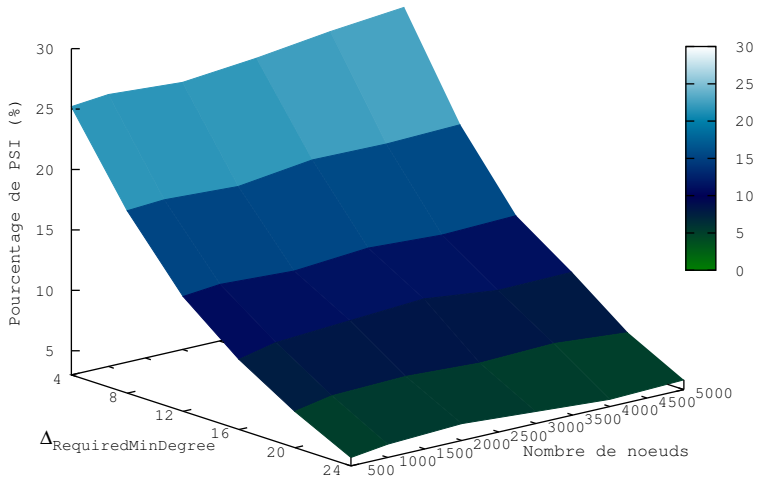


FIGURE 5.3 – Protocole Gia : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

Nous constatons d'une part, que pour chaque protocole P2P sous-jacent et pour chaque valeur de $\Delta_{RequiredMinDegree}$, le pourcentage de PSI formés reste relativement constant lorsque la taille du réseau augmente. Ce qui confirme le passage à l'échelle de notre solution.

D'autre part, comme nous pouvions l'imaginer, les résultats représentés dans les figures 5.3, 5.4 et 5.5 montrent que le pourcentage de PSI formés diminue, lorsque le degré minimal requis augmente. Ainsi, en fonction du protocole P2P sous-jacent, nous donnons dans ce

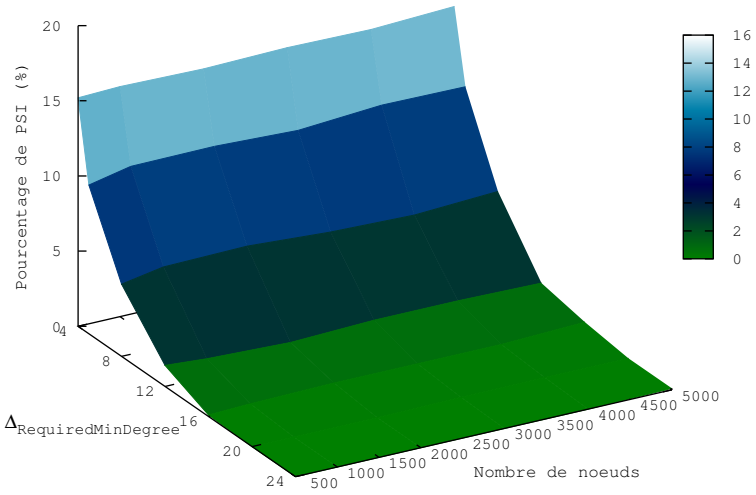


FIGURE 5.4 – Protocole Pastry : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

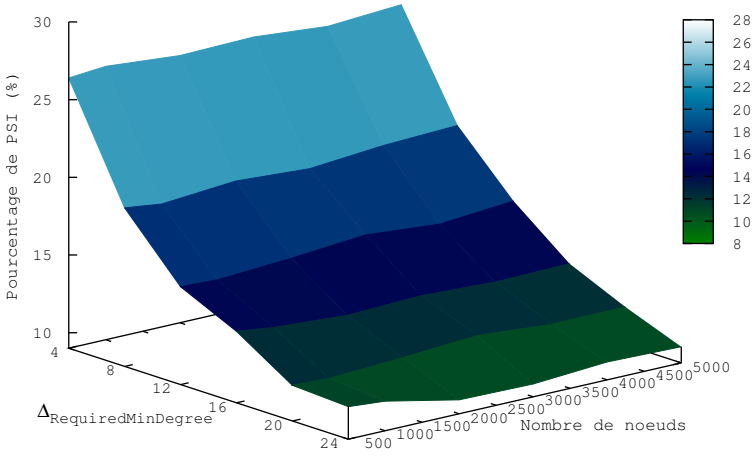


FIGURE 5.5 – Protocole Kademlia : Pourcentage de PSI formés en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

qui suit, les valeurs de $\Delta_{RequiredMinDegree}$ qui fournissent un meilleur dimensionnement du système, c'est-à-dire des pourcentages de PSI compris entre 5% et 20%.

– Cas du protocole Gia. La Figure 5.3 montre que le pourcentage de PSI formés varie entre $3,6\%$ et $27,1\%$. Donc, les valeurs du paramètre $\Delta_{RequiredMinDegree}$ qui fournissent une meilleure distribution de clusters sont comprises entre 8 et 20.

– Cas du protocole Pastry. Dans la figure 5.4, nous observons que le pourcentage de PSI formés varie entre 0 and $15,33\%$. Ainsi, les valeurs du paramètre $\Delta_{RequiredMinDegree}$ qui fournissent une meilleure distribution de clusters sont comprises entre 4 et 8.

– Cas du protocole Kademlia. La Figure 5.5 montre que le pourcentage de PSI formés varie entre $9,83\%$ et $26,6\%$. De ce fait, les valeurs du paramètre $\Delta_{RequiredMinDegree}$ qui offrent une meilleure distribution de clusters sont comprises entre 8 et 24 ;

D'une manière générale, nous constatons que le protocole Pastry fournit un meilleur dimensionnement du système avec des valeurs de $\Delta_{RequiredMinDegree}$ relativement faibles. Contrairement à ce protocole, Kademlia construit des réseaux denses. C'est pour cela que les valeurs de $\Delta_{RequiredMinDegree}$ fournissant un meilleur dimensionnement sont relativement élevées. On remarque enfin que le protocole Gia fournit de meilleures performances en termes de distribution de clusters.

En fait, le protocole Gia assure que les nœuds à plus haut degré sont les nœuds à plus haute capacité en termes de CPU, bande passante, mémoire, etc.. De plus le protocole intègre un mécanisme de contrôle de flux pour éviter la surcharge des nœuds. En outre, chaque nœud calcule indépendamment son niveau de satisfaction (S). Ainsi, dans la mesure où un nœud est pas entièrement satisfait, l'adaptation de la topologie va continuer à rechercher des voisins appropriés pour améliorer le niveau de satisfaction.

Diamètre de l'arbre couvrant en fonction du degré minimal requis

Nous étudions dans cette section l'impact de $\Delta_{RequiredMinDegree}$ sur le diamètre de l'arbre couvrant qui représente le nombre maximal de sauts à effectuer lors de la procédure de découverte de services. En effet, ce diamètre correspond à la valeur en terme de nombres de sauts dans le pire des cas c'est-à-dire, lorsque la ressource recherchée se trouve à l'extrémité la plus éloignée du point d'entrée. Rappelons que l'arbre couvrant est construit lors de la phase de structuration du réseau et est constitué uniquement des PSI ainsi formés.

Nous considérons ainsi des réseaux avec un nombre de nœuds variant entre 500 et 5000 et pour chaque taille de réseau, nous faisons varier le paramètre $\Delta_{RequiredMinDegree}$ entre 4 et 24, par intervalles de 4.

Les figures 5.6, 5.7 and 5.8 représentent le diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau, suivant respectivement le protocole P2P Gia, Pastry et Kademlia.

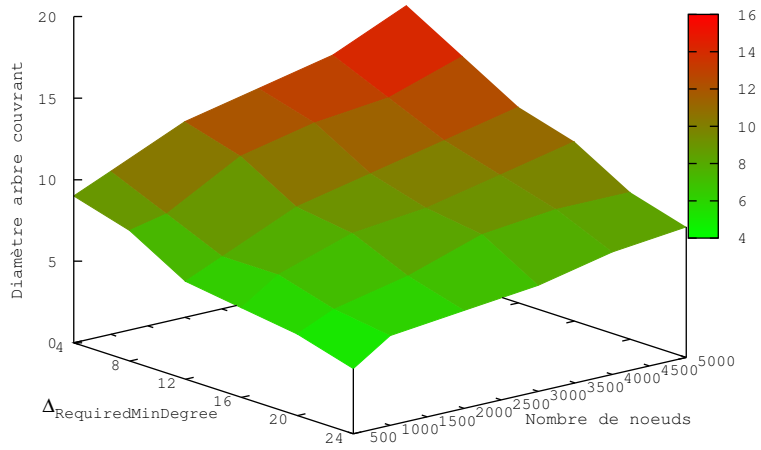


FIGURE 5.6 – Protocole Gia : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

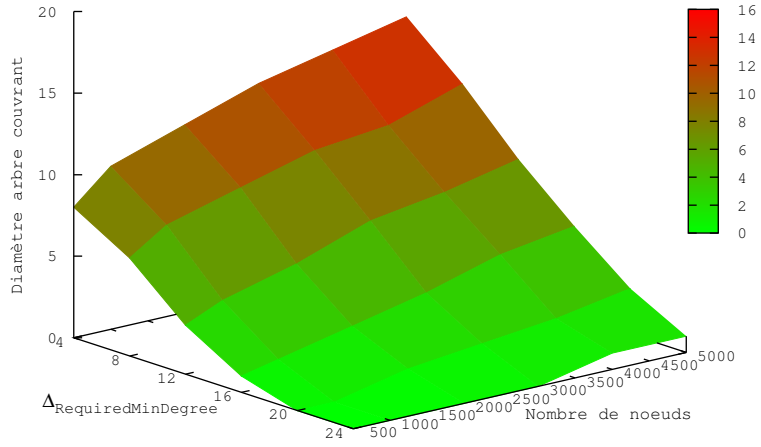


FIGURE 5.7 – Protocole Pastry : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

Nous observons d'une part, qu'au niveau des différents protocoles P2P sous-jacents et pour chaque valeur du degré minimal requis, le diamètre de l'arbre couvrant croît de manière logarithmique lorsque la taille du réseau augmente. Ce qui confirme encore une fois le passage à l'échelle de notre solution.

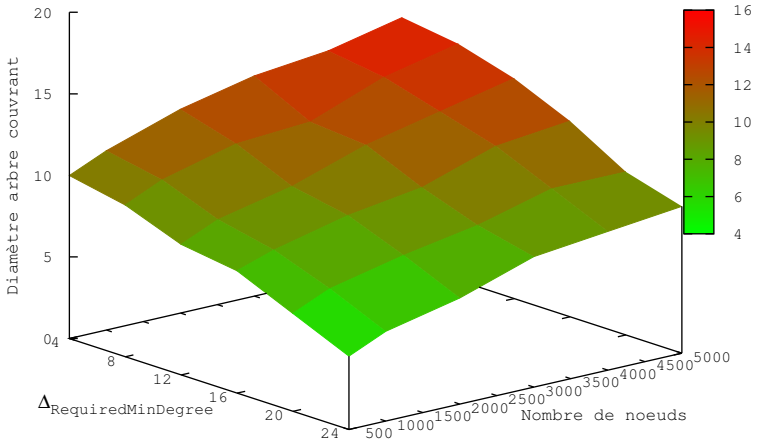


FIGURE 5.8 – Protocole Kademlia : Diamètre de l'arbre couvrant en fonction du degré minimal requis ($\Delta_{RequiredMinDegree}$) et de la taille du réseau

D'autre part, les résultats présentés dans les figures 5.6, 5.7 et 5.8 montrent que le diamètre de l'arbre couvrant diminue, lorsque le degré minimal requis augmente. Ce qui était aussi prévisible. Nous remarquons que pour chaque taille de réseau, le degré de diminution varie d'un protocole à un autre. En effet, dans le cas du protocole Gia, le diamètre de l'arbre couvrant diminue proportionnellement avec l'augmentation du degré minimal requis. Cette diminution du diamètre de l'arbre est relativement faible si nous considérons le protocole Kademlia. Ce qui atteste du fort degré de connexité des nœuds d'un réseau opérant avec ce protocole. Par contre, le diamètre de l'arbre couvrant diminue très considérablement avec l'augmentation du degré minimal requis dans le cas du protocole Pastry.

Coût des communications en termes de messages

Afin de mieux analyser et évaluer le coût des communications en termes de messages, nous introduisons le degré minimal requis idéal ($\Delta_{RequiredMinDegree}$ idéal) que nous définissons comme suit :

Définition 5.1. *Soit un protocole P2P quelconque donné, le degré minimal requis idéal est définit comme étant la valeur du $\Delta_{RequiredMinDegree}$ qui fournit un pourcentage de PSI plus proche de $k = 12,5\%$; c'est-à-dire, la médiane entre la limite inférieure et celle supérieure du pourcentage optimal de cluster.*

Ainsi, en fonction du protocole P2P sous-jacent, les résultats des simulations précédemment illustrés (les figures 5.3, 5.4 and 5.5) donnent les valeurs suivantes :

- Cas du protocole Gia : $\Delta_{RequiredMinDegree}$ idéal $= 12$;
- Cas du protocole Pastry : $\Delta_{RequiredMinDegree}$ idéal $= 5$;
- Cas du protocole Kademlia : $\Delta_{RequiredMinDegree}$ idéal $= 18$.

a) Coût en termes de messages de structuration

Pour évaluer le coût des communications en termes de messages échangés pour la formation de communautés virtuelles , nous considérons des réseaux avec un nombre de nœuds variant entre 500 et 5000.

La Figure 5.9 représente le nombre total de messages échangés pour la formation de communautés virtuelles, en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau.

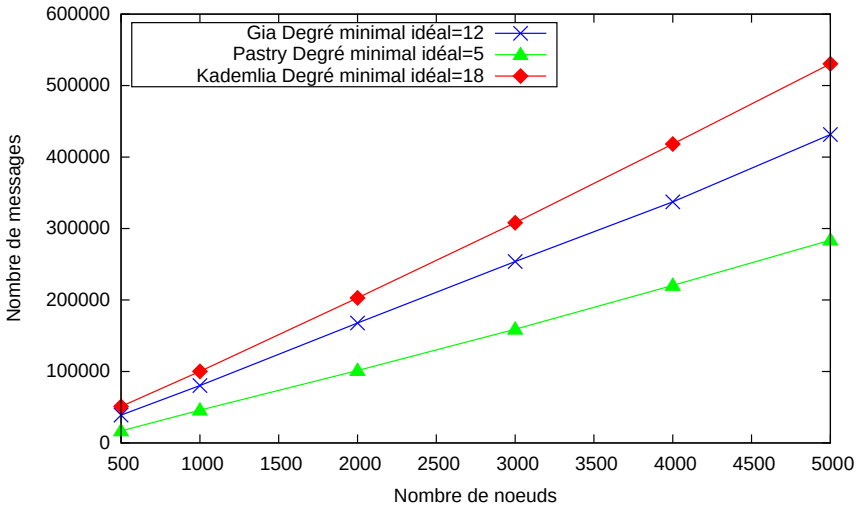


FIGURE 5.9 – Nombre total de messages échangés en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau

Nous observons que pour chaque protocole P2P sous-jacent avec son $\Delta_{RequiredMinDegree}$ idéal, le nombre total de messages échangés augmente linéairement avec le nombre de nœuds dans le système. Ainsi, comme nous pouvons le prévoir, l'augmentation de la taille du réseau engendre plus de communications.

La particularité de cette approche est que la construction de l'arbre couvrant se fait au moment même de la structuration du système. Ce qui permet de minimiser le coût des communications en termes de messages. En effet, les messages `clusterManagement`

encapsulent aussi bien les informations sur les services partagés que celles sur les voisins PSI de l'émetteur. Ainsi, pour chaque nouveau lien créé entre deux PSI, dans le cadre de la construction de l'arbre couvrant, un seul message supplémentaire de notification (`masterRouteManagement`) est produit dans le système.

D'autre part, nous observons que pour la formation de communautés virtuelles, le protocole Pastry génère moins de messages que les autres protocoles. C'est le protocole Kademlia qui génère le plus de messages. Le degré minimal requis idéal a un impact sur ces résultats obtenus. En effet, comme le montre la Figure 5.9, plus le $\Delta_{RequiredMinDegree}$ idéal est élevé, plus le nombre de messages générés est élevé.

b) Coût en termes de messages de découverte de services

Notre approche de découverte de services se base sur le parcours de l'arbre couvrant constitué des PSI du système de grille pair-à-pair. Ainsi, pour évaluer le coût de recherche, nous déterminons le diamètre de l'arbre couvrant, qui représente le nombre maximal de sauts à effectuer lors d'une procédure de découverte, pour atteindre la ressource demandée.

La figure représente le diamètre de l'arbre couvrant, en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau.

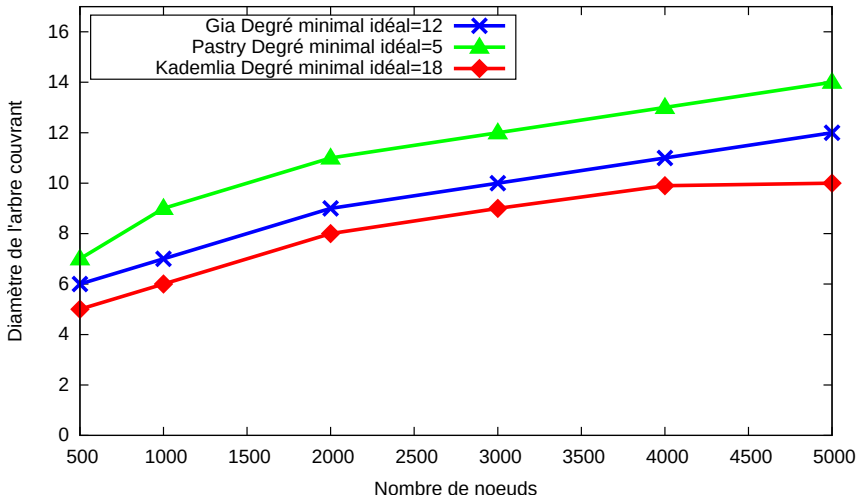


FIGURE 5.10 – Diamètre de l'arbre couvrant, en fonction du $\Delta_{RequiredMinDegree}$ idéal du protocole P2P sous-jacent et de la taille du réseau

Ce diamètre correspond donc à la valeur en terme de nombres de sauts dans le pire des cas ; c'est-à-dire, lorsque la ressource recherchée se trouve à l'extrémité la plus éloignée du point d'entrée.

Les expériences illustrées dans la figure montrent que pour chaque protocole P2P sous-jacent avec son $\Delta_{RequiredMinDegree}$ idéal, le diamètre de l'arbre couvrant croît de façon logarithmique avec le nombre de nœuds dans le réseau. Cela confirme donc le passage à l'échelle de notre mécanisme de découverte de service.

D'autre part, nous observons que pour un processus de découverte services à travers la communauté de grille de pair-à-pair, le protocole Kademlia offre de meilleures performances en termes de nombre de sauts.

5.3.4 Impact des pannes sur la découverte de services

En l'absence de pannes dans le système, la recherche de services fournit des résultats exhaustifs grâce au maintien d'un arbre couvrant de PSI. Dans cette section, nous étudions l'influence de la dynamique du système sur la découverte de services. Pour ce faire, nous évaluons le taux de succès (ou taux de réussite) des requêtes de recherche de services en présence de pannes d'un certain nombre de nœuds dans le système.

Nous réalisons des simulations en faisant varier jusqu'à 30% le nombre de nœuds en panne sur des réseaux d'une taille de 1000 nœuds. La Figure 5.11 représente le taux de succès des requêtes de recherche de service en fonction du pourcentage de nœuds en panne et du protocole P2P sous-jacent. Pour chaque protocole, les expérimentations sont faites avec son $\Delta_{RequiredMinDegree}$ idéal.

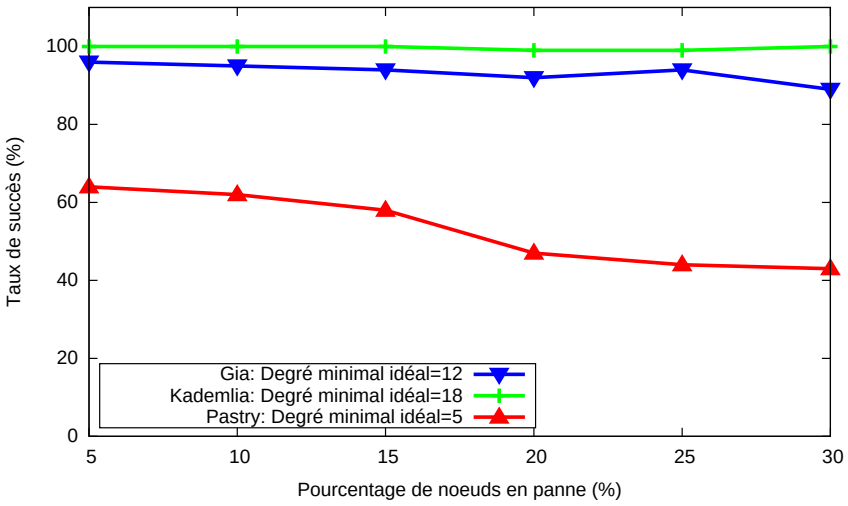


FIGURE 5.11 – Taux de succès d'une recherche de service en fonction du pourcentage de nœuds en panne

Nous remarquons d'une manière générale que le comportement du système P2PGS face aux pannes est tributaire du protocole P2P utilisé. En effet, nous constatons d'une

part que le *churn* affecte peu les performances de P2P4GS basé sur le protocole Gia. Comme l'illustre la Figure 5.11, pour les différentes valeurs de pourcentage de nœuds en panne, le taux de réussite varie entre 89% et 96%.

D'autre part, nous observons que c'est le protocole Kademlia qui résiste mieux au *churn*. En effet, le taux de réussite des requêtes tourne entre 99% et 100% pour les différentes valeurs de pourcentage de nœuds en panne. Ceci s'explique par le fait que les nœuds d'un réseau Kademlia présentent une très grande densité de connexion. La connaissance est par conséquence fortement répliquée.

Enfin, nous remarquons que contrairement aux protocoles Gia et Kademlia, le protocole Pastry offre un taux de réussite beaucoup plus faible. En effet, la Figure 5.11 montre qu'en fonction du pourcentage de nœuds en panne, le taux de succès varie entre 43% et 64%. Les résultats des expérimentations attestent que ce protocole résiste moins au *churn*. En outre, nous observons que plus le pourcentage de nœuds en panne augmente, plus le taux de succès diminue. Ce qui atteste du faible degré de connexité des nœuds d'un réseau opérant avec le protocole Pastry.

5.4 Synthèse

Dans ce chapitre, nous avons évalué les performances de notre spécification P2P4GS par le biais d'une campagne de simulations. Nous avons tout d'abord décrit notre environnement de simulation ainsi que les protocoles implémentés à savoir Gia, Pastry et Kademlia. Par la suite, nous avons défini les différentes métriques d'analyse de performances de notre système et présenté les résultats de simulations.

Les tests de performances ont montré que notre solution assure un passage à l'échelle en termes de dimensionnement du réseau et aussi de coût des communications. En outre, nous avons d'une part remarqué que parmi les protocoles implémentés, Gia fournit de meilleures performances en termes de distribution de clusters. D'autre part, nous avons observé que le protocole Pastry génère moins de messages que les autres protocoles. Tandis que, le protocole Kademlia offre de meilleures performances en termes de nombre de sauts lors d'un processus de découverte de services. Enfin, nous avons constaté que le P2P4GS fournit une meilleure résistance aux pannes lorsqu'il exploite Gia ou Kademlia en tant que protocole P2P sous-jacent.

D'une manière générale, nous pouvons conclure que parmi les protocoles P2P implémentés dans P2P4GS, Gia semble être le plus adapté, vu les performances qu'il offre dans les différentes expérimentations que nous avons effectué.

Conclusion et perspectives

Conclusion

Dans cette thèse, nous nous sommes intéressés à la gestion dynamique de services dans un environnement de grille pair-à-pair à large échelle. À cet effet, nous avons proposé un modèle qui, en plus d'exploiter les propriétés inhérentes et fondamentales qu'offrent les systèmes pair-à-pair, apportent des solutions aux limites des modèles classiques.

Ce modèle appelé P2P4GS (*Peer-To-Peer For Grid Services*) présente l'originalité de ne pas lier l'infrastructure pair-à-pair à la plate-forme d'exécution de services. La couche de gestion de la grille pair-à-pair est en fait séparée de la couche de localisation et d'invocation de services. Nous avons ainsi proposé un modèle d'architecture constitué de quatre couches d'abstraction. Ces couches superposées mettent en évidence les différents mécanismes sous-jacents à l'environnement de grille pair-à-pair, ainsi que les interactions entre les différentes entités du système.

De plus, l'intergiciel P2P4GS est générique, c'est-à-dire qu'il est applicable sur toute architecture pair-à-pair. Pour garantir cette propriété, vu que les systèmes distribués à large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous avons proposé de structurer le système de grille pair-à-pair en communautés virtuelles aussi appelées *clusters*. Cette structuration présente deux caractéristiques intéressantes à savoir la limitation des communications et le passage à l'échelle.

L'approche de structuration ainsi proposée est, d'une part, complètement distribuée et se base uniquement sur le voisinage des nœuds pour l'élection des PSI (*Proxy Système d'Information*) responsables clusters. En vue de ne pas surcharger les PSI, nous avons proposé de répartir certaines tâches sur d'autres types de nœuds distingués. En effet, lorsqu'un processus de découverte d'un service aboutit, le nœud ayant déclenché ce processus devient temporairement, en fonction de ses capacités (CPU, RAM, etc.), PI (*Proxy Invoquant*) ou PL (*Proxy Localisant*) pour ce service. De cette manière, les nœuds proxys vont assurer collectivement la gestion des ressources partagées.

D'autre part, dans le but de bien orchestrer les communications au sein des différentes communautés virtuelles et aussi permettre une recherche efficace et exhaustive de service dans le système, un arbre couvrant constitué uniquement des PSI est maintenu. En conséquence, les requêtes de recherche vont être acheminées le long de cet arbre couvrant. La particularité de la solution est que le processus de structuration du système P2P4GS et la construction de l'arbre couvrant se font simultanément. Ce qui permet de minimiser le coût des communications en termes de messages de gestion du réseau *overlay*.

Outre le mécanisme de découverte de services, nous avons proposé des mécanismes de déploiement, de publication et d'invocation de services.

Enfin, pour analyser les performances du système P2P4GS, nous avons effectué une campagne de simulations sous OverSim. Afin d'illustrer la propriété de généricité du modèle, nous l'avons implémenté sur des protocoles P2P opérant de manière totalement différentes, à savoir Gia (qui construit un *overlay* non-structuré), Pastry (qui construit un *overlay* structuré en anneau) et Kademlia (qui construit un *overlay* structuré en hypercube bien que souvent modélisé en arbre).

Les résultats de simulations ont montré, d'une part, que notre solution garantit un passage à l'échelle en termes de dimensionnement du réseau et aussi de coût communications. Nous avons étudié l'impact du degré minimal requis ($\Delta_{RequiredMinDegree}$) sur les performances de système afin de déterminer, pour chaque protocole P2P implémenté, les valeurs de $\Delta_{RequiredMinDegree}$ qui offrent un meilleur dimensionnement du système. Nous avons aussi déterminé pour chaque protocole P2P, sa valeur de $\Delta_{RequiredMinDegree}$ idéale ; à partir de cette valeur, nous avons mesuré le diamètre de l'arbre couvrant du système correspondant. Ce diamètre représente le nombre maximal de sauts à effectuer lors de la procédure de découverte de services. D'autre part, nous avons étudié l'impact des pannes sur les performances du système. Nous avons conclu que parmi les trois protocoles P2P implémentés, l'intergiciel P2P4GS fournit une meilleure résistance aux pannes lorsqu'il exploite Gia ou Kademlia en tant que protocole P2P sous-jacent.

Perspectives

Les principales perspectives que nous envisageons visent d'une part à étendre les fonctionnalités du système P2P4GS et d'autre part à améliorer ses performances.

Robustesse et qualité de service

Une manière d'assurer la disponibilité d'un service, c'est de le répliquer. La réplication d'un service est une technique qui consiste à dupliquer le service sur plusieurs nœuds.

Nous envisageons ainsi de mettre en œuvre une stratégie de réplication de service en se basant sur sa réputation. L'idée est d'associer un facteur de réplication à chaque service. Ce facteur sera incrémenté après chaque nouvelle invocation du service. Lorsque le facteur atteint une certaine valeur k (paramètre de configuration), on réplique le service correspondant sur un de ces nœuds Proxy Invoquant (PI). On pourra opter comme critère de choix le PI ayant effectué le plus grand nombre d'invocations. Le critère discriminatoire pourra aussi être le PI le plus proche en termes de TTL.

D'autre part, les résultats de simulations ont montré que le système P2P4GS résiste bien aux pannes de nœuds. En ce qui concerne les PSI, l'arbre couvrant est maintenu de manière dynamique. En effet, les mécanismes de tolérance aux pannes proposés dans le chapitre 3 permettent l'élection d'autres candidats en cas de pannes de PSI. Cependant, les PSI qui maintiennent l'arbre n'ont pas une vision globale de ce dernier. Des mécanismes de gestion plus avancés de l'arbre couvrant sont nécessaires en cas de pannes simultanées d'un très grand nombre de PSI.

Vers l'unification de systèmes de grilles pair-à-pair

Outre les caractéristiques qu'offre le système P2P4GS, nous comptons exploiter une autre propriété qu'elle pourra garantir : l'unification de plusieurs systèmes de grilles pair-à-pair isolés. En effet, vu la structure modulaire de son architecture, sa propriété de généricité et son couplage faible avec le protocole P2P sous-jacent, pour former le réseau de recouvrement, il suffira d'adapter la spécification en ajoutant quelques modules ou fonctionnalités comme :

- un mécanisme de nommage qui est propre à la spécification en vue d'assurer l'unicité des identifiants des nœuds issus des différents réseaux ;
- un mécanisme de maintien de liens virtuels entre des nœuds se trouvant dans des réseaux pair-à-pair différents.

L'objectif de cette unification est d'étendre les sphères de collaboration et de partage de ressources, et aussi donner la possibilité de faire fédérer plusieurs réseaux fonctionnant sous des protocoles différents.

Implementation et application

Nous envisageons en outre de développer un prototype du modèle P2P4GS. Pour ce faire, nous nous baserons sur une implémentation pair-à-pair *open source* (comme FreePastry [8] par exemple) pour développer ce prototype. L'objectif à court terme est de

8. http ://www.freepastry.org

définir l'ensemble des services d'infrastructure d'une plate-forme de grille pair-à-pair pour l'enseignement à distance.

Bibliographie

[AA+12] Hesham Ali, Moataz Ahmed *et al.* : Hprdg : A scalable framework hypercube-p2p-based for resource discovery in computational grid. *In Computer Theory and Applications (ICCTA), 2012 22nd International Conference on*, pages 2–8. IEEE, 2012.

[ACGC05] Nazareno Andrade, Lauro Costa, Guilherme Germóglio et Walfredo Cirne : Peer-to-peer grid computing with the OurGrid community. *In Proceedings of the SBRC*, pages 1–8, 2005.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno et Vijay Machiraju : *Web services*. Springer, 2004.

[AFdSY+04] E Schaeffer Alberto Filho, Luciano C da Silva, Adenauer C Yamin, Iara Augustin, Lincoln L De Morais et Cláudio Fr Geyer : Perdis : A scalable resource discovery service for the isam pervasive environment. *In Peer-to-Peer Systems, 2004. International Workshop on Hot Topics in*, pages 80–85. IEEE, 2004.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica *et al.* : A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AG94] Ankh Arora et Mohamed Gouda : Distributed reset. *Computers, IEEE Transactions on*, 43(9):1026–1038, 1994.

[AH01] Karl Aberer et Manfred Hauswirth : Peer-to-peer information systems : concepts and models, state-of-the-art, and future systems. *In ACM SIGSOFT Software Engineering Notes*, volume 26, pages 326–327. ACM, 2001.

[ALR04] Algirdas Avižienis, Jean-Claude Laprie et Brian Randell : Dependability and its threats : a taxonomy. *In Building the Information Society*, pages 91–120. Springer, 2004.

[ATS04] Stephanos Androutsellis-Theotokis et Diomidis Spinellis : A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[AVX+12] Navid Amini, Alireza Vahdatpour, Wenyao Xu, Mario Gerla et Majid Sarrafzadeh : Cluster size optimization in sensor networks with decentralized cluster-based protocols. *Computer communications*, 35(2):207–220, 2012.

[Bas99] Stefano Basagni : Distributed clustering for ad hoc networks. *In Parallel Architectures, Algorithms, and Networks, 1999.(I-SPAN'99) Proceedings. Fourth InternationalSymposium on*, pages 310–315. IEEE, 1999.

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab *et al.* : Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[BFR08] Alain Bui, Olivier Flauzac et Cyril Rabat : Dasor, a grid model based simulation library. *In I2CS ?08, 8th International Conference on Innovative Internet Community Systems*, 2008.

[BFS+13] Anirban Basu, Simon Fleming, James Stanier, Stephen Naicken, Ian Wakeman et Vijay K. Gurbani : The state of peer-to-peer network simulators. *ACM Comput. Surv.*, 45(4):46 :1–46 :25, August 2013.

[BG03] B Beverly et Hector Garcia : Designing a super-peer network. *In Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 49–60. IEEE, 2003.

[BHK+06] Rüdiger Berlich, Marcus Hardt, Marcel Kunze, Malcolm Atkinson et David Fergusson : Egee : building a pan-european grid training organisation. *In Proceedings of the 2006 Australasian workshops on Grid computing and e-research-Volume 54*, pages 105–111. Australian Computer Society, Inc., 2006.

[BHK07] Ingmar Baumgart, Bernhard Heep et Stephan Krause : OverSim : A flexible overlay network simulation framework. *In IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[BJ15] HMN Bandara et Anura P Jayasumana : P2p-based, multi-attribute resource discovery under real-world resources and queries. *ACM Transactions on Internet Technology (TOIT)*, 15(1):5, 2015.

[BM07] Ingmar Baumgart et Sebastian Mies : S/kademlia : A practicable approach towards secure key-based routing. *In Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–8. IEEE, 2007.

[BMH10] Amos Brocco, Apostolos Malatras et Béat Hirsbrunner : Enabling efficient information discovery in a self-structured grid. *Future Generation Computer Systems*, 26(6):838–846, 2010.

[BPSM+98] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler et François Yergeau : Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210*, 16, 1998.

[BS06] Salman A Baset et Henning Schulzrinne : An analysis of the skype peer-to-peer internet telephony protocol. 2006.

[BT85] Gabriel Bracha et Sam Toueg : Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[BWH06] Edward Benson, Glenn Wasson et Marty Humphrey : Evaluation of uddi as a provider of resource discovery services for ogsa-based grids. *In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 9–pp. IEEE, 2006.

[Cas01] Henri Casanova : Simgrid : A toolkit for the simulation of application scheduling. *In Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2001.

[Cat02] C Catlett : The teragrid : A primer, 2002.

[CCMN04] Girish B Chafle, Sunil Chandra, Vijay Mann et Mangala Gowri Nanda : Decentralized orchestration of composite web services. *In Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143. ACM, 2004.

[CD06] Eddy Caron et Frédéric Desprez : DIET : A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi et Sanjiva Weerawarana : Unraveling the web services web : an introduction to soap, wsdl, and uddi. *IEEE Internet computing*, 6(2):86, 2002.

[CDK05] George F Coulouris, Jean Dollimore et Tim Kindberg : *Distributed systems : concepts and design*. pearson education, 2005.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec et Antony IT Rowstron : SCRIBE : A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[CDT06] Eddy Caron, Frédéric Desprez et Cédric Tedeschi : Dynamic prefix tree for service discovery within large scale grids. *In Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on*, pages 106–116. IEEE, 2006.

[CDT08] Eddy Caron, Frédéric Desprez et C Tedesch : Efficiency of tree-structured peer-to-peer service discovery systems. *In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[CFF+04] Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke et William Vambenepe : The ws-resource framework version 1.0. *Initial draft release from*, 3(05), 2004.

[CFFK01] Karl Czajkowski, Steven Fitzgerald, Ian Foster et Carl Kesselman : Grid information services for distributed resource sharing. *In High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 181–194. IEEE, 2001.

[CFK+98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith et Steven Tuecke : A resource management architecture for metacomputing systems. *In Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998.

[Cha82] Ernest J. H. Chang : Echo algorithms : Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, (4):391–401, 1982.

[CHvR+04] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers *et al.* : Universal description, discovery and integration (uddi) version 3.0.2. *OASIS UDDI Specification Technical Committee*, 2004.

[CMG05] Adeep S Cheema, Moosa Muhammad et Indranil Gupta : Peer-to-peer discovery of computational resources for grid applications. *In Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 7–pp. IEEE, 2005.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman et Sanjiva Weerawarana : Web services description language (wsdl) version 2.0 : Core language. *W3C recommendation*, 26:19, 2007.

[Coh02] Bram Cohen : Bittorrent protocol specification v1.0. *WWW, June*, 2002.

[Coh03] Bram Cohen : Incentives build robustness in bittorrent. *In Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham et Scott Shenker : Making gnutella-like p2p systems scalable. *In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM, 2003.

[CT91] Tushar Deepak Chandra et Sam Toueg : Unreliable failure detectors for asynchronous systems (preliminary version). *In Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 325–340. ACM, 1991.

[CT96] Tushar Deepak Chandra et Sam Toueg : Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[CTT05] Carmela Comito, Domenico Talia et Paolo Trunfio : Grid services : principles, implementations and use. *International Journal of Web and Grid Services*, 1(1):48–68, 2005.

[Dan03] Jérôme Daniel : *Services Web : Concepts, techniques et outils*. Vuibert informatique, 2003.

[Dav09] Jeff Davis : *Open source SOA*. Manning Publications Co., 2009.

[DFvG83] Edsger W. Dijkstra, W.H.J. Feijen et A.J.M. van Gasteren : Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[Dij65] Edsger W. Dijkstra : Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, 1965.

[Dij74] Edsger W. Dijkstra : Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, novembre 1974.

[DMMRS15] Pasquale De Meo, Fabrizio Messina, Domenico Rosaci et Giuseppe ML Sarné : An agent-oriented, trust-aware approach to improve the qos in dynamic grid federations. *Concurrency and Computation : Practice and Experience*, 27(17):5411–5435, 2015.

[DR01] Peter Druschel et Antony Rowstron : Past : A large-scale, persistent peer-to-peer storage utility. *In Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 75–80. IEEE, 2001.

[DS80] Edsger W Dijkstra et Carel S. Scholten : Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[Epp99] David Eppstein : Spanning trees and spanners. *Handbook of computational geometry*, pages 425–461, 1999.

[Erl08] Thomas Erl : *Soa : principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[Fas03] Fasttrack accessed on-line. 2003.

[FBA+03] Luis Ferreira, Viktors Berstis, Jonathan Armstrong, Mike Kendzierski, Andreas Neukoetter, Masanobu Takagi, Richard Bing-Wo, Adeeb Amir, Ryo Murakawa, Olegario Hernandez *et al.* : *Introduction to grid computing with globus*. IBM Corporation, International Technical Support Organization, 2003.

[FCT15] Maurice Djibril Faye, Eddy Caron et Ousmane Thiare : Autonomic management using self-stabilization for hierarchical and distributed middleware. *In Computer and Information Technology ; Ubiquitous Computing and Communications ; Dependable, Autonomic and Secure Computing ; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, pages 2043–2048. IEEE, 2015.

[FFK+97] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor Von Laszewski, Warren Smith et Steven Tuecke : A directory service for configuring high-performance distributed computations. *In High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, pages 365–375. IEEE, 1997.

[FI03] Ian Foster et Adriana Iamnitchi : On death, taxes, and the convergence of peer-to-peer and grid computing. *In Peer-to-Peer Systems II*, pages 118–128. Springer, 2003.

[Fie00] Roy Thomas Fielding : *Architectural styles and the design of network-based software architectures*. Thèse de doctorat, University of California, Irvine, 2000.

[FK99] Ian Foster et Carl Kesselman, éditeurs. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[FKNT02a] Ian Foster, Carl Kesselman, Jeffrey M Nick et Steven Tuecke : Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[FKNT02b] Ian Foster, Carl Kesselman, Jeffrey M Nick et Steven Tuecke : The physiology of the grid : An open grid services architecture for distributed systems integration. 2002.

[FKS+06] Ian Foster, Hiro Kishimoto, Andreas Savva, D Berry, A Djaoui, A Grimshaw, B Horn, F Maciel, F Siebenlist, R Subramaniam *et al.* : The open grid services architecture (OGSA), version 1.5. OGF Specification GFD-I. 080, July 2006.

[FKT01] Ian Foster, Carl Kesselman et Steven Tuecke : The anatomy of the grid : Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[FLP85] Michael J Fischer, Nancy A Lynch et Michael S Paterson : Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[Fos05] Ian Foster : Globus toolkit version 4 : Software for service-oriented systems. *In Network and parallel computing*, pages 2–13. Springer, 2005.

[FTL+02] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster et Steven Tuecke : Condor-g : A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[G+04] W3C Web Services Architecture Working Group *et al.* : Web services architecture requirements. *W3C Working Draft*, February 2004.

[Gär03] Felix C Gärtner : A survey of self-stabilizing spanning-tree construction algorithms. Rapport technique, 2003.

[GD06] Saikat Guha et Neil Daswani : An experimental study of the skype peer-to-peer voip system. In IPTPS'06 : The 5th International Workshop on Peer-to-Peer Systems, 2006.

[GFN12a] Bassirou Gueye, Olivier Flauzac et Ibrahima Niang : Services pour les grilles pair-à-pair. *In 11th African Conference on Research in Computer Science and Applied Mathematics (CARI'12)*, pages 127–134, 2012.

[GFN12b] Bassirou Gueye, Olivier Flauzac et Ibrahima Niang : Services pour les grilles pair-à-pair. *In Actes du 4e Colloque National sur la Recherche en Informatique et ses Applications (CNRIA'12)*, Bambey - Thies, Sénégal, 2012.

[GFNR14] Bassirou Gueye, Olivier Flauzac, Ibrahima Niang et Cyril Rabat : P2P4GS : une spécification de grille pair-à-pair de services auto-gérés. *Journal of ARIMA*, 17:155–175, 2014.

[GFRN13] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : Gestion des services dans P2P4GS. *In Actes du 5e Colloque National sur la Recherche en Informatique et ses Applications (CNRIA'13)*, Ziguinchor, Sénégal, 2013.

[GFRN14a] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : P2P4GS : A Specification for Services management in Peer-to-Peer Grids. *In The Fourth International Conference on Advanced Communications and Computation (INFOCOMP'14)*, pages 41–46. IARIA XPS, Paris, 2014.

[GFRN14b] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : A self–adaptive structuring for P2P-based Grids. *In 14th IEEE International Conference on Innovations for Community Services (I4CS'14)*, pages 121–128, Reims, 2014.

[GFRN15] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : Structuration auto-adaptative des grilles pair-à-pair à large échelle. *In Actes du 6e Colloque National sur la Recherche en Informatique et ses Applications (CNRIA'15)*, Thies - EPT, Sénégal, 2015.

[GFRN16a] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : A self-adaptive structuring for large-scale p2p grid environment : Design and experimental analysis. *International Journal Grid and Utility Computing*, 7(2):To appear, 2016.

[GFRN16b] Bassirou Gueye, Olivier Flauzac, Cyril Rabat et Ibrahima Niang : Structuration auto-adaptative d'un système de grilles pair-à-pair à large échelle. *Journal of ARIMA*, 25:To appear, 2016.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar et Yves Lafon : Simple object access protocol (soap) 1.2. *World Wide Web Consortium*, 2003.

[GKL+03] Thomer Gil, Frans Kaashoek, Jinyang Li, Robert Morris et Jeremy Stribling : p2psim, a simulator for peer-to-peer protocols, 2003.

[GNG+11] Bassirou Gueye, Ibrahima Niang, Bamba Gueye, M.O Deye et Y Slimani : Constraints-based response time for efficient QoS in web services composition. *In 7th International Conference on Next Generation Web Services Practices (NWeSP)*, pages 141–146. IEEE, 2011.

[GNGD13] Bassirou Gueye, Ibrahima Niang, Bamba Gueye et Mohamed Ould Deye : QoS4WSC : A framework for web services composition based on

QoS constraints. *International Journal of Computer Information Systems and Industrial Management Applications*, 5(ISSN 2150-7988):488–498, 2013.

[Gou95] Mohamed Gouda : The triumph and tribulation of system stabilization. *Distributed Algorithms*, pages 1–18, 1995.

[GPM+05] Pedro García, Carles Pairot, Rubén Mondéjar, Jordi Pujol, Helio Tejedor et Robert Rallo : *Planetsim : A new overlay network simulation framework*. Springer, 2005.

[HB02] Oliver Heckmann et Axel Bock : The edonkey 2000 protocol. *Rapport technique, Multimedia Communications Lab, Darmstadt University of Technology*, 13, 2002.

[HB04] Hugo Haas et Allen Brown : Web services glossary. *W3C Working Group Note (11 February 2004)*, 9, 2004.

[HCE10] Abdelkader Hameurlain, Deniz Cokuslu et Kayhan Erciyes : Resource discovery in grid systems : a survey. *International Journal of Metadata, Semantics and Ontologies*, 5(3):251–263, 2010.

[HLR+08] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell et JB Kopena : Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14, 2008.

[HM15] Mohammad Hasanzadeh et Mohammad Reza Meybodi : Distributed optimization grid resource discovery. *The Journal of Supercomputing*, 71(1): 87–120, 2015.

[HML09] Eduardo Huedo, Rubén S Montero et Ignacio M Llorente : A recursive architecture for hierarchical grid resource management. *Future Generation Computer Systems*, 25(4):401–405, 2009.

[HN15] Wang Huan et Hideroni Nakazato : Failure detection in p2p-grid system. *IEICE TRANSACTIONS on Information and Systems*, 98(12):2123–2131, 2015.

[IF01] Adriana Iamnitchi et Ian Foster : On fully decentralized resource discovery in grid environments. *In Grid Computing ?GRID 2001*, pages 51–62. Springer, 2001.

[IF04] Adriana Iamnitchi et Ian Foster : A peer-to-peer approach to resource location in grid environments. *In Grid resource management*, pages 413–429. Springer, 2004.

[IFN02] Adriana Iamnitchi, Ian Foster et D Nurmi : A peer-to-peer approach to resource discovery in grid environments. *In IEEE High Performance Distributed Computing*, 2002.

[JA14] C Jeyabharathi et Pethalakshmi Annamalai : New approaches with chord in efficient p2p grid resource discovery. *CoRR*, 4(arXiv :1401.2008):1, 2014.

[JEB05] Guillaume Jourjon et Didier El-Baz : Some solutions for peer-to-peer global computing. *In 13th International Conference on Parallel, Distributed and Network based Processing*, pages 49–58, 2005.

[JM08] Emmanuel Jeanvoine et Christine Morin : Rw-ogs : an optimized randomwalk protocol for resource discovery in large scale dynamic grids. *In Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 168–175. IEEE, 2008.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer *et al.* : Oceanstore : An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[KKRG10] Ram Mohan Rao Kovvur, Vijayakumar Kadappa, S Ramachandram et A Govardhan : Adaptive resource discovery models and resource selection in grids. *In parallel distributed and grid computing (PDGC), 2010 1st international conference on*, pages 95–100. IEEE, 2010.

[KL12] Taskin Kocak et Daniel Lacks : Design and analysis of a distributed grid resource discovery protocol. *Cluster Computing*, 15(1):37–52, 2012.

[KM02] Tor Klingberg et Raphael Manfredi : The gnutella protocol specification v0.6. *Technical specification of the Protocol*, 2002.

[KNS14] Jik-Soo Kim, Beomseok Nam et Alan Sussman : Scalable and effective peer-to-peer desktop grid system. *Cluster Computing*, 17(4):1185–1201, 2014.

[Kru56] Joseph B Kruskal : On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[KS07] Damandeep Kaur et Jyotsna Sengupta : Resource discovery in web-services based grids. *World Academy of Science, Engineering and Technology*, 31:284–288, 2007.

[LA12] Peter A Lee et Thomas Anderson : *Fault tolerance : principles and practice*, volume 3. Springer Science & Business Media, 2012.

[Lam78] Leslie Lamport : Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lap88] JC Laprie : Surete de fonctionnement et tolerance aux fautes : concepts de base. *Rapport technique LAAS-88287, Laboratoire d'automatique et d'analyse des systemes (Toulouse)*, 1988.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li et Scott Shenker : Search and replication in unstructured peer-to-peer networks. *In Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM, 2002.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ritu Sharma et Sharon Lim : A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005.

[LDPC10] Andrei Lavinia, Ciprian Dobre, Florin Pop et Valentin Cristea : A failure detection system for large scale distributed systems. *In Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 482–489. IEEE, 2010.

[LJDW01] Simon St Laurent, Joe Johnston, Edd Dumbill et Dave Winer : *Programming web services with XML-RPC*. O'Reilly Media, Inc., 2001.

[LL77] Gérard Le Lann : Distributed systems-towards a formal approach. *In IFIP Congress*, volume 7, pages 155–160. Toronto, 1977.

[LMH+09] Steven Lynden, Arijit Mukherjee, Alastair C Hume, Alvaro AA Fernandes, Norman W Paton, Rizos Sakellariou et Paul Watson : The design and implementation of ogsa-dqp : A service-based distributed query processor. *Future Generation Computer Systems*, 25(3):224–236, 2009.

[LSG+95] The LHC Study Group *et al.* : The large hadron collider, conceptual design. Rapport technique, CERN/AC/95-05 (LHC) Geneva, 1995.

[Lyn96] Nancy A Lynch : *Distributed algorithms*. Morgan Kaufmann, 1996.

[MB08] Eric A Marks et Michael Bell : *Service Oriented Architecture (SOA) : a planning and implementation guide for business and technology*. John Wiley & Sons, 2008.

[MBM12] Thomas Mager, Ernst Biersack et Pietro Michiardi : A measurement study of the wuala on-line storage service. *In Peer-to-Peer Computing (P2P), 12th International Conference on*, pages 237–248. IEEE, 2012.

[MG11] Peter Mell et Tim Grance : The NIST definition of cloud computing. 2011.

[MJ09] Alberto Montresor et Márk Jelasity : Peersim : A scalable p2p simulator. *In Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[MKL+02] Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins et Zhichen Xu : Peer-to-peer computing, 2002.

[MLM+06] C Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, Rebekah Metz et Booz Allen Hamilton : Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.

[MM02] Petar Maymounkov et David Mazieres : Kademlia : A peer-to-peer information system based on the xor metric. *In Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[MMO07] Moreno Marzolla, Matteo Mordacchini et Salvatore Orlando : Peer-to-peer systems for discovering resources in a dynamic grid. *Parallel Computing*, 33(4):339–358, 2007.

[MSC+12] Tinghuai Ma, Sunyuan Shi, Hao Cao, Wei Tian et Jin Wang : Review on grid resource discovery : Models and strategies. *IETE Technical Review*, 29(3):213–222, 2012.

[MTV05] Carlo Mastroianni, Domenico Talia et Oreste Verta : A super-peer model for building resource discovery services in grids : Design and simulation analysis. *In Advances in Grid Computing-EGC 2005*, pages 132–143. Springer, 2005.

[MW87] Shlomo Moran et Yaron Wolfstahl : Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3): 145–151, 1987.

[NRNH14] Nima Jafari Navimipour, Amir Masoud Rahmani, Ahmad Habibizad Navin et Mehdi Hosseinzadeh : Resource discovery mechanisms in grid systems : A survey. *Journal of Network and Computer Applications*, 41:389–410, 2014.

[Ora01] Andrew Oram : *Peer-to-peer : harnessing the benefits of a disruptive technology*. O'Reilly Media, Inc., 2001.

[PMB+05] Diego Puppin, Stefano Moncelli, Ranieri Baraglia, Nicola Tonellotto et Fabrizio Silvestri : A grid information service based on peer-to-peer. *In Euro-Par 2005 Parallel Processing*, pages 454–464. Springer, 2005.

[PRR99] C Greg Plaxton, Rajmohan Rajaraman et Andrea W Richa : Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[RA81] Glenn Ricart et Ashok K. Agrawala : An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[Ray85] Michel Raynal : *Algorithmes distribués et protocoles*. Editions Eyrolles, Paris, 1985.

[Ray91] Michel Raynal : A simple taxonomy for distributed mutual exclusion algorithms. *ACM SIGOPS Operating Systems Review*, 25(2):47–50, 1991.

[RD01] Antony Rowstron et Peter Druschel : Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *In Middleware 2001*, pages 329–350. Springer, 2001.

[RD10] Rodrigo Rodrigues et Peter Druschel : Peer-to-peer systems. *Communications of the ACM*, 53(10):72–82, 2010.

[RDM06] Tania Gomes Ramos et Alba Cristina Magalhaes Alves De Melo : An extensible resource discovery mechanism for grid computing environments. *In Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 115–122. IEEE, 2006.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp et Scott Shenker : *A scalable content-addressable network*, volume 31. ACM, 2001.

[RJTB08] O Abu Rahmeh, P Johnson et A Taleb-Bendiab : A dynamic biased random sampling scheme for scalable and reliable grid networks. *INFOCOMP Journal of Computer Science*, 7(4):1–10, 2008.

[Sch01] Rüdiger Schollmeier : A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. *In Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102, 2001.

[Seg83] Adrian Segall : Distributed network protocols. *Information Theory, IEEE Transactions on*, 29(1):23–35, 1983.

[SGG01] Stefan Saroiu, P Krishna Gummadi et Steven D Gribble : Measurement study of peer-to-peer file sharing systems. *In Electronic Imaging 2002*, pages 156–170. International Society for Optics and Photonics, 2001.

[SM04] John Sucec et Ivan Marsic : Hierarchical routing overhead in mobile ad hoc networks. *Mobile Computing, IEEE Transactions on*, 3(1):46–56, 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek et Hari Balakrishnan : Chord : A scalable peer-to-peer lookup service for internet

applications. *ACM SIGCOMM Computer Communication Review*, 31(4): 149–160, 2001.

[SNM+02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee et Henri Casanova : Overview of GridRPC : A remote procedure call api for grid computing. *In Grid Computing GRID 2002*, pages 274–278. Springer, 2002.

[Sol01] CDS Solutions : Gnutella protocol specification v0. 4. *http//www.clip2.com/GnutellaProtocol04.pdf*, 2001.

[SR01] Anurag Singla et Christopher Rohrs : Ultrapeers : Another step towards gnutella scalability, December 18, 2001.

[SYAD05] Keith Seymour, Asim YarKhan, Sudesh Agrawal et Jack Dongarra : Netsolve : Grid enabling scientific computing environments. *Advances in Parallel Computing*, 14:33–51, 2005.

[TCF+02] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman et Peter Vanderbilt : Grid service specification – draft 11/4/02. ogsi working group, global grid forum, 2002.

[TCF+03] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman, T Maquire, Thomas Sandholm, David Snelling et Peter Vanderbilt : Open grid services infrastructure (OGSI), version 1.0. June 2003.

[Tel94] Gerard Tel : *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[THCC11] C Tedeschi, Haiwu He, F Chuffart et E Caron : Implementation and evaluation of a p2p service discovery system : Application in a dynamic large scale computing infrastructure. *In 2011 IEEE 11th International Conference on Computer and Information Technology*, pages 41–46. IEEE, 2011.

[TLL12] Yi-Hong Tan, Kevin Lü et Ya-Ping Lin : Organisation and management of shared documents in super-peer networks based semantic hierarchical cluster trees. *Peer-to-Peer Networking and Applications*, 5(3):292–308, 2012.

[TNS+03] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura et Satoshi Matsuoka : Ninf-G : A reference implementation of RPC–based programming middleware for grid computing. *Journal of Grid computing*, 1(1):41–51, 2003.

[Tor12] Javad Akbari Torkestani : A distributed resource discovery algorithm for p2p grids. *Journal of Network and Computer Applications*, 35(6):2028 – 2036, 2012.

[TTP+07] Paolo Trunfio, Domenico Talia, Harris Papadakis, Paraskevi Fragopoulou, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov et Seif Haridi : Peer-to-peer resource discovery in grids : Models and systems. *Future Generation Computer Systems*, 23(7):864–878, 2007.

[TTZ07] Domenico Talia, Paolo Trunfio et Jingdi Zeng : Peer-to-peer models for resource discovery in large-scale grids : a scalable architecture. *In High Performance Computing for Computational Science-VECPAR 2006*, pages 66–78. Springer, 2007.

[Vin97] Steve Vinoski : CORBA : Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[WA99] Barry Wilkinson et Michael Allen : *Parallel programming*, volume 999. Prentice hall New Jersey, 1999.

[ZHS14] Zhongping Zhang, Long He et Shanshan Shi : Grid resource discovery algorithm of the multi-layer overlay network model based on distance. *Journal of Software*, 9(10):2658–2664, 2014.

SERVICES AUTO-ADAPTATIFS POUR LES GRILLES PAIR-A-PAIR

La gestion de ressources distribuées à l'échelle planétaire dans plusieurs organisations virtuelles implique de nombreux défis. Dans cette thèse, nous proposons un modèle pour la gestion dynamique de services dans un environnement de grille pair-a-pair a large échelle. Ce modèle, nommé P2P4GS, présente l'originalité de ne pas lier l'infrastructure pair-a-pair a la plate-forme d'exécution de services. De plus, il est générique, c'est-à-dire applicable sur toute architecture pair-a-pair. Pour garantir cette propriété, vu que les systèmes distribues a large échelle ont tendance à évoluer en termes de ressources, d'entités et d'utilisateurs, nous avons proposé de structurer le système de grille pair-a-pair en communautés virtuelles (clusters).
L'approche de structuration est complétement distribuée et se base uniquement sur le voisinage des nœuds pour l'élection des responsables de clusters appelés PSI (Proxy Système d'Information).
D'autre part, afin de bien orchestrer les communications au sein des différentes communautés virtuelles et aussi permettre une recherche efficace et exhaustive de service, un arbre couvrant constitue uniquement des PSI est maintenu. Les requêtes de recherche vont ainsi être acheminées le long de cet arbre. Outre la découverte de services, nous avons proposé des mécanismes de déploiement, de publication et d'invocation de services. Enfin, nous avons implémente et analyse les performances de P2P4GS. Afin d'illustrer sa généricité, nous l'avons implémente sur Gia, Pastry et Kademlia des protocoles P2P opérant de manière totalement différentes. Les tests de performances ont montrés que le P2P4GS fournit une bonne résistance aux pannes et garantit un passage à l'échelle en termes de dimensionnement du réseau et également de coût de communications.

Systèmes pair-a-pair, Grilles de services, gestion de ressources, modélisation, structuration, tolérance aux pannes.

SELF-ADAPTIVE SERVICES FOR P2P GRID

Resource management management worldwide distributed in several virtual organizations is a key issue. In this thesis, we propose a model for dynamic services management in large-scale peer-to-peer Grid environments. This model named P2P4GS, presents originality not to link peer-to-peer infrastructure to the execution services platform. In addition, the middleware is generic and can be applied on any peer-to-peer architecture. Meanwhile, the increasing size of resources and users in large-scale distributed systems has lead to a scalability problem. To ensure scalability, we propose to organize the peer-to-peer Grid nodes in virtual communities so called clusters. The structuring approach is completely distributed, and only requires local knowledge about nodes neighborhood for election of cluster managers called ISP (Information System Proxy). On the other hand, in order orchestrate communications in the various virtual communities and also enable an efficient service discovery, during structuring process, a spanning tree only constituted of ISP is maintained. Therefore, search queries will be routed along the spanning tree.
Besides the service discovery, we proposed service deployment, publication and invocation mechanisms. Finally, we implemented and analyzed the performance of P2P4GS. To illustrate its genericity, we implemented it on protocols which operate in fully different way. These protocols are Gia, Pastry and Kademlia. Performance tests show that, on the one hand, the P2P4GS provides good fault tolerance and ensures the scalability in terms of the clusters distribution and communication cost.

Peer-to-peer Systems, Grid services, resources management, modelisation, clustering, Fault tolerance.

Discipline : INFORMATIQUE

Université Cheikh Anta Diop de Dakar

Laboratoire LID.

Dakar, Sénégal

Université de Reims Champagne-Ardenne

EA 3804 CReSTIC - UFR Sciences Exactes et Naturelles

Moulin de la Housse – 51687 REIMS CEDEX 2