# Vers un desktop-grid pair-à-pair

Vincent Leroy, Marin Bertier, Anne-Marie Kermarrec

**RenPar'18 / SympA'2008 / CFSE'6**
**Fribourg, Suisse, du 11 au 13 février 2008**

# Vers un desktop-grid pair-à-pair

Vincent Leroy[1], Marin Bertier[1], Anne-Marie Kermarrec[2]
[1]IRISA / INSA Rennes [2]IRISA / INRIA Rennes – Bretagne Atlantique

Institut de Recherche en Informatique et Systèmes Aléatoires,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex - France
{Vincent.Leroy,Marin.Bertier,Anne-Marie.Kermarrec}@irisa.fr

**Résumé**
Les *desktop-grid* permettent d'utiliser la puissance des ordinateurs du grand public dans le cadre de projets scientifiques. L'étude des architectures actuelles montre l'existence de points de centralisation qui sont des freins à la tolérance aux fautes et au passage à l'échelle. Nous proposons donc une architecture qui, en s'appuyant sur la construction de trois overlay pair-à-pair, permet de concevoir un *desktop-grid* de façon complètement décentralisée.

**Mots-clés :** desktop-grid, pair-à-pair, grille de calcul, gossip, overlay

## 1. Introduction

Les grilles de calcul déployées sur Internet connaissent un succès grandissant après du public et permettent de disposer à moindre coût de beaucoup de puissance de calcul. Ces applications de *volunteer computing*, *desktop computing*, ou *global computing*, qui ont pris leur essor avec le projet Seti@home[2] en 1999, restent pour l'instant réservées à de grands projets scientifiques.
Les systèmes actuels qui permettent de déployer des *desktop-grid*, aussi appelés réseaux de *peer-to-peer computing*, présentent souvent des points de centralisation sous la forme de serveurs auxquels les autres machines se connectent. Or, le modèle pair-à-pair repose justement sur une répartition équitable de la charge entre toutes les machines du réseau.
Nous proposons une architecture de *desktop-grid* complètement décentralisée, fondée sur l'utilisation conjointe de trois overlay pair-à-pair. Pour offrir des performances intéressantes dans le cadre de tâches utilisant plusieurs machines, nous étudions un algorithme de formation de groupes optimisés suivant un critère de regroupement. Un système de routage parmi ces groupes est également présenté. Nous mettons en évidence les problèmes de stabilité qu'entraîne la superposition d'overlay, et nous proposons quelques améliorations de stabilisation.
L'article est organisé comme suit : la section 2 présente des systèmes de *desktop-grid* existants et met en évidence leurs éléments de centralisation. Puis, dans la section 3 nous présentons nos objectifs et l'architecture globale de notre système. La section 4 décrit rapidement le fonctionnement des deux premiers overlay qui composent notre architecture, le troisième, intégrant la formation de groupes, est détaillé en section 5. Une évaluation de notre système est présentée en section 6 et nous concluons en section 7.

## 2. Architectures existantes

Les solutions existantes pour le calcul distribué sur Internet utilisent différentes stratégies pour exécuter efficacement les programmes. Les 3 logiciels présentés ici sont représentatifs des organisations de réseaux que l'on peut trouver.

### 2.1. Modèle centralisé : BOINC

BOINC[1] est une application qui permet au grand public de mettre les ressources de son ordinateur à disposition d'un projet scientifique. Un client choisit un programme auquel il veut participer en s'inscrivant sur le site du projet et décide quelle quantité de ressources il lui alloue. Le serveur du projet lui

transmet alors des tâches à exécuter. Chaque tâche est exécutée par plusieurs clients et les résultats sont comparés afin de vérifier leur validité.
BOINC s'articule autour d'une architecture clients-serveurs, dans laquelle chaque client offre ses ressources de calcul à plusieurs serveurs qui lui distribuent des tâches. De part son fonctionnement, BOINC est réservé à des projets de grande ampleur qui s'inscrivent dans la durée et peuvent mobiliser les utilisateurs. Les serveurs de ces projets sont des éléments critiques sans lesquels les calculs ne progressent plus.

### 2.2. Modèle partiellement distribué : XtremWeb

XtremWeb[4] a été conçu pour offrir une architecture plus distribuée que celle de BOINC. Ce système de gestion de grille de calcul est également capable d'utiliser les ressources d'ordinateurs individuels pour en tirer profit, mais à la différence de BOINC, il a une approche beaucoup plus ouverte de la soumission de tâches. Les personnes mettent leurs ressources à disposition, et ce pour n'importe quel programme qui pourrait en avoir besoin. De même, n'importe quel utilisateur peut lancer un programme depuis sa machine sans forcément disposer d'un serveur performant. Un coordonnateur est alors chargé de répartir les tâches de l'ensemble des programmes à exécuter en utilisant toutes les ressources disponibles.
Dans son implantation actuelle, XtremWeb centralise la tâche des coordonnateurs sur un seul serveur. Il est donc plus ouvert que BOINC en ce qui concerne l'attribution des ressources, mais la présence du coordonnateur centralisé rend son architecture très vulnérable aux défaillances.

### 2.3. Modèle pair-à-pair : Zorilla

À la différence de BOINC et d'XtremWeb, Zorilla[7] n'a pas été pensé pour être utilisé sur un très grand nombre d'ordinateurs sur Internet, mais sur des grilles de calcul. Il permet d'utiliser conjointement plusieurs clusters comme s'il s'agissait d'un ensemble uniforme de calculateurs. Mais, de part son approche complètement décentralisée de la gestion de la grille de calcul, il est également intéressant de l'étudier. L'ensemble des calculateurs est considéré comme un vaste réseau pair-à-pair. Chaque machine est à la fois client et serveur, capable de soumettre une tâche au réseau et de participer aux calculs. L'ordonnancement des tâches est, contrairement aux deux systèmes précédemment décrits, complètement décentralisé. C'est le noeud qui soumet la tâche qui est chargé de trouver d'autres noeuds prêts à participer à l'exécution de son programme et effectue cette recherche grâce à une inondation. Zorilla tolère la défaillance de tous les noeuds du système, sauf s'il s'agit d'un noeud ayant lancé l'exécution d'une tâche, auquel cas le résultat est perdu.
Le fonctionnement de Zorilla s'inscrit bien dans une logique pair-à-pair, mais il n'est pas adapté aux conditions d'Internet. Les inondations sont trop coûteuses pour passer à l'échelle, et aucune vérification des résultats n'est effectuée.

## 3. Objectifs et design

Notre objectif est de fournir des outils permettant de réaliser un environnement de calcul distribué complètement pair-à-pair sur Internet. Les applications actuellement utilisées reposent sur des mécanismes centralisés (BOINC, XtremWeb), ou sont conçues pour des environnements contrôlés et fiables (Zorilla) bien différents des conditions d'Internet. Nous avons donc défini une liste des services que nous devons fournir pour permettre de déployer un *desktop-grid* sur Internet, ainsi que des contraintes à respecter pour que l'application bénéficie des avantages du pair-à-pair.

### 3.1. Une application pair-à-pair

Notre architecture est complètement décentralisée. Chacun des services que nous fournissons repose sur des techniques pair-à-pair, sans aucun point de centralisation. Cette logique a également une conséquence sur l'accès aux ressources mises à disposition : chaque utilisateur du système peut lancer une application distribuée en mobilisant les ressources non utilisées. Chaque machine met donc sa puissance à disposition de n'importe quel utilisateur, sans distinction.
L'architecture pair-à-pair sur laquelle repose notre système lui permet de fonctionner dans un environnement très dynamique, caractérisé par l'arrivée et le départ fréquent d'utilisateurs. Il faut toutefois noter que si dans certaines applications pair-à-pair, comme le partage de fichiers, le temps de connexion d'une machine est extrêmement variable et peut être très court, dans une application de calcul distribué

l'utilisateur souhaite mettre sa machine à disposition des autres utilisateurs et il restera donc probablement connecté plus longtemps.
Notre application est conçue pour être déployée sur Internet, sur des machines que nous ne contrôlons pas et dont la fiabilité n'est pas garantie. Nous tentons, dans la mesure du possible, de tolérer les défaillances byzantines. Dans notre cas, nous avons limité celles-ci aux erreurs de calcul qui peuvent survenir lors de l'exécution d'une tâche. Nous avons donc mis en place un système d'exécutions redondantes et de comparaison des résultats, comme c'est le cas dans XtremWeb et BOINC.

### 3.2. Mode déconnecté

Notre réseau ne comporte aucune machine dédiée, il est formé d'ordinateurs de particuliers. Par conséquent, nous autorisons un utilisateur qui a lancé l'exécution d'un programme à se déconnecter sans que l'exécution soit interrompue ou que les résultats soient perdus. Notre système choisit automatiquement une machine du réseau qui doit prendre en charge le déroulement de l'application, et un système d'identification des programmes permet de retrouver ce noeud pour consulter l'avancement des calculs et les résultats.
Chaque programme exécuté génère des fichiers de résultats qui peuvent être très volumineux. Ce phénomène est amplifié par la nécessité de procéder à des exécutions redondantes pour comparer les résultats. Nous avons donc mis en place un système de stockage des données distribué et répliqué, qui permet de tolérer des déconnexions sans perdre d'informations relatives aux applications et de répartir l'utilisation de l'espace disque et de la bande passante.

### 3.3. Modèle d'applications

L'étude des solutions de *desktop-computing* existantes a fait ressortir deux types d'applications distribuées. BOINC et XtremWeb sont limités aux applications de type "Bag of Tasks"[5], pour lesquelles chaque noeud traite une tâche indépendante seul avant d'envoyer son résultat et de passer à une autre tâche, alors que Zorilla permet d'exécuter des applications dans lesquelles les noeuds communiquent au cours de l'exécution. Pour que notre système s'adapte au plus grand nombre de programmes possible, nous avons retenu la seconde solution. Autoriser les machines à échanger des messages au cours de l'exécution pose problème pour la fiabilité des calculs, puisqu'un seul noeud parmi tous ceux qui interviennent peut suffire à fausser les résultats. Afin de pouvoir garantir que les résultats obtenus sont corrects, en effectuant des calculs redondants et en comparant les résultats, nous avons défini une structure de programme qui encapsule les opérations qui produisent un résultat déterministe et peuvent être comparées, tout en autorisant le traitement d'une tâche par plusieurs machines qui communiquent.
Un programme est formé de plusieurs tâches. Le graphe de dépendance des tâches est connu, il permet de déterminer dans quel ordre elles doivent être exécutées. Une tâche produit un résultat déterministe, et le programmeur choisit combien de fois elle doit être exécutée avec succès en produisant le même résultat pour être considérée comme valide. Une tâche peut nécessiter plusieurs machines pour s'exécuter, les tâches communicantes sont donc autorisées. Les différentes exécutions d'une même tâche ne doivent jamais faire intervenir un même pair.
Nous avons donc généralisé la structure de "Bag of Tasks" pour qu'elle autorise les programmes déployés sur plusieurs ordinateurs et adapté la politique d'exécutions redondantes pour que la fiabilité des résultats ne soit pas remise en question. Plus une tâche nécessite de machines, plus le nombre d'exécutions supplémentaires pour en vérifier le résultat doit être élevé. De même, plus une tâche est subdivisée, moins il est coûteux de procéder à des nouveaux calculs en cas de résultats erronés. C'est à l'utilisateur du *desktop-grid* de découper efficacement son programme, nous nous contentons d'offrir le plus de choix possible.

### 3.4. Recherche de ressources

Nous avons mis en place un système de recherche de ressources disponibles pour qu'un noeud responsable de l'exécution d'un programme puisse leur attribuer des tâches. L'ouverture de notre *desktop-grid* aux tâches communicantes implique qu'une tâche peut nécessiter plusieurs machines pour s'exécuter. Afin d'optimiser l'exécution des tâches communicantes, il est important que le temps de transmission d'un message réseau entre tous les noeuds qui participent à leur exécution soit minimal. Par conséquent, nous avons utilisé un algorithme qui permet à un noeud de connaître les autres noeuds du système qui

sont proches de lui au sens de la latence réseau. Grâce à ces informations, nous formons des groupes de noeuds qui ont une latence faible pour pouvoir exécuter efficacement les tâches communicantes.

### 3.5. Approche utilisée

Nous voulons donc construire un système dans lequel n'importe quel utilisateur peut soumettre un programme qui sera exécuté même en cas de déconnexion. Ce programme est formé de "Bag of Tasks" indépendants qui peuvent utiliser plusieurs machines qui communiquent entre elles simultanément. Pour pouvoir les exécuter, nous avons besoin d'un système de recherche qui peut trouver suffisamment de machines disponibles et proches.
Les différents services que nous souhaitons offrir reposent sur des réseaux ayant différentes propriétés. Nous nous sommes donc orientés donc vers une architecture, présentée figure 1, basée sur l'utilisation conjointe de plusieurs overlay, assurant chacun un rôle précis. Les trois overlay que nous avons mis en place sont :

**DHT** : Elle permet d'assurer le stockage des données grâce à un système de fichiers distribué. Ses fonctionnalités sont également utilisées pour attribuer la responsabilité de chaque programme à un noeud. Enfin, sa résistance aux défaillances garantit la connectivité du réseau.

**Overlay de latence** : Il s'agit d'un overlay non structuré qui regroupe les noeuds suivant la latence réseau. Chaque noeud calcule des coordonnées réseau et trouve ses plus proches voisins grâce à des algorithmes de *gossip*. Cet overlay est ensuite utilisé pour former des groupes dans l'overlay de ressources.

**Overlay de ressources** : Ce service utilise l'overlay de latence pour placer chaque noeud disponible dans un groupe et déterminer des responsables de groupe. Ceux-ci organisent un overlay non structuré grâce à des algorithmes de *gossip* afin de pouvoir router des requêtes de recherche de ressources suivant la taille du groupe.

L'utilisation conjointe de ces overlay permet d'assurer toutes les fonctionnalités de notre système. Ils sont fortement interdépendants puisque la DHT et l'overlay de latence sont couplés, et que l'overlay de ressources est construit par dessus l'overlay de latence. Nous décrivons rapidement le fonctionnement de la DHT et de l'overlay de latence pour ensuite nous concentrer sur l'overlay de ressources qui fait intervenir les mécanismes de formation de groupes.

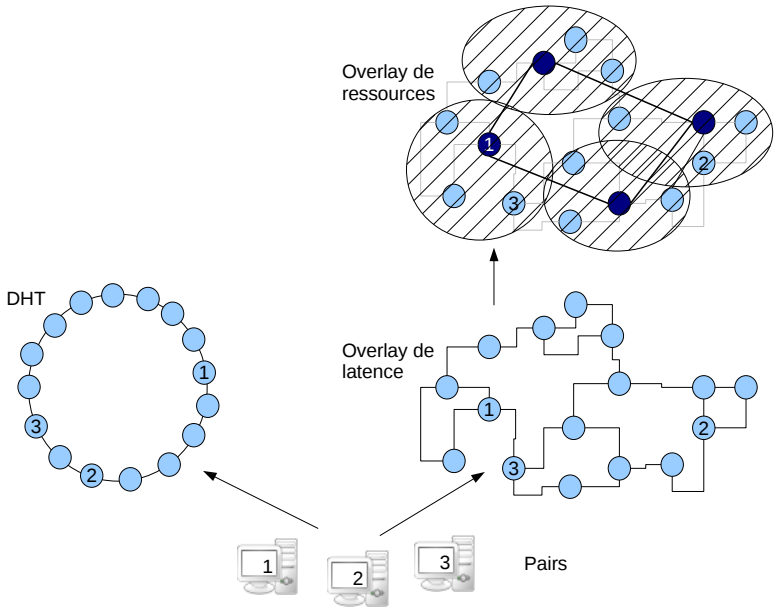


FIG. 1 – Architecture globale de notre système

## 4. DHT et overlay de latence

### 4.1. DHT

Une DHT (table de hachage distribuée) est un overlay structuré qui permet de router un message vers une clé. Chaque noeud possède un identifiant aléatoire et unique déterminé à sa connexion, et est responsable des clés qui sont les plus proches de son identifiant. Dans notre cas, nous considérons l'utilisation de Pastry[12]. Chaque noeud connaît les noeuds ayant des identifiants les plus proches de part et d'autre, son *leafset*, et une table de routage contenant les adresses de noeuds dont les identifiants sont distants. La terminaison du routage s'effectue donc grâce au *leafset*, et la table de routage permet de naviguer rapidement dans l'espace des identifiants. Ces structures sont très solides grâce au maintien agressif du *leafset* qui garantit la connectivité du réseau.

Chaque application qui est soumise à notre système possède un identifiant dans l'espace des clés de la DHT, celle-ci assure donc qu'il y aura en permanence un noeud qui en sera responsable. C'est ce noeud qui est chargé de trouver des ressources pour l'application et de distribuer l'exécution des tâches. L'état de l'application est répliqué sur le *leafset* de ce noeud, ce qui permet d'assurer la continuité de l'exécution même en cas de défaillance du noeud responsable. Il suffit alors de connaître l'identifiant du programme pour pouvoir consulter son état et ses résultats en transmettant une requête à la DHT.

Nous utilisons également Pastis[3], un système de fichier multi-utilisateurs distribué basé sur Pastry. Il permet aux noeuds de lire les fichiers d'une application et d'écrire ses résultats.

### 4.2. Overlay de latence

L'overlay de latence est un overlay non structuré construit grâce à des algorithmes de *gossip*. Chaque noeud contacte un de ses voisins et lui transmet la liste de ses voisins. Celui-ci répond en envoyant sa propre liste, et chacun des deux noeuds sélectionne de nouveaux voisins grâce à cette connaissance augmentée du réseau. Nous avons choisi d'utiliser Vicinity [14], et notre critère de sélection des voisins est basé sur la proximité réseau. Pour cela, nous utilisons le système de coordonnées réseau Vivaldi[6]. Ces coordonnées sont transmis et affinées au cours des différentes communications et permettent d'obtenir par un calcul local une estimation de la latence entre deux noeuds.

Pour éviter un éclatement du réseau en plusieurs composantes connexes, un ensemble de voisins clusterisé doit être associé à un ensemble de voisins aléatoire, qui garantissent la connectivité du réseau. Nous pourrions pour cela utiliser l'algorithme Cyclon [13], mais, comme nous allons le voir par la suite, la DHT peut également remplir ce rôle.

Grâce aux coordonnées réseau, chaque noeud est capable d'évaluer sa latence vis à vis d'un autre noeud du réseau. Il utilise cette information pour sélectionner ses plus proches voisins. Cet overlay sera utilisé par la suite pour construire l'overlay de ressources.

### 4.3. Joint overlay

Lorsque l'on utilise plusieurs overlay distincts pour former une architecture réseau, il est possible de partager les informations entre eux, afin de réduire leur coût ou d'augmenter leur efficacité[11]. Dans notre cas, l'utilisation d'une DHT et d'un overlay de proximité permet d'appliquer cette idée. Le *leafset* de la DHT garantit la connectivité, et, puisqu'il n'est pas corrélé avec la proximité réseau, offre un ensemble de pairs aléatoires à l'overlay de latence. De même, l'ensemble clusterisé de l'overlay de latence permet d'alimenter la table de routage de la DHT avec des noeuds proches, offrant ainsi de bonnes performances de routage. Le principe des overlay joints nous permet donc de réduire le coût global de notre architecture, en ne considérant plus les overlay comme des réseaux logiques complètement séparés, mais en mettant en commun leurs informations.

## 5. Architecture de l'overlay de ressources

L'overlay de ressources permet de rechercher efficacement des groupes de noeuds disponibles pour exécuter des tâches. Seules les machines disponibles sont concernées par cet overlay, les ordinateurs qui exécutent des calculs n'en font pas partie mais participent tout de même à sa construction en relayant des messages. Comme expliqué dans la section 3.4, nous souhaitons être capables d'effectuer des recherches sur des groupes de machines à faible latence en indiquant le nombre minimal de machines nécessaires.

### 5.1. Nécessité de construire des groupes

On peut dans un premier temps considérer qu'un groupe de machines est constitué d'un noeud et de ses voisins sur l'overlay des distances réseau qui, comme lui, sont disponibles. Dans ce cas, chaque noeud fait partie de l'overlay des recherches et déclare un groupe de la taille de l'ensemble ainsi défini. Si l'on considère que la topologie réseau permet d'effectuer une clusterisation clairement délimitée, on a alors pour chaque groupe de taille $n$, $n$ noeuds dans l'overlay des ressources. Chacun d'entre eux affirme disposer d'un groupe de $n$ machines, et lorsque l'on recherche un groupe d'une taille $n$, pour $k$ groupes réellement existants, on obtient $k * n$ réponses. Ces réponses créent beaucoup de messages inutiles puisqu'elles sont redondantes, et, de plus, elles empêchent le noeud qui effectue la recherche de déterminer clairement combien de groupes il a à sa disposition. Enfin, avec ce système, un noeud peut faire partie de plusieurs groupes différents. Chaque noeud décide si ses voisins font partie de son groupe, un noeud ne se réserve pas exclusivement à un groupe.

Pour offrir une recherche plus précise et alléger l'overlay, nous avons décidé de mettre en place un système de création de groupes plus élaboré. Chaque noeud ne fait partie que d'un seul groupe, et chaque groupe est représenté par un responsable qui fait partie de l'overlay de recherche. Ainsi, chaque groupe, et indirectement chaque noeud, n'est représenté qu'une fois dans l'overlay de recherche. Le nombre de réponses obtenues lors d'une recherche dans cet overlay est donc représentatif des ressources réellement disponibles.

### 5.2. Formation des groupes

Les groupes de noeuds que nous voulons former rassemblent des machines qui sont proches au sens de la latence réseau. Nous disposons de l'overlay de latence, décrit partie 4.2, qui permet à chaque noeud de connaître les machines qui sont proches de lui. L'unité de mesure de distance entre deux noeuds devient alors le nombre de sauts minimum qu'il faut faire sur l'overlay pour passer d'un noeud à l'autre. Chaque groupe est formé autour d'un noeud, le responsable du groupe, et inclut tous les noeuds situés à une distance inférieure ou égale à $r$, rayon du groupe. Certains noeuds peuvent être inclus de cette façon dans plusieurs groupes, ils doivent choisir un seul et unique groupe auquel se rattacher.

#### 5.2.1. Algorithme général

Ces groupes doivent être créés dynamiquement, ils sont modifiés au cours du temps car l'overlay de proximité réseau est lui aussi modifié. Il faut donc réévaluer régulièrement quels sont les noeuds qui font partie d'un groupe. De plus, nous voulons que si deux groupes se trouvent trop proches dans l'overlay, ils fusionnent en un seul groupe, et que si un noeud reste trop longtemps sans groupe, il prenne l'initiative de devenir responsable d'un groupe. L'algorithme SONDe[9] permet de répartir des objets sur un overlay de façon équilibrée, en garantissant une distance maximale entre un noeud de l'overlay et un de ces objets. L'idée générale de ce système peut être adaptée pour produire le résultat que nous cherchons. Tout d'abord, nous supposons que chaque noeud est capable d'identifier un message d'inondation qu'il a déjà reçu afin de ne pas le propager à nouveau. Ceci peut facilement être réalisé en choisissant un numéro aléatoire associé à chaque inondation et en maintenant une mémoire locale des dernières inondations connues à chaque noeud. Nous utilisons deux types de messages : des inondations, provenant des responsables de groupe, et qui sont transmises sur un rayon $r$ aux voisins de l'overlay de latence, et des messages "membre", qui sont des messages directs envoyés par des noeuds aux responsables de groupe. Aux données attachées à chaque message de notre environnement de calcul distribué nous ajoutons la taille du groupe lorsque le noeud est responsable de groupe.

- Un noeud responsable de groupe envoie régulièrement un message d'inondation à tous ses voisins de l'overlay de latence (algo 1 ligne 22). Cette inondation se fait sur un rayon $R$, et atteint donc tous les noeuds à une distance inférieure ou égale à $R$ (algo 2 ligne 15). Tous les messages "membre" reçus entre deux inondations permettent de connaître les noeuds qui font partie du groupe (algo 3 ligne 3). La liste des membres est réinitialisée à chaque nouvelle inondation (algo 1 ligne 18), et donc, après l'inondation numéro $n$, la liste des membres du groupe est formée de tous les noeuds qui ont envoyé un message "membre" au responsable du groupe entre les inondations $n-1$ et $n$.
- Un noeud qui fait partie d'un groupe et qui reçoit un message d'inondation de la part d'un responsable de groupe répond par un message "membre" si ce responsable est bien celui du groupe dont il fait partie (algo 2 ligne 12). S'il n'avait pas de responsable de groupe, il choisit celui-ci et répond par

un message “membre”. Sinon, il calcule, grâce aux coordonnées de l’overlay de latence, quel est le responsable de groupe le plus proche et en change si nécessaire (algo 2 ligne 9).
– Un responsable de groupe qui reçoit une inondation d’un autre responsable de groupe ne fait rien si son groupe est plus grand, sinon il arrête d’être responsable de groupe. En cas d’égalité, c’est le groupe le plus jeune qui disparaît (algo 2 ligne 3).
– Un noeud qui n’a pas de responsable de groupe pendant une période donnée devient responsable de groupe (algo 1 ligne 13).
– Un noeud qui a un responsable de groupe et qui ne reçoit plus de message d’inondation venant de lui pendant une période donnée n’a plus de responsable de groupe (algo 1 ligne 10).

**Algorithme 1** : Procédure répétée au cours de l’exécution

```
/* T : constante du programme, définit le temps entre 2 inondations;
R : constante du programme, définit le rayon d'un groupe;
SEUIL1 : constante du programme, définit le temps d'attente avant qu'un noeud membre d'un groupe considère que son
responsable de groupe s'est déconnecté;
SEUIL2 : constante du programme, définit le temps d'attente avant qu'un noeud sans groupe décide d'en créer un;
adresseNoeud : adresse réseau du noeud courant;
responsableGroupe : booléen, vrai si le noeud courant est responsable d'un groupe, faux sinon;
timer : entier, utilisé par les responsables de groupe pour compter l'âge du groupe et par les autres noeuds pour compter les
rondes sans inondation de son responsable et sans groupe;
voisins : ensemble des voisins sur l'overlay de latence du noeud courant;
responsable : adresse du noeud responsable du groupe dont le noeud courant fait partie;
membres : ensemble des adresses des noeuds qui font partie du groupe dont le noeud courant est responsable;
reçus : ensemble des adresses des noeuds qui ont répondu à un responsable de groupe entre deux inondations;
tailleGroupe : entier, indique la taille du groupe dont le noeud courant est responsable;
                                                                                                              */
1  initialisation
2      responsableGroupe← faux;
3      timer ← 0;
4      responsable ← null ;
5  fin d'initialisation
6  répéter
7      attendre T temps;
8      timer ← timer + 1;
9      si !responsableGroupe et responsable ≠ null et timer > SEUIL1 alors
10         responsable ← null;
11         timer ← 0;
12     si !responsableGroupe et responsable = null et timer > SEUIL2 alors
13         responsableGroupe ← vrai;
14         membre s← ∅;
15         reçus ← ∅;
16         timer ← 0;
17     si responsableGroupe alors
18         membres ← reçus;
19         reçus ← ∅ ;
20         tailleGroupe ← cardinal(membres);
21         pour chaque noeud de voisins faire
22             envoyer(["flood", R, tailleGroupe, timer, adresseNoeud]);
23 indéfiniment
```

**Algorithme 2** : Réception d'un message d'inondation

```
/* inondationDejaRecue : fonction indiquant si un message d'inondation a déjà été reçu;
distance : fonction indiquant la distance réseau du noeud courant au noeud en paramètre, si le paramètre est null la distance est infinie;
*/
1  à la réception de ["inondation", r, taille, age, adresse] faire
2      si !inondationDejaRecue() alors
3          si responsableGroupe et (taille > tailleGroupe ou (taille = tailleGroupe et timer < age)) alors
4              responsableGroupe ← faux;
5              responsable ← adresse;
6          si !responsableGroupe alors
7              si responsable ≠ adresse alors
8                  si distance (adr)<distance (resp) alors
9                      resp ← adr;
10             si responsable = adresse alors
11                 timer ← 0;
12                 envoyer ["membre", adresseNoeud] à adresse;
13                 si r > 1 alors
14                     pour chaque noeud de voisins faire
15                         envoyer(["inondation", r − 1, taille, age, adresse]);
```

**Algorithme 3** : Réception d'un message de participation au groupe

```
1  à la réception de ["membre", adresse] faire
2      si responsableGroupe alors
3          reçus ← reçus ∪{adresse};
```

Ce protocole équilibre le nombre de transmissions de messages entre les noeuds du réseau. Il faut en effet éviter de surcharger les responsables de groupe. Pour cela, la signalisation du groupe se fait grâce à une inondation, et est donc relayée par les autres noeuds de l'overlay. Le responsable de groupe ne reçoit jamais de messages destinés à vérifier s'il est toujours présent, puisqu'il se signale régulièrement avec ses messages d'inondation. Les membres de son groupe détectent sa défaillance grâce à un temporisateur. Le responsable de groupe reçoit après chaque inondation un message de chacun des membres du groupe, c'est ainsi qu'il détecte les défaillances des noeuds membres de son groupe et permet la mise à jour du groupe en fonction des modifications de l'overlay de proximité réseau. Les collisions entre groupes trop proches sont détectées lorsqu'un responsable de groupe reçoit un message d'inondation d'un autre responsable de groupe. On donne alors la priorité au groupe le plus grand, afin de minimiser le nombre de noeuds qui doivent trouver un nouveau groupe. Lorsqu'un noeud peut choisir entre plusieurs groupes, il préfère répondre au responsable dont il est le plus proche. On privilégie donc les groupes ayant une latence interne faible.

#### 5.2.2. Optimisations

Ce protocole peut être facilement amélioré de façon à éviter les collisions de groupes. Lorsqu'un responsable de groupe se déconnecte, tous les membres de ce groupe vont attendre pour abandonner ce responsable, puis attendre à nouveau avant de créer un groupe s'ils ne reçoivent pas de messages d'inondation. Il faut bien entendu moduler cette dernière durée par un facteur aléatoire, afin d'éviter que tous les noeuds qui faisaient partie d'un groupe, et donc sont proches les uns des autres, créent tous

un groupe au même moment.
L'overlay utilisé comme support pour créer les groupes de ressources peut changer au cours du temps. Étant donné que nous procédons par inondations successives pour former nos groupes, il est important que les ensembles de noeuds que nous utilisons pour transmettre les messages restent le plus stable possible entre deux phases d'inondations. De cette façon, les membres d'un groupe resteront les mêmes. Les noeuds les moins stables de l'overlay sont ceux qui, dans la liste de voisins triés par ordre de proximité, sont les derniers. Plus la distance par rapport au noeud est élevée, plus le nombre de candidats dans le réseau est important, et une petite variation de coordonnées peut donc facilement modifier ces voisins. À l'inverse, les noeuds considérés comme très proches auront plus de chance de rester présents. On peut donc modifier l'inondation pour ne diffuser le message qu'à un certain pourcentage des voisins, triés par ordre de proximité, pour renforcer la stabilité des groupes.
Dans le cas idéal, le responsable d'un groupe serait le noeud qui représenterait le mieux le centre du groupe, et qui donc minimiserait les distances par rapport aux autres membres du groupe. Or, ces noeuds seront souvent choisis comme voisins dans l'overlay de latence réseau, puisqu'ils seront proches de nombreuses machines. Ils auront donc un degré entrant important, et seront plus susceptibles de recevoir des messages d'inondation provenant d'autres noeuds. Ils ont donc moins de chances de devenir responsables de groupe. À l'inverse, les noeuds très isolés, qui ne sont choisis comme voisins par personne, vont forcément devenir responsables de groupe. Ils disposent de voisins et procèdent donc à une inondation, formant un groupe de machines dont ils sont distants, et qui peuvent même être distantes entre elles, comme on le voit figure 2.
Nous voulons donc que ces noeuds restent isolés et forment un groupe dont ils seront le seul membre. La solution est alors d'ignorer les messages d'inondations servant à former des groupes lorsque la relation de voisinage n'est pas réciproque. Le noeud vérifie donc que le message d'inondation lui a été transmis par un noeud qui appartient à sa liste de noeuds voisins avant de traiter l'information. La liste des voisins réseau est donc utilisée dans sa totalité pour effectuer un filtrage des messages entrants, et une sous-partie, formée des meilleurs éléments, sert à choisir les destinataires des messages.

### 5.3. Construction de l'overlay de recherche de groupes

#### 5.3.1. Formation de l'overlay

Les noeuds responsables de groupes forment un overlay qui doit permettre de chercher un groupe en indiquant la taille de groupe minimale souhaitée. Une recherche n'attend pas un unique résultat, les tâches sont calculées plusieurs fois et on a donc besoin de plusieurs groupes pour chacune d'entre elles. Il faut donc être capable de router les recherches vers les groupes concernés et de les diffuser à plusieurs groupes. La recherche s'effectue donc en deux étapes :
– acheminement de la requête vers un responsable de groupe correspondant aux critères grâce à une liste de responsables de groupes de tailles variées
– transmission de la recherche à des groupes qui correspondent aux critères grâce à une liste de groupes ayant une taille proche
Chaque noeud de l'overlay des ressources a donc besoin de maintenir ces deux ensemble de noeuds. Le premier a pour but de disposer de la plus grande diversité possible de tailles de groupes de façon à être capable de router rapidement. Lorsque l'on ajoute un noeud à cet ensemble et qu'il dépasse sa taille maximale, on classe ses éléments par ordre de taille de groupe et on calcule la différence entre deux noeuds successifs. Le noeud qui produit la différence la plus faible est retiré de la liste. Le second ensemble contient les noeuds de l'overlay dont la taille des groupes est proche. Ces deux ensembles sont entretenus par *gossip*, exactement de la même façon que pour l'overlay de latence réseau en section 4.2.

#### 5.3.2. Optimisations

Cet overlay est très dynamique, les responsables de groupes qui acceptent une tâche ne sont plus disponibles et sortent de l'overlay. Dans les deux ensembles de noeuds maintenus, nous devons donc toujours privilégier les noeuds dont nous avons une information récente. Les tailles de groupes varient également avec le temps. Il suffit qu'un noeud d'un groupe se déconnecte pour que sa taille diminue. Les variations liées à l'overlay de latence augmentent encore l'instabilité. Par conséquent, notre gestion de l'ensemble des noeuds de taille proche doit être tolérante à ces changements de taille, afin de ne pas systématiquement devoir trouver de nouveaux voisins à chaque changement de taille de groupe. On ajoute donc

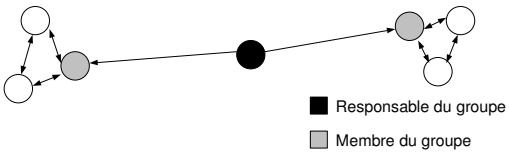


FIG. 2 – Formation d'un groupe non optimal à cause d'un noeud isolé

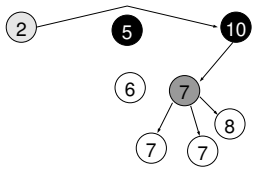


FIG. 3 – Recherche de groupes de taille 7 sur l'overlay des ressources

une tolérance, en limitant l'optimisation de la liste de groupes de taille voisine par un paramètre `delta`. Ainsi, même lorsque cette liste aura convergé, on trouvera toujours des groupes dont la taille varie sur un petit intervalle, ce qui fait que cette liste restera en grande partie valide même si la taille du groupe change. Par exemple, avec un `delta` de 1, si la taille du groupe change de 1, la liste restera statistiquement valide aux $\frac{2}{3}$. Nous obligeons chaque groupe à avoir dans ses voisins au minimum un groupe de taille supérieure à la sienne et un groupe de taille inférieure. De cette façon, nous assurons la connexité de l'overlay.

#### 5.3.3. Routage des recherches de groupe

Comme cet overlay est non-structuré et très dynamique, la recherche n'est pas forcément déterministe, et il faut pouvoir donner une seconde chance à une recherche dans une impasse qui ne donne aucun résultat. Les noeuds qui viennent de rejoindre l'overlay ne connaissent pas forcément beaucoup d'autres noeuds, si bien que si un message de recherche de groupes arrive sur un de ces noeuds, il risque de ne plus progresser.

L'objectif premier lors du routage d'une recherche de ressources est d'atteindre un groupe qui a une taille supérieure ou égale à la taille recherchée. Cette phase s'effectue grâce à l'ensemble des groupes de taille variée. Si un noeud dispose de plusieurs candidats répondant à ce critère, il choisit la taille de groupe la plus faible. Si le noeud ne connaît aucun groupe de taille suffisante, il transmet le message au groupe de plus grande taille qu'il connaît. Si ce groupe n'a pas une taille supérieure au sien, c'est que le message est dans une impasse, il ne peut plus progresser. La recherche passe alors en mode aléatoire et a un nombre de transmissions maximum pour aboutir. Ce fonctionnement basé sur un *TTL* permet d'éliminer les recherches qui n'ont pas de résultats. Le message est alors autorisé à ne plus progresser, et les noeuds le transmettent à un candidat choisi aux hasard parmi les x meilleures destinations.

Une fois que la requête est arrivée à un responsable de groupe répondant aux critères, on essaie de la transmettre au groupe de taille la plus petite possible mais suffisante pour la recherche. Cette phase, également limitée grâce à un *TTL*, permet d'éviter de casser des groupes de grande taille pour des petites tâches.

Enfin, la dernière étape est une inondation grâce à l'ensemble des groupes de taille proche. Cette inondation a un rayon limité, et n'est transmise qu'aux noeuds qui pourraient potentiellement atteindre un groupe de bonne taille, dont la taille est supérieure à la taille recherchée moins le nombre de sauts restants dans l'inondation multiplié par `delta`. Lorsqu'un responsable de groupe correspond aux critères, il contacte le noeud demandeur à l'origine de la recherche pour lui signaler sa disponibilité. Si le noeud qui a lancé la recherche a toujours besoin de ces ressources, il peut alors réserver le groupe mais également indiquer si l'inondation doit être relancée, avec un rayon réinitialisé, pour trouver des groupes supplémentaires. La figure 3 présente le principe général du routage dans l'overlay des ressources.

### 5.4. Recherche de l'overlay

Pour chercher des ressources, les noeuds responsables de l'exécution de programmes doivent connaître un noeud de cet overlay. De même, un noeud qui devient responsable de groupe doit pouvoir se rattacher à l'overlay de ressources. Chaque noeud, lors d'une communication, indique s'il est responsable de groupe. Il est donc possible de maintenir une liste des responsables de groupes dont un noeud a eu connaissance, en privilégiant les informations récentes, pour qu'il connaisse un point d'entrée lorsqu'il en a besoin. Si le noeud ne connaît toujours pas de noeuds de l'overlay lorsqu'il en a besoin, il peut

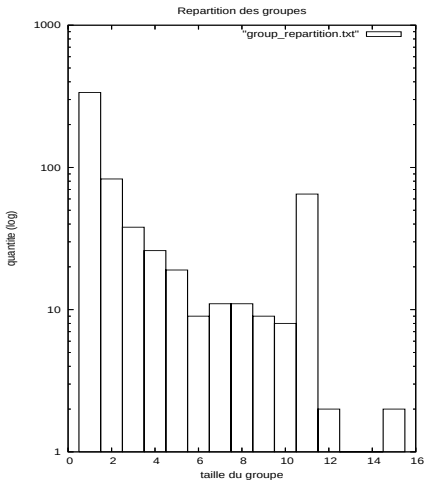


FIG. 4 – Répartition des groupes suivant leur taille

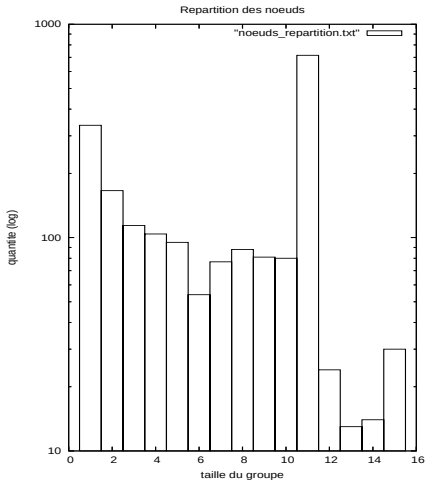


FIG. 5 – Répartition des noeuds dans les différents groupes

interroger les noeuds proches de lui grâce à une inondation. Enfin, en dernier recours, nous pouvons utiliser la DHT pour nous assurer que certains noeuds connaissent toujours une machine de l'overlay des ressources. Pour cela, il suffit de choisir un suffixe de clé dans la DHT, et chaque noeud responsable d'une clé ayant ce suffixe doit maintenir une liste de noeuds de l'overlay de façon agressive. L'utilisation d'un suffixe permet de répartir les noeuds équitablement sur la DHT.

### 5.5. Bilan

L'overlay des ressources est utilisé pour permettre de trouver rapidement des machines disponibles pour effectuer des calculs. Celles-ci sont groupées par proximité réseau de façon à être affectées à des tâches communicantes. Nous utilisons un algorithme inspiré de SONDe pour délimiter ces groupes et en attribuer la responsabilité à un noeud. Seuls ces noeuds responsables font partie de l'overlay, ainsi la recherche est plus efficace et les résultats sont plus précis. Il est alors possible de router les recherches de ressources suivant la taille du groupe désirée. Comme cet overlay ne contient qu'une sous-partie des noeuds du système, nous avons prévu un mécanisme de localisation de l'overlay afin de pouvoir le rejoindre ou lui transmettre des messages.

## 6. Évaluation

Nous avons évalué notre solution à l'aide du simulateur à évènements discrets Peersim[10] afin de tester notre algorithme de création de groupes. Pour pouvoir prendre en compte la latence entre les machines du réseau, nous avons généré un réseau de machines dont la topologie est proche du modèle d'Internet en utilisant l'algorithme FKP[8]. Nous avons dans un premier temps vérifié la stabilité de notre overlay de latence basé sur les coordonnées réseau, puis nous avons étudié plus précisément la formation des groupes de machines. Étant donné que notre architecture repose sur une DHT et des algorithmes de *gossip*, le passage à l'échelle est déjà assuré. Les principales mesures à effectuer concernent donc la stabilité des groupes ainsi que la taille des ensembles de noeuds formés. Les simulations à évènements discrets sont très coûteuses en temps de calcul, c'est pourquoi nous nous sommes par la suite concentrés sur un nombre de noeuds relativement réduit : 2000 machines.

La figure 4 nous montre que nous obtenons bien des groupes de taille variée, et la figure 5 indique que les noeuds sont répartis de manière uniforme parmi ces tailles de groupe. Nous disposons donc de ressources variées, adaptées à l'exécution de différents types d'applications. D'autres résultats, non présentés ici, confirment la stabilité de ces groupes et donc la mise en place effective de l'algorithme de recherche de ressources décrit en section 5.3.

D'autres expérimentations doivent être effectuées pour valider entièrement notre architecture afin de tester le comportement de l'overlay de ressources lorsque des machines sont effectivement réquisitionnées et utilisées, et sont donc retirées.

## 7. Conclusion

Dans cet article, nous avons proposé une architecture complètement décentralisée permettant la mise en place d'un réseau de partage de ressources de calcul. Son fonctionnement repose sur 3 overlay et fait intervenir des techniques de réduction de coût de maintenance. L'overlay de ressources est particulier, puisqu'il est construit au dessus de l'overlay de latence. Cette superposition et la formation de groupes entraînent des problèmes de stabilité que nous avons étudiés. Nous proposons également un algorithme permettant de former des groupes de pairs suivant leur disponibilité et d'effectuer une recherche parmi ces groupes.
Notre solution repose sur une DHT et des algorithmes de *gossip*, elle permet donc un passage à l'échelle dans un contexte pair-à-pair. Le système de formation de groupe est également fonctionnel mais des tests plus approfondis doivent être réalisés pour étudier son comportement dans des situations plus dynamiques.

## Bibliographie


1. D.P. Anderson. BOINC : a system for public-resource computing and storage. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.
2. D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, et D. Werthimer. SETI@ home : an experiment in public-resource computing. *Communications of the ACM*, 45(11) :56–61, 2002.
3. J.M. Busca, F. Picconi, et P. Sens. Pastis : A highly-scalable multi-user peer-to-peer file system. *Lecture notes in computer science*, pages 1173–1182.
4. F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, et O. Lodygensky. Computing on large-scale distributed systems : XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3) :417–437, 2005.
5. W. Cirne, F. Brasileiro, J. Sauve, N. Andrade, D. Paranhos, E. Santos-Neto, R. Medeiros, et F. Silva. Grid computing for Bag-of-Tasks applications. *Proceedings of the IFIP I3E2003*, 2003.
6. F. Dabek, R. Cox, F. Kaashoek, et R. Morris. Vivaldi : a decentralized network coordinate system. *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, 2004.
7. N. Drost, R.V. van Nieuwpoort, et H. Bal. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. *Proceedings of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid06)*, 2006.
8. A. Fabrikant, E. Koutsoupias, et C.H. Papadimitriou. Heuristically optimized trade-offs : A new paradigm for power laws in the Internet. *ICALP 2002*, pages 110–122, 2002.
9. V. Gramoli, A.M. Kermarrec, E. Le Merrer, et D. Neveux. SONDe : Contrôle de densité auto-organisante de fonctions réseaux pair à pair. *Proceedings of Algotel 2006*, 2006.
10. M. Jelasity, G.P. Jesi, A. Montresor, et S. Voulgaris. PeerSim : A Peer-to-Peer Simulator. *URL : http ://peersim. sourceforge. net*.
11. B. Maniymaran, M. Bertier, et A.M. Kermarrec. Build One, Get One Free : Leveraging the Coexistence of Multiple P2P Overlay Networks. *Proceedings of the IEEE ICDCS 2007*.
12. A. Rowstron et P. Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 11 :329–350, 2001.
13. S. Voulgaris, D. Gavidia, et M. van Steen. CYCLON : Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13(2) :197–217, 2005.
14. S. Voulgaris et M. van Steen. Epidemic-style management of semantic overlays for content-based searching. *Intl Conf. on Parallel and Distributed Computing (Euro-Par)*.