Table des matières

Table des figures

Introduction Générale

L'augmentation perpétuelle de la quantité des données à traiter, due à la convergence des technologies de l'information, tend à rendre le traitement sur des ordinateurs isolés[1] obsolète. De ce fait, il se manifeste un besoin de pouvoir accéder à tout un ensemble d'informations à tout moment et de n'importe où ; ce qui pourrait poser en toute évidence quelques problèmes : la localisation des données, le temps d'accès aux ressources et le temps de traitement associé.

Une première solution serait donc d'agréger les capacités (de stockage et de calcul) de plusieurs ordinateurs pour réaliser un seul et même service. Cet agrégat est appelé ***grille informatique***. Ce type de système bien qu'il soit distribué nécessite des réseaux de haut débit. A cet effet, une seconde solution utilisant l'internet est apparue avec les systèmes Pair-à-Pair[2](ou P2P). Les environnements P2P sont de nature complètement distribués. Dans ces systèmes, chaque pair peut être à la fois client et serveur. Plus généralement, chaque pair peut jouer l'ensemble des rôles existants dans le système.

Depuis des années, des technologies basées sur le concept Pair-à-Pair ne cessent de gagner du terrain à cause des nombreuses et diverses utilisations que l'on peut en faire. Le partage à large échelle d'informations (fichiers, objets multimédias, …) proposé par les premiers systèmes Pair-à-Pair a démontré toute leur importance dans le domaine du multimédia et du commerce en ligne.

Cependant, même si l'utilisation des systèmes P2P s'avère plus complexe, elle peut être très bénéfique pour des systèmes à forte charge transactionnelle tels que les systèmes de réservation. Pour ce faire, les données doivent être répliquées pour garantir la disponibilité des données tout en permettant de paralléliser les traitements. Néanmoins, garantir un accès rapide et cohérent des données dans de telles architectures constitue un défi majeur. En particulier, une gestion centralisée est inadéquate à cause de sa vulnérabilité et de son incapacité à passer à l'échelle[3].

Une solution à ce problème a été proposée dans les travaux présentés dans *DTR* [2] pour améliorer celle déjà proposée par Leganet [1]. Les algorithmes utilisés dans [2] s'appuient sur

[1] Aucune interconnexion entre ces ordinateurs

[2] Connu sous le nom Peer-to-Peer ou P2P en anglais

[3] L'augmentation du nombre de nœuds ne dégrade pas la performance du système

une hypothèse forte à savoir la connaissance à priori du temps d'exécution des transactions, pour s'avoir vers quel nœud il faut acheminer la requête.

L'objectif principal de ce stage est de mettre en place des algorithmes efficaces pour le traitement des transactions dans les bases de données à large échelle tout en ignorant la durée d'exécution de ces transactions et qui prennent en compte les caractéristiques de ces systèmes. Une base de données à large échelle étant une base qui repose sur un environnement Pair-à-Pair.

Pour ce faire, nous adoptons un plan composé de trois chapitres. Notre premier chapitre est consacré à l'étude des systèmes à large échelle tels que les grilles de calcul et les systèmes P2P. Le deuxième chapitre présente les mécanismes de traitement de transactions dans ces systèmes et les différents algorithmes que nous proposons. Enfin dans le troisième chapitre, nous présentons et commentons les résultats que nous avons obtenus à la suite de nos différentes simulations effectuées avec l'outil FreePastry en vue de valider nos algorithmes.

Chapitre 1

Etude et Compréhension des systèmes à large échelle

Dans ce chapitre du mémoire, nous introduisons notre contexte d'étude. Nous présentons les infrastructures d'exécution que nous visons : les systèmes Pair-à-Pair et les grilles informatiques.

L'objectif consiste à mieux cerner le terme système à large échelle, en précisant notamment leur mode de fonctionnement, les avantages et inconvénients dans la gestion des données.

En premier lieu nous faisons l'étude des systèmes Pair-à-Pair pour terminer en second lieu par les Grilles Informatiques.

I. Les Systèmes P2P: Avantages et Inconvénients

Les systèmes Pair-à-Pair sont en plein essor depuis maintenant plusieurs années. Ce sont des réseaux distribués, sans aucune organisation hiérarchique ou de contrôle centralisé. Ce paradigme de réseau entièrement décentralisé permet de concevoir des systèmes de très grande taille à forte disponibilité, tout cela à faible coût. Les pairs forment des réseaux overlay auto-organisant au dessus du réseau physique IP. Ils permettent l'accès à ses ressources par d'autres systèmes et supportent un partage de ressources, qui exige les propriétés de tolérance aux pannes, d'auto-organisation et d'extensibilité massive. Les systèmes P2P vont au-delà des services offerts par le client-serveur en ayant une symétrie dans des rôles où un client peut aussi être un serveur. Les différents systèmes P2P peuvent se différencier par leurs fonctionnalités. Certains permettent des recherches complexes (du type "nom de fichier contenant x et y"), alors que d'autres se limitent à des recherches exactes (du type "nom de fichier = x").

Cependant, nous pouvons les classer selon leurs fonctionnalités en trois grandes familles: les ***réseaux hybrides*** dans lesquels il existe encore des serveurs centraux ; les ***réseaux non structurés***, dans lesquels les liens entre les nœuds sont relativement aléatoires, et les ***réseaux structurés***, dans lesquels chaque nœud possède une table de routage précise.

1. Les réseaux hybrides

Les réseaux hybrides ne sont pas entièrement décentralisés. Un serveur est utilisé comme annuaire des ressources disponibles, permettant une recherche et une localisation faciles. En revanche, les transferts de données se passent directement entre les pairs, sans faire intervenir le serveur. Ces réseaux ont été les premiers proposés dans une optique de partage de fichiers. Comme exemple, nous pouvons citer Napster et KaZaA [11].

1.1. Napster

Napster repose sur un unique serveur d'annuaire. Chaque nœud qui se connecte au réseau s'inscrit sur ce serveur et y publie la liste des fichiers qu'il partage. Lors d'une recherche, le serveur est interrogé et il re tourne la liste des pairs satisfaisant la requête. Les requêtes peuvent être complexes, mais la seule limite est la puissance du serveur. Cette puissance est d'ailleurs la limite de ce type de réseau car le serveur constitue un goulot d'étranglement. Bien que n'intervenant pas dans les transferts de fichiers, le serveur est nécessaire lors de chaque insertion ou recherche de données, ainsi qu'au fur et à mesure du téléchargement pour actualiser les sources. Son dimensionnement et sa disponibilité sont donc critiques pour le fonctionnement du système tout entier.

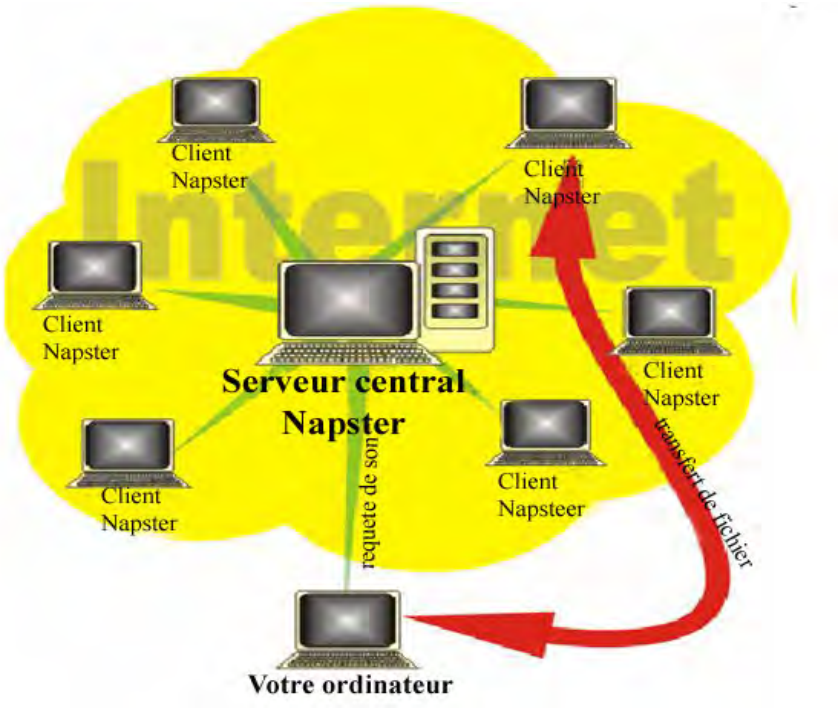

Figure 1 : Topologie du système Napster

1.2. KaZaA

KaZaA [11] étend le système centralisé de Napster à un serveur lui même distribué. KaZaA introduit ainsi le concept de Super-Pair, un sous-ensemble de pairs qui joue le rôle du serveur de Napster. KaZaA utilise des super-pairs désignés qui ont une plus grande bande passante de connectivité, et se portent volontaire pour se faire élire pour faciliter la recherche par la mise en cache des métadonnées. Chaque pair ***"normal"*** connaît un Super-Pair, sur lequel il publie ses partages et y envoie toutes ses requêtes. Les Super-Pairs communiquent ensuite entre eux. Pour relier les Super-Pairs, des réseaux non structurés ou structurés, qui seront décrits dans les sections suivantes de ce mémoire, peuvent être utilisés. Cependant, les développeurs de KaZaA ont opté pour une approche non structurée.

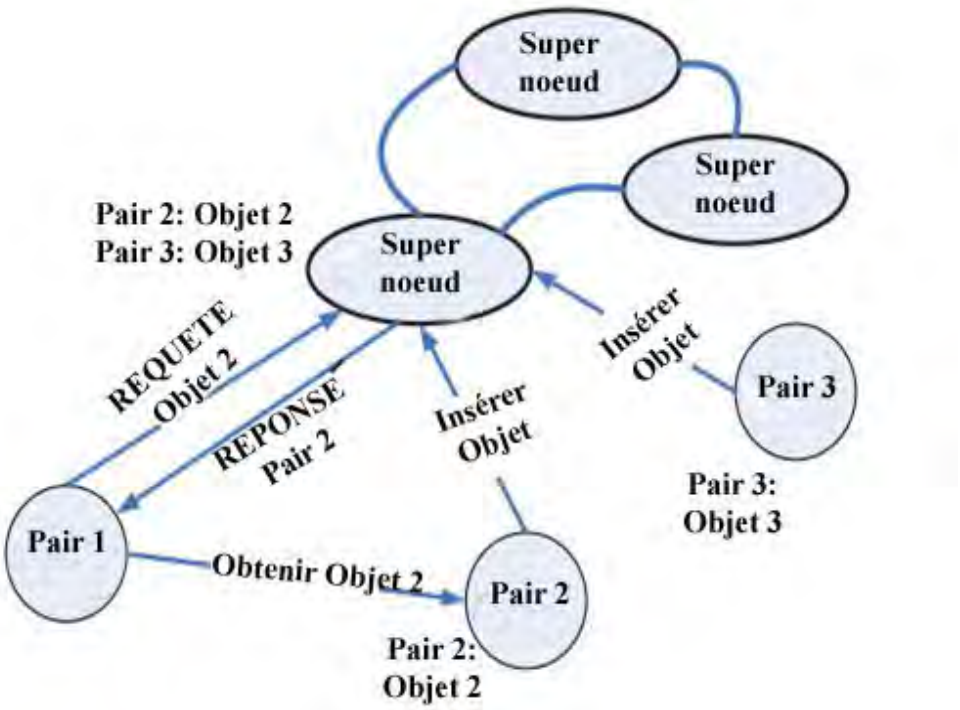

Figure 2 : Topologie du Système KaZaA et mode de recherche

Cette organisation permet de réaliser des requêtes aussi complexes que dans Napster, tout en évitant les faiblesses de passage à l'échelle de la centralisation. Néanmoins, le choix des Super-Pairs n'est pas facile, du moment que le bon fonctionnent du réseau est inhérent à la puissance des nœuds et de la stabilité de leur connexion. Bien qu'il soit intéressant de séparer les pairs en catégories selon leurs capacités, il est dommage de se limiter à deux types de nœuds quand les différences de capacités peuvent atteindre plusieurs ordres de magnitude.

En l'absence de centralisation claire, la présence d'une autorité ponctuelle est remise en cause. Les Super-Pairs ne sont plus contrôlés par une seule autorité, puisqu'il s'agit de machines d'utilisateurs du réseau. En tant que machines non contrôlées, certains Super-Pairs peuvent donc être eux-mêmes malicieux ; ces Super-Pairs ne peuvent donc pas de manière triviale distribuer des certificats fiables aux autres nœuds. De plus, si chaque pair se connecte à un et un seul Super-Pair qui peut se trouver être malicieux, leur accès au réseau peut être entièrement biaisé.

2. Les réseaux non structurés

Les réseaux P2P non structurés sont ceux dans lesquels il n'y a ni un répertoire centralisé ni un contrôle précis sur la topologie du réseau ou emplacement de fichiers. Ce sont des réseaux qui reposent sur la génération de graphes aléatoires entre les nœuds. Chaque nouveau nœud doit connaître un nœud appartenant au réseau, qui lui sert de bootstrap (ou nœud de démarrage) pour s'insérer dans le réseau. Les requêtes se passent ensuite sous la forme d'inondation (demande à tous les voisins qui demandent à tous leurs voisins, etc.) ou de marche aléatoire (demande à un voisin qui demande lui aussi à un autre voisin, etc.). Ici, nous présentons Gnutella [12] qui a été le premier réseau non structuré proposé, et Freenet [12].

2.1. Gnutella

Gnutella [12] permet de relier les ordinateurs les uns aux autres, il permet le transfert de fichiers d'une machine à une autre de façon plus ou moins anonyme. Chaque machine est à la fois un serveur de fichiers et un client (appelée souvent sous l'acronyme ***servent***). Contrairement à Napster qui était relié à un serveur central unique, il n'existe pas de tel serveur pour Gnutella, chacun fait partie intégrante du réseau et ce réseau se modifie au cours des connexions.

Un nouveau servent(ou nœud) pour rejoindre le réseau se connecte. Il commence d'abord par rechercher tous les nœuds Gnutella du réseau. Pour cela il transmet une trame d'identification (**PING**) à tous ces voisins qui eux-mêmes la transmettront à leurs voisins. Ces envois sont encapsulés dans une trame TCP. Pour borner la recherche, le mécanisme de recherche joue sur le TTL de la trame. A chaque nœud du réseau, le TTL est décrémenté et lorsqu'il devient égal à 0 la retransmission est stoppée.

Ce mécanisme permet d'éviter les boucles dans la transmission des trames d'identification. Lorsqu'une trame est reçue elle est stockée pendant un court laps de temps. Si le nœud reçoit

pendant ce laps de temps une trame identique il la rejette car elle est déjà traitée. Lorsqu'un nœud est identifié, il envoie à l'émetteur une trame de réponse (**PONG**).

Ce système utilise un protocole qui fonctionne au moyen de cinq descripteurs principaux qui permettent la transmission des informations entre les servents (ou nœuds) du réseau, et d'un ensemble de règles qui régissent l'échange de ces descripteurs.

- Ping : utilisé pour trouver les autres nœuds du réseau. Un servent recevant un Ping doit répondre avec un (ou plusieurs) Pong.
- Pong : réponse à un Ping. Le servent répondant par un Pong livre son adresse IP ainsi que des informations sur les données qu'il partage.
- Query : requête visant à trouver un ou plusieurs fichiers vérifiant certains critères.
- QueryHit : réponse à un Query. Donne une liste de fichiers correspondant à la requête, ainsi que l'adresse IP des servents où ces fichiers on été trouvés.
- Push : Mécanisme permettant aux contributeurs situés derrière un Firewall de se raccorder au réseau.

Pour obtenir une ressource, le pair lance une requête vers certains pairs du réseau qui la relaient vers les pairs auxquels ils sont connectés, qui eux même la transmettent. Si l'un des pairs dispose d'une ressource qui pourrait convenir au pair demandeur, il transmet l'information vers ce pair. Ce dernier pourra ainsi ouvrir une connexion directe vers cet ordinateur et obtenir la ressource.

Ce modèle, tout en étant décentralisé, est beaucoup plus robuste qu'un modèle centralisé puisqu'il n'est pas dépendant d'un serveur, qui est le point de défaillance potentiel d'un réseau centralisé. Il tire partie de l'intermittence des connexions des nœuds car si l'un des nœuds se déconnecte du réseau, la requête pourra être poursuivie vers les autres ordinateurs connectés.

Un grand avantage de ce nouveau type de réseaux, est le total anonymat qu'il procure. En effet en évitant de communiquer avec une machine centralisant les demandes et les annuaires, on évite les problèmes de récupération des données utilisateurs.

En revanche, en raison de la façon dont sont transmises les requêtes (**broadcast**), la bande passante nécessaire pour chaque requête croît exponentiellement quand le nombre de pairs croît linéairement. De plus, ce type de mécanisme est très facilement victime d'activités malicieuses. Des membres malintentionnés peuvent envoyer en grande quantité des requêtes erronées qui produisent une lourde charge sur le réseau, réduisant ainsi son efficacité.

2.2. Freenet

Freenet [12] est un réseau P2P adaptatif qui permet aux pairs de stocker et de rechercher des éléments de données, qui sont identifiés par des clés indépendantes de la localisation. C'est un exemple de réseau P2P décentralisé approximativement structuré avec un placement de fichiers basé sur l'anonymat. Afin de protéger les hébergeurs tous les fichiers sont cryptés. Ainsi, le possesseur d'un nœud ne peut pas connaître le contenu des fichiers stockés sur son nœud. Le fichier est crypté grâce à une paire de clés asymétriques (générée aléatoirement) qui caractérisent le propriétaire du fichier. La partie privée de cette clé sert pour signer le fichier (contrôle d'intégrité). La partie publique, concaténée et hachée avec un texte descriptif du fichier donne la clé de recherche.

L'algorithme de routage pour le stockage et la récupération des données est conçu pour ajuster adaptativement les chemins dans le temps et offrir une performance efficace tout en utilisant les connaissances locales. Chaque nœud Freenet gère localement ses propres données qui sont disponibles à la fois en lecture et en écriture. Il dispose également d'une table de routage dynamique contenant les adresses IP d'autres nœuds du système ainsi que les clés de fichiers qu'ils détiennent.

Lorsqu'un utilisateur désire retrouver une donnée, la première chose qu'il doit faire est d'obtenir (sur un site WEB par exemple) ou de calculer la clé binaire correspondant à cette donnée. Ensuite, le nœud de l'utilisateur va créer un message *DataRequest* qui contiendra cette clé, ainsi qu'une valeur *hops to live* qui correspond à un compteur qui est décrémenté à chaque passage dans un nœud (c'est l'équivalent du *Time To Live* de Gnutella). Ce message est alors envoyé à l'un des voisins du nœud. Lorsqu'un nœud reçoit un message *DataRequest*, il consulte ses données locales afin de contrôler s'il est bien le possesseur de la donnée demandée. Si c'est le cas, il va retourner au voisin qui lui a transmis le *DataRequest* la donnée en question à l'aide d'un message *DataReply* (message 10 de la Figure 3). Si ce nœud ne possède pas la donnée, il va chercher dans sa table de routage une similitude (déterminée par la distance lexicographique, la plus courte, de deux clés) avec la clé binaire du fichier contenue dans la requête et il fait suivre le message *DataRequest* au nœud voisin susceptible d'avoir la réponse.

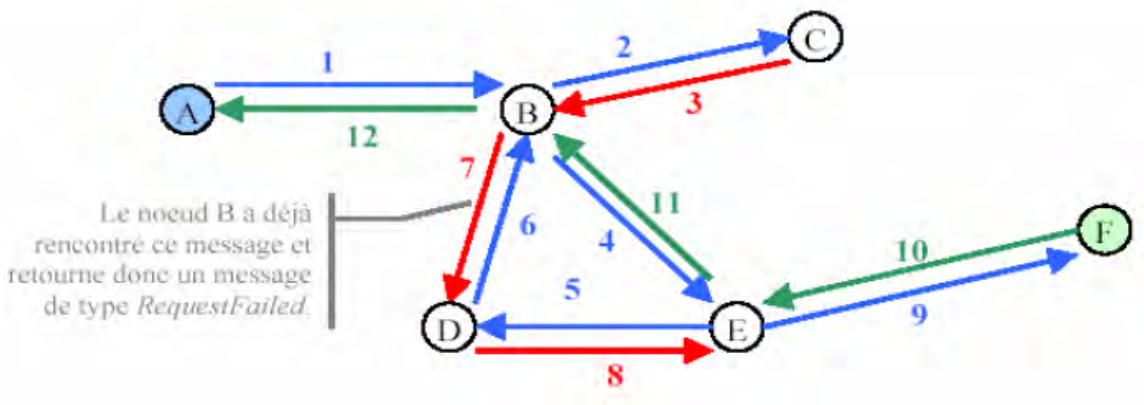

Figure 3 : Recherche par le nœud A d'une donnée que possède le nœud F

les liens bleus correspondent aux messages *DataRequest*,

les rouges aux *RequestFailed*

les verts aux *DataReply*.

Lorsqu'un nœud reçoit un message *DataReply*, il stocke la donnée contenue dans ce message dans son propre cache local avant de faire suivre le message à son prédécesseur, tout comme il ajoutera dans sa table de routage une nouvelle entrée associant la source de la donnée et la clé de la requête.

Un nœud ne peut satisfaire une requête soit parce qu'il ne possède plus de voisins à qui envoyer le message (comme le nœud C dans la Figure 3), soit parce que la valeur du *hops to live* du message a atteint 0, soit encore parce que le nœud a déjà reçu la requête. Lorsqu'un nœud se retrouve dans l'un des deux derniers cas, il répond par un message *RequestFailed* (comme par exemple les messages 3,7 et 8 de la Figure 3). Dans ce cas son prédécesseur enverra donc la requête vers un autre nœud de sa table de routage.

Cependant, il faut remarquer que Freenet a des limites telles que :

- Le temps nécessaire pour trouver un fichier dans Freenet est très honorable. La méthode de routage par clé semble, dans le temps, porter ces fruits en « spécialisant » les nœuds. Mais une fois un fichier trouvé sur un nœud, il doit repasser par tous les nœuds du chemin où le message avait passé. Même si ces transferts devraient être (dans le futur) tunellés c'est-à-dire les nœuds n'attendent pas d'avoir complètement reçu le fichier pour le retransmettre, il n'en demeure pas moins que toutes ces copies ont un coût en temps non négligeables. On peut cependant noter que ce temps de recopie diminuer en fonction de l'intérêt que le fichier suscite.

- Il est (pratiquement) impossible de retrouver une information dans Freenet sans connaître sa clé. Bien que les auteurs évoquent l'improbable possibilité d'une forme d'indexation automatique, il faut donc que celui qui l'a insérée dans le système dispose un moyen pour diffuser cette clé. Les développeurs envisagent la mise en place d'un site Web permettant de retrouver les clés des documents. Mais un tel concept va à l'encontre même d'une décentralisation totale qui fait partie des idées de base qui ont fait naître Freenet. Cependant, il est possible avec la version actuelle de créer un "site web" sur Freenet (donc sans base fixe) pouvant permettre l'indexation du reste. Cela étant dit, il est, pour l'instant, relativement difficile de récupérer des informations depuis Freenet.

3. Les réseaux structurés

Les réseaux P2P structurés reposent sur le principe d'une table de hachage distribuée (Distributed *Hash Table (DHT))*. Une DHT permet de répartir des données sur les nœuds en utilisant une fonction de hachage. Les deux principales fonctions fournies sont *insert(hash, data)* qui insère une donnée dans la DHT et *lookup(hash)* qui retourne la donnée stockée en *hash*.

Tout comme pour les réseaux non structurés, un nœud doit connaître un autre pour s'inscrire dans le réseau. Ensuite, chaque nœud et chaque ressource se voient attribuer un identifiant unique (par exemple, *hash(ip)* pour un nœud, *hash(data)* pour un fichier, . . .), situé dans un espace commun.

Le routage d'un nœud vers une ressource s'effectue par "bonds" successifs dans cette espace, en se rapprochant de l'identifiant de la ressource demandée. Afin de passer à l'échelle, les algorithmes de routage sont généralement en $O(log(N))$ ou $O(1)$, N étant le nombre de nœuds du réseau. Au niveau de la DHT, les requêtes sont limitées aux ressources dont l'identifiant est connu (pas de requêtes complexes comme dans les réseaux non structurés). Il est néanmoins possible de fournir un système de recherche au dessus de la DHT. Dans cette section nous faisons en premier lieu l'étude de CAN[13] et ensuite celle de Chord[14].

3.1. CAN (Content Addressable Network)

CAN[13] est une infrastructure P2P décentralisée distribuée qui fournit la fonctionnalité de table de hachage sur l'évolution de l'Internet. Le modèle d'architecture est un espace

virtuel de coordonnées cartésien multidimensionnel logique. L'espace de coordonnées entier est dynamiquement partitionné entre tous les pairs dans le système de telle sorte que chaque pair possède sa propre et distincte zone dans l'espace total.

Chaque pair maintient les adresses IP de ceux qui ont des zones de coordonnées voisines de sa zone. Cet ensemble de voisins immédiats dans l'espace de coordonnées sert comme une table de routage de coordonnées qui permet un routage efficace entre les points de cet espace.

L'espace virtuel des coordonnées est utilisé pour le stockage de paires (clé, valeur) comme suit: pour stocker une paire (K, V), la clé K est mappée vers un point P dans l'espace des coordonnées en utilisant une fonction de hachage uniforme déterministe. Cependant, pour rechercher une entrée correspondante à la clé K, n'importe quel pair peut appliquer la même fonction de hachage déterministe pour mapper K vers le point P et ensuite récupérer la valeur correspondante V du point P. Si le pair demandant ou ses voisins immédiats ne sont pas propriétaire du point P, la requête doit être acheminée par le biais de l'infrastructure de CAN jusqu'à ce qu'elle atteigne le pair où se trouve P.

Un nouveau pair qui joint le réseau doit avoir sa propre portion d'espace de coordonnées. Cette zone peut être obtenue en divisant celle d'un pair existant en deux, retenir une moitié pour le pair et alloué l'autre moitié au nouveau pair. Ce pair à qui la zone doit être divisée est obtenu à l'aide des pairs ***bootstrap***. Un pair ***bootstrap*** est un nœud qui maintient une liste partielle des pairs de CAN. En effet, le système possède un nom de domaine DNS qui est résolu par l'adresse IP d'un ou plusieurs pairs ***bootstrap***. Le nouveau pair regarde le nom de domaine dans le DNS pour trouver l'adresse IP d'un pair ***bootstrap*** ; ce dernier lui fournit les adresses IP de quelques pairs choisis aléatoirement dans le système. Il choisit alors aléatoirement un point P dans l'espace et envoie une requête de jointure destinée pour ce point. Chaque pair du système utilise le mécanisme de routage pour transmettre le message jusqu'à ce qu'il atteigne le pair où se trouve le point P.

Quand un pair quitte le système, la zone qu'il occupait et la base de données (clé, valeur) associée sont remises explicitement à un de ses voisins. Si la zone d'un des voisins peut être fusionnée avec la zone du pair partant pour produire une zone unique valide, alors la fusion est faite. Sinon la zone est passée à un voisin à qui la zone actuelle est la plus petite, et que le pair manipulera temporairement toutes les deux zones. Le pair met à jour son ensemble de voisins pour éliminer les pairs qui ne sont plus ses voisins. Ainsi, chaque pair dans le système envoie des mises à jour pour assurer que tous ses voisins apprennent le changement et mettent à jour leur ensemble de voisins.

Cependant, le nombre de voisins qu'un pair maintient dépend seulement de la dimension de l'espace de coordonnées et il est indépendant du nombre total de pairs dans le système. C'est ainsi que pour un espace d-dimensionnel partitionné en n zones égales, la longueur moyenne du chemin de routage est $(d/4)*(n^{1/d})$ sauts et chaque pair maintient une liste de 2*d voisins.

Une amélioration de l'algorithme de CAN est effectuée en maintenant plusieurs espaces de coordonnées indépendants avec chaque pair dans le système assigné à une zone différente dans chaque espace de coordonnées, appelé *réalité*. Ainsi, pour un CAN avec ***r*** *réalités*, un pair simple est assigné à ***r*** zones de coordonnées, une dans chaque *réalité* disponible et ce pair possède ***r*** ensembles de voisins. Les tables de hachage sont répliquées sur chaque *réalité*, ainsi augmentant la disponibilité des données.

Pour plus d'amélioration de la disponibilité des données, CAN utilise ***k*** différentes fonctions de hachage pour mapper une clé donnée vers ***k*** points dans l'espace de coordonnées. Cela se traduit dans la réplication d'une paire (clé, valeur) unique sur ***k*** pairs distincts dans le système. Une paire est alors non disponible seulement quand tous les ***k*** répliques sont simultanément non disponibles. Ainsi, des requêtes pour une entrée particulière de la table de hachage sont transmises à tous les ***k*** pairs en parallèle de façon à réduire la latence moyenne d'une requête.

3.2. Chord

Chord[14] utilise un hachage pour affecter des clés à chaque pair. Il tend toujours à équilibrer la charge dans le système puisque chaque pair reçoit le même nombre de clés, et il y'a peu de mouvements des clés quand des pairs quittent et joignent le système. SHA-1 est utilisé comme fonction de hachage pour affecter un identifiant unique de *m*-bits pour chaque pair et un identifiant unique aux clés. Un identifiant d'un pair est choisi en hachant l'adresse IP du pair, alors qu'un identifiant de clé est produit en hachant la clé. Les identifiants sont classés sur un cercle d'identifiant modulo 2^m appelé *anneau Chord*. La clé K est attribuée au pair dont l'identifiant est supérieur ou égal à K dans l'espace d'identifiants. Ce pair est appelé le pair successeur de la clé K, dénoté par successeur(K). Les identifiants étant représentés sur un cercle de chiffres de 0 à 2^{m-1}, alors successeur(K) est le premier pair dans le sens des aiguilles d'une montre à partir de K.

Le nombre de bits de l'espace clé/NodeID (identifiant de nœud) étant égale à *m*, chaque pair maintient une table de routage avec un maximum de *m* entrées, appelée finger table ou

table d'indexe. La i^{ieme} entrée de la table d'un pair ***n*** contient l'identité du premier pair **s** qui succède à ***n*** par au moins 2^{i-1} sur le cercle d'identifiant, c'est-à-dire s = successeur $(n+2^{i-1})$, où $1 \leq i \leq m$. Le pair ***s*** est le i^{ieme} indexe du pair ***n*** (n.indexe [i]). Une entrée de la table d'indexe comprend à la fois l'identifiant Chord et l'adresse IP du pair.

La Figure 4 montre un anneau Chord (m = 6) qui a dix pairs et qui stocke cinq clés. Le successeur de l'identifiant 10 est le pair 14, donc la clé 10 sera localisée au NodeID 14. La Figure montre aussi la table d'indexe du pair 8, et la première entrée d'indexe pour ce pair pointe sur le pair 14 puisque successeur$(8 +2^0)$ modulo 26 = 9. De même, le dernier indexe du pair 8 pointe sur le pair 42, c'est-à-dire le premier p air qui a comme identifiant 40 (successeur$(8+2^5)$ modulo 26 = 40).

De cette façon, les pairs stockent l'information sur seulement un petit nombre de pairs, et ont plus de connaissances sur leurs pairs successeurs dans l'anneau que sur les autres pairs. Par contre, une table d'indexe d'un pair ne contient pas une information suffisante pour déterminer directement le successeur d'une clé K arbitraire.

Pour maintenir le mappage du hachage cohérent quand un pair ***n*** rejoint le réseau, certaines clés précédemment affectées à son successeur lui sont maintenant affectées et les pointeurs, successeurs de certains pairs, doivent être changés. Si toutefois un pair joint le système avec l'identifiant 26, il stockerait la clé avec l'identification 24 du pair qui a l'identifiant 32. Il est d'une importance capitale que les pointeurs successeurs soient à jour à tout moment puisque l'exactitude des recherches n'est pas garantie autrement. C'est pour cela que Chord utilise un protocole de stabilisation s'exécutant périodiquement en arrière-plan pour mettre à jour les pointeurs successeurs et les entrées dans les tables d'indexe. Par contre lorsque le pair **n** quitte le système Chord, toutes les clés qui lui sont assignées sont réaffectées à son successeur.

Lors des requêtes de recherche une correspondance entre la clé et le NodeID est effectuée. C'est ainsi qu'un identifiant donné pourrait passer autour de l'anneau par l'intermédiaire de ses pointeurs successeurs jusqu'à ce qu'il rencontre une paire de nœud qui inclut l'identifiant désiré ; le second nœud dans la paire est le nœud défini par la requête. Par exemple à la Figure 4 le pair 8 effectue une recherche pour la clé 54. C'est ainsi qu'il invoque l'opération find_successor pour cette clé, qui retourne finalement le successeur de cette clé, c'est-à-dire le pair 56.

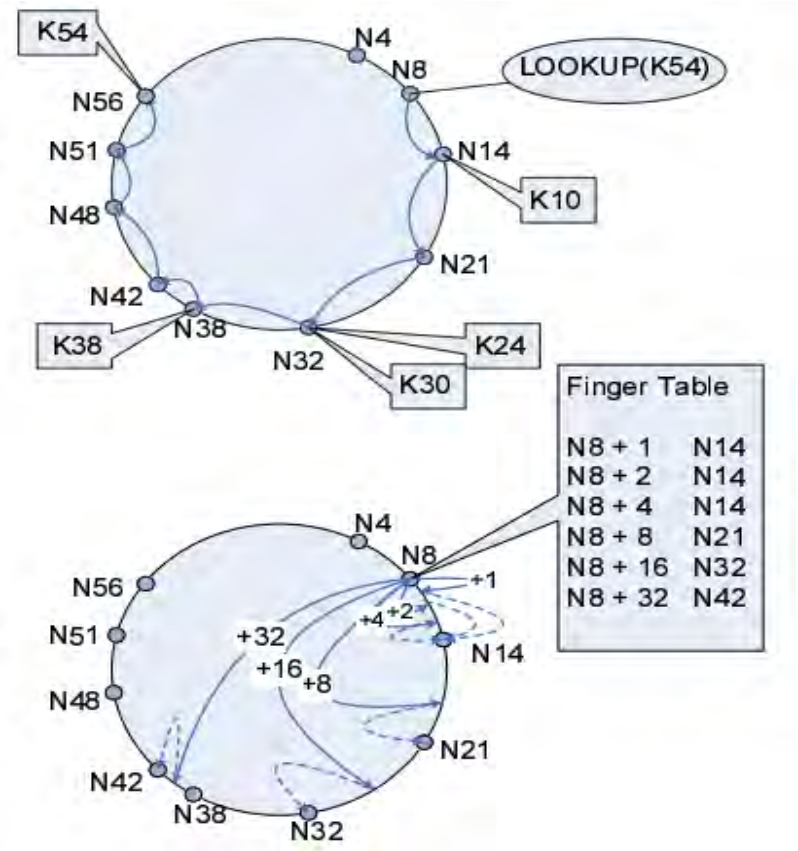

Figure 4 : Anneau Chord avec un cercle d'identifiant de dix pairs et cinq clés

L'exactitude du protocole Chord repose sur le fait que chaque pair a une connaissance parfaite de ses successeurs. Lorsque des pairs tombent en panne, il est possible qu'un pair ne connaisse pas son nouveau successeur, et qu'il n'ait aucune chance d'apprendre cette défaillance. Ainsi, pour éviter une telle situation, les pairs maintiennent une liste de successeurs de taille ***r***, qui contient les premiers ***r*** successeurs d'un pair. Lorsque le pair successeur ne répond pas, le pair contacte simplement le prochain pair sur sa liste de successeurs. En supposant que les défaillances d'un pair se produisent avec une probabilité p, la probabilité que tous les pairs référencés sur la liste de successeurs soient en panne est p^r. Ainsi, augmenter ***r*** permettra de rendre le système Chord plus robuste.

II. Les Grilles Informatiques

L'augmentation perpétuelle de la quantité des données à traiter, due à la convergence des technologies de l'information, tend à rendre le traitement sur des ordinateurs isolés obsolète. De ce fait, il se manifeste un besoin de pouvoir accéder à tout un ensemble d'informations à tout moment et de n'importe où ; ce qui pourrait poser en toute évidence quelques problèmes :

la localisation des données, le temps d'accès aux ressources et le temps de traitement associé. La solution serait donc d'agréger les capacités (de stockage et de calcul) de plusieurs ordinateurs de conception « ordinaire » pour réaliser un seul et même service. Cet agrégat est appelé ***grille informatique***. Cependant, Il existe différents types de grilles, chacun permettant de répondre à des besoins différents.

1. Définitions des grilles informatiques

Une grille est définie dans [5] comme étant une ***infrastructure matérielle et logicielle qui fournit un accès fiable, cohérent, accessible de n'importe où, peu cher et avec des capacités de calcul haut de gamme.*** Une classification souvent employée permet de distinguer deux catégories de grilles : les grilles de calcul (mise en commun de machines dédiées aux calculs) et les grilles de données (mise en commun de machines dédiées au stockage d'énormes quantités de données, telles que des bases de connaissances dans un domaine).

1.1. Les grilles de calcul

L'objectif à t erme des grilles de calcul est de fournir un ensemble de ressources informatiques virtuellement infini pour le calcul de manière transparente pour les utilisateurs. La localisation de ces ressources informatiques est géographiquement distribuée à travers une ou plusieurs nations.

Une grille de calcul peut être définie comme un système distribué constitué de l'agrégation de ressources réparties sur différents sites et mises à disposition par plusieurs organisations différentes. Chaque site dispose d'un grand nombre de ressources.

Un site est un lieu géographique regroupant un ensemble de ressources informatiques administrées de manière autonome et uniforme. Les sites d'une grille de calcul peuvent être organisés de manières différentes, par exemple, sous la forme d'une (ou d'une collection de) grappe(s) de machines ou de supercalculateurs, voir un mélange des deux. Une grappe de machines (en anglais cluster) est un regroupement de machines indépendantes interconnectées par un même équipement réseau, localisées dans un même site et disposant d'au moins une machine spécifique de gestion appelée machine frontale. Généralement, les machines qui composent une grappe ont des caractéristiques matérielles homogènes (processeurs, mémoire vive, technologies d'interconnexion réseau, etc.).

Nous présentons dans cette section les caractéristiques physiques de ces infrastructures, à travers deux exemples représentatifs de grilles de calcul. Nous décrivons tout d'abord la grille expérimentale Grid'5000 [6], puis la grille de calcul TeraGrid [7].

- **Grid'5000 [6]:** Le projet Grid'5000 consiste en la construction d'une grille expérimentale. En effet, elle diffère des autres grilles de calcul par sa capacité d'être hautement reconfigurable et contrôlable. Elle permet à chaque utilisateur de déployer son propre système d'exploitation. À terme, Grid'5000 devrait être constituée de 5000 processeurs (actuellement 2600 sur 1244 machines) répartis sur neuf sites français : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. L'interconnexion entres les différents sites est assurée par le réseau Renater qui fournit un débit de 10 Gb / s. La latence entre les machines de différents sites varie de 4 ms à 29 ms, selon les paires de sites considérées. La Figure 5 présente l'interconnexion des différents sites de Grid'5000. Les machines des différents sites sont généralement équipées de biprocesseurs Opteron d'AMD. Cette grille est toutefois hétérogène : elle comporte également des machines équipées de processeurs Intel ou PowerPC. Les machines à l'intérieur d'un site sont interconnectées par des liaisons Gigabit Ethernet.

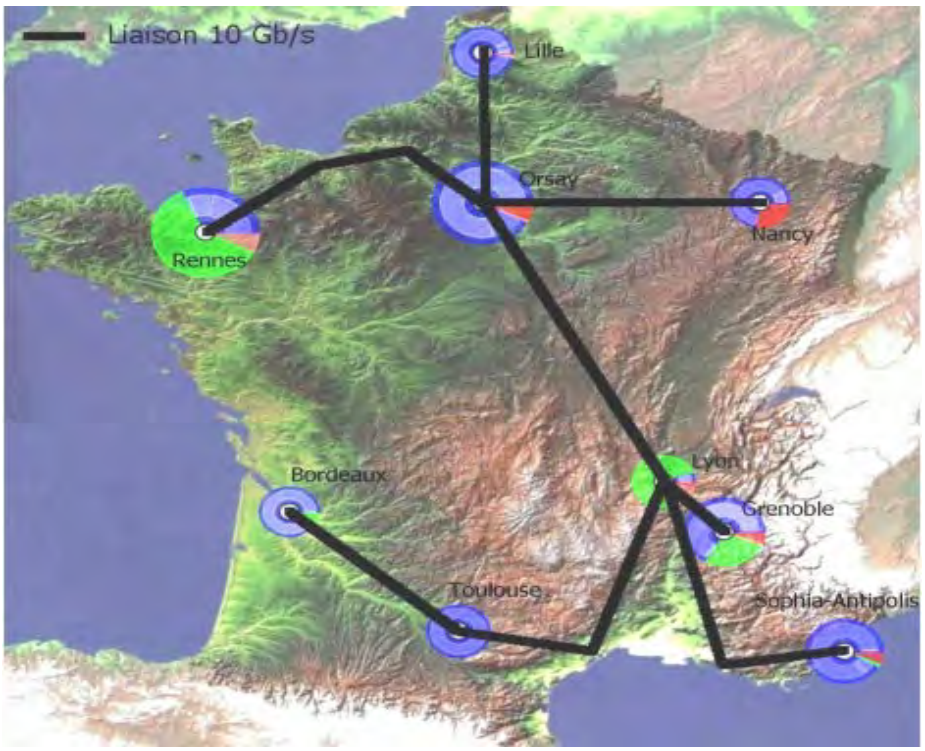

Figure 5 : L'interconnexion des différents sites de la grille expérimentale Grid'5000

- **TeraGrid [7] :** Le projet TeraGrid est un projet américain visant à mettre en place la plus grande et la plus puissante grille de calcul scientifique au monde. Environ 22 000 processeurs sont répartis sur neuf sites : Austin, Bloomington, Boulder, Chicago, Oark

Ridge, Pittsburgh, San Diego, Urbana et West Lafayette. Un réseau de fibre optique assure la liaison entre les différents sites et la Figure 6 en présente l'interconnexion. Ce réseau peut atteindre à certains endroits un débit de 30 Gb / s. La latence entre les machines de différents sites varie de 3 ms à 80 ms, selon les paires de sites considérées. Les machines constituant les différents sites sont très variées : Intel Itanium, Power4 IBM, SGI Altix, etc. Les systèmes d'exploitation sont également variés : Linux, Unix, etc.

Figure 6 : L'interconnexion des différents sites de la grille de calcul TeraGrid

1.2. Les grilles de données

Certaines applications, comme la physique des particules, peuvent générer des quantités astronomiques de données pouvant atteindre plusieurs téraoctets. Il est impossible de stocker ces dernières sur une seule et même machine. On a donc souvent recours à une grille de données.

Une grille de données ne permet pas seulement de découper des fichiers et de les ranger dans plusieurs machines. Elle doit mettre en place des mécanismes de recherche, d'indexation, d'intégrité pour assurer un a ccès fiable et permanent. Il faut aussi que la répartition des données soit transparente pour l'utilisateur final. En effet, il est impossible de savoir précisément où est stockée chaque donnée. Un fichier peut être découpé en plusieurs

morceaux et réparti sur plusieurs machines. Il peut être répliqué sur plusieurs entités du réseau pour permettre une meilleure disponibilité. C'est ainsi qu'il faudra avoir un mécanisme qui répartit la charge sur chaque serveur (machine du réseau).

Pour montrer les caractéristiques de ces types de grilles nous prenons comme exemple Datagrid[10]

Datagrid est une grille européenne qui a rassemblé 21 partenaires scientifiques et industriels européens, regroupant jusqu'à 15 000 ordinateurs de type standard et plus de 15 téraoctets de stockage répartis sur 25 sites en Europe. Le tout est mis au service de 500 scientifiques et de trois domaines d'applications majeurs : la physique des hautes énergies, les applications biomédicales et l'observation de la terre.

2. Infrastructures logicielles des grilles informatiques

La section précédente a montré que les grilles informatiques sont hétérogènes et complexes. Or, l'objectif des grilles est de fournir les ressources de manière transparente et la plus simple possible. Atteindre un tel objectif nécessite l'utilisation de logiciels spécifiques, appelés intergiciels pour grilles, pour leur gestion.

Un intergiciel (en anglais middleware) est un logiciel qui se trouve entre l'application de l'utilisateur et les ressources informatiques matérielles et logicielles sous-jacentes, afin d'aider l'application à utiliser ces ressources.

Par ailleurs, on di stingue principalement deux types d'intergiciels pour grilles : les intergiciels de gestion et d'accès aux ressources et les intergiciels pour la programmation d'applications.

- **Intergiciels de gestion et d'accès aux ressources** : le rôle de ces intergiciels est de donner une vision globale et uniforme des ressources partagées au sein d'une grille de calcul. Ainsi, ils s'attachent à la gestion de plusieurs aspects différents qui sont décrits ci-dessous.
    1. l'accès aux différentes ressources informatiques, c'est-à-dire l'allocation des ressources nécessaires pour les applications ainsi que le lancement des applications sur ces ressources.
    2. le placement des données nécessaires en entrée aux applications, ainsi que la récupération des données produites par celles-ci à la fin de leur exécution.
    3. la mise en place de services d'information permettant de connaître l'état des ressources mises à disposition par une grille

4. la mise en place des politiques de sécurité définies par les administrateurs, par exemple la méthode choisie pour l'authentification des utilisateurs

L'intergiciel grille le plus utilisé pour la gestion et l'accès aux ressources d'une grille est Globus [8] que nous allons présenter ci-dessous. Nous allons aussi faire une présentation de l'intergiciel Legion[9].

Globus est une collection d'outils génériques (bibliothèques et outils en ligne de commande) pour la gestion et l'accès aux ressources d'une grille. Il est principalement développé à Argonne National Laboratory, sous la direction d'Ian Foster. Le module de gestion des ressources de Globus s'appelle GRAM (pour Globus Resource Allocation Manager). Il permet de soumettre des tâches sur des ressources distantes, ainsi que de contrôler ces tâches. Pour ce faire, un script RSL (Resources Specification Language) doit être écrit par les utilisateurs afin de spécifier les fichiers exécutables à l ancer, les données à utiliser, etc. À partir de ce fichier, Globus génère l'ensemble des requêtes nécessaires aux multiples gestionnaires de ressources (potentiellement de types différents) des différentes grappes de machines de la grille utilisée. Une politique d'allocation simultanée d'ensembles de ressources est disponible dans Globus, sous le nom de DUROC (Dynamically Updated Request Online Co-allocator). Le service d'information de Globus est appelé MDS (Monitoring & Discovery System). Il permet par exemple d'obtenir des informations sur l'état des ressources d'une grille. Enfin, le module GSI (Grid Security Infrastructure) permet l'authentification des utilisateurs grâce à des certificats.

Legion a pour objectif de donner une vision d'un unique supercalculateur virtuel par agrégation des ressources de la grille. Pour atteindre ce but, Legion utilise une approche dans laquelle toutes les ressources (matérielles, logicielles, etc.) sont des objets. Les objets sont décrits par un langage de description d'interface, et les communications entre les différentes instances des objets utilisent le modèle de programmation par appel de procédures (Remote Procedure Call, RPC). Ce projet vise la gestion de centaines de machines et de millions d'objets. Les différentes interfaces des objets définis par Legion sont spécialisables, héritables, etc. afin de rendre le système extensible.

- **Intergiciels pour la programmation d'applications :** l'objectif des intergiciels pour la programmation d'applications est d'offrir des paradigmes de programmation, parfois spécifiques, mais de haut niveau, aux développeurs d'applications. Ainsi, ceux-ci peuvent se concentrer sur la logique de leur application (appelé *code métier*), le cycle de vie de l'application étant entièrement géré par l'intergiciel utilisé. Contrairement

aux intergiciels précédents, ceux-ci n'obligent pas le développeur à connaître parfaitement les différents éléments qui composent une grille de calcul et à interagir avec eux. De nombreux intergiciels pour la programmation d'applications existent, parmi lesquels nous pouvons citer les suivants :

- CORBA et ses utilisations pour bâtir des intergiciels de plus haut niveau pour la programmation d'applications. L'intergiciel Grid-RPC DIET en est un exemple typique ainsi que les implémentations du modèle de composant CCM, telles que OpenCCM et MicoCCM par exemple.
- Les intergiciels implémentant des modèles composant tels que Fracta, Iceni, Grid.it et Darwin par exemple;

- **Systèmes d'exploitation pour grilles de calcul** : l'objectif des intergiciels de gestion et d'accès aux ressources étant de fournir l'illusion d'un unique supercalculateur, plusieurs projets visent à modifier directement les systèmes d'exploitation des machines. L'objectif est alors de construire ce qui peut s'appeler un *système d'exploitation pour grilles de calcul*, en y incluant des fonctionnalités jusque-là incluses dans les intergiciels pour grilles. Par exemple, les mécanismes de découverte et d'allocation de ressources ou de tolérance aux pannes, tels que les points de reprise, peuvent être intégrés à un système d'exploitation pour grilles.

  L'avantage de ce type de système est alors de fournir ces fonctionnalités de manière transparente pour les applications. Les applications ne doivent pas être portées d'un intergiciel à un autre puisque les fonctionnalités sont directement incluses dans le système d'exploitation et sont donc fournies par celui-ci. En outre, les systèmes d'exploitation pour grilles permettent de factoriser des mécanismes en partie similaires, redéveloppés à chaque fois par les différents intergiciels pour grilles qui existent.

Remarque :

- Dans la littérature il existe un troisième type de grille qui est souvent utilisé ; les grilles de vol de cycles (en anglais desktop grids ou cpu scavenging grids) mais elles sont généralement incluses dans les grilles de calcul. Elles consistent en la mise en commun pendant la nuit, de machines de bureau utilisées dans la journée. La récupération de cycles de calcul permet d'utiliser ces machines pour exécuter des tâches diverses.

- La distinction entre les grilles de calcul et les grilles de données est floue. Cependant, il faut noter que les grandes bases de connaissances, issues des grilles de données, sont de plus en plus souvent accédées par des applications s'exécutant sur des grilles de calcul.

Conclusion

Dans ce chapitre nous avons fait une large étude des systèmes à large échelle notamment les systèmes P2P et les grilles informatiques. Nous avons pu voir leur fonctionnement, leurs avantages dans la gestion des données à large échelle en prenant certains exemples.

Vu leurs importances, le chapitre suivant permettra de définir des mécanismes d'exploitation de ces d onnées qui y sont stockées. Cela consistera surtout à définir des algorithmes de routage des différentes opérations de lecture et d'écriture.

Chapitre 2

Mécanismes de traitement des transactions dans les systèmes à large échelle

Dans ce chapitre nous allons définir comment les transactions seront acheminées dans les systèmes à large échelle qui ont fait l'objet du chapitre 1 de notre stage. L'objectif principal vise à mettre en place des algorithmes efficaces pour le traitement des transactions dans ces systèmes.

Dans un premier temps nous faisons un état de l'art sur le traitement des transactions à large échelle avant de présenter l'architecture globale de notre système inspirée de celle présentée dans [2].

En deuxième lieu nous donnerons des définitions sur les différents concepts qui sont pris en compte pour le traitement des transactions avant de terminer le chapitre avec notre algorithme de routage.

I. Etat de l'art sur le traitement des transactions à large échelle

Les systèmes à large échelle comme les systèmes P2P fournissent l'accès à des ressources massives de stockage et de traitement. Bien que les premiers systèmes proposés furent principalement destinés au partage d'informations, ils sont maintenant considérés comme une solution viable pour les systèmes à forte charge transactionnelle tel que les systèmes de réservation et le commerce en ligne. Pour cela, les données sont répliquées dans le système dans l'optique d'assurer leur disponibilité et de permettre une rapide exécution des transactions grâce au parallélisme. Cependant, garantir à la fois la cohérence et la rapidité d'accès aux données sur de telles architectures pose des problèmes à plusieurs niveaux. En particulier, le contrôle centralisé est prohibé à cause de sa faible disponibilité et de la

congestion que cela engendre à grande échelle. Des solutions décentralisées sont proposées comme celles présentées dans [2] et [15].

Dans [2], les auteurs ont proposé une solution pour la gestion décentralisée du routage des transactions dans un réseau large échelle. Pour assurer la disponibilité, les données sont répliquées et le système de réplication multi-maîtres (ou réplication partout) est utilisé. Chaque nœud peut être mis à jour par n'importe quelle transaction entrante et il est appelé dans ce cas nœud initial de la transaction. Les autres nœuds sont rafraichis plutard en propageant les transactions à travers des transactions de rafraichissement.

Chaque transaction (ou requête simple) lit un ensemble de relations et chaque mise à jour écrit sur un ensemble de relations. Les transactions doivent accéder à des données fraîches, et ceci est contrôlé par les applications. Pour ce faire, les applications peuvent associées une obsolescence tolérée aux requêtes. L'obsolescence tolérée reflète le niveau de fraîcheur qu'une requête requiert pour être exécutée sur un nœud donné.

Cependant, la cohérence mutuelle peut être comprise à cause des mises à jour concurrentes s'exécutant sur différents nœuds. Pour résoudre ce problème, les transactions de mise à jour sont exécutées sur les nœuds de bases de données dans des ordres compatibles, produisant ainsi mutuellement des états cohérents sur toutes les répliques des bases de données. Les requêtes sont envoyées sur tout nœud qui est assez frais mais respectant les exigences de la requête. Afin d'assurer la cohérence globale, ils maintiennent dans leur annuaire un graphe, appelé graphe d'ordre de précédence global. Ce graphe garde les traces des dépendances conflictuelles sur les transactions actives, i.e. les transactions en cours d'exécution dans le système et qui ne sont pas encore validées. Il est basé sur la notion de conflit potentiel : une transaction entrante est en conflit potentiel avec une transaction en cours d'exécution si elles accèdent potentiellement au moins sur une relation en même temps, et au moins une de ces transactions effectue une écriture sur la relation.

Pour effectuer l'acheminement des transactions, ils utilisent une stratégie basée sur le coût et qui utilise une synchronisation différée, prenant ainsi en compte le coût de rafraichissement du nœud avant de lui envoyer une transaction.

Cette approche offre une forte disponibilité des données puisqu'elles sont répliquées sur les différents nœuds. Cependant, pour assurer la cohérence mutuelle des répliques des bases de données, elle fournit une sérialisabilité globale, mais puisqu'elle se base sur les ensembles potentiels de lecture et d'écriture, il peut inutilement réduire le parallélisme. De plus l'algorithme qui est utilisé s'appuie sur une fonction qui évalue d'abord le coût global

d'exécution nécessaire pour une transaction avant qu'elle ne soit envoyée au nœud qui aura donné le moindre coût. Aussi, dans l'évaluation de celui-ci, il est supposé que le temps moyen d'exécution d'une transaction est le même sur toutes les répliques.

[15] présente un système dénommé ***Ganimed*** qui est un intergiciel basé système capable d'offrir à la fois la scalabilité et la cohérence sans avoir à partitionner les données ou à déclarer/extraire les éléments accédés par les transactions. ***Ganimed*** n'impose aucunes restrictions sur les requêtes envoyées. Le composant principal de ***Ganimed*** est un léger planificateur qui achemine les transactions vers un ensemble de répliques en utilisant RSI-PC, un nouvel algorithme de planification. L'idée clef de RSI-PC est la séparation des transactions de mise à jour et des requêtes à lecture seule. Un planificateur RSI-PC est responsable pour un ensemble n de répliques basé sur l'isolation par cliché. Une des répliques est maître, les n-1 autres répliques sont des esclaves. Le planificateur fait une distinction claire entre les transactions à lecture seule et les transactions de mise à jour.

Les instructions de toute transaction de mise à jour arrivante sont directement envoyées vers un maître, elles ne sont jamais différées. Le planificateur prend une information de l'ordre dans lequel les transactions de mise à jour seront validées sur le maître. Après une validation réussie d'une transaction de mise à jour, le planificateur assure que l'ensemble des écritures correspondant soit envoyé sur les n-1 esclaves et que chaque esclave applique les ensembles d'écriture des différentes transactions dans le même ordre correspondant à la validation effectue sur la réplique maître. Le planificateur utilise aussi la notification du numéro de version globale de la base de données. Lorsqu'une transaction de mise à jour est validée sur un maître, le numéro de version globale de la base de données est incrémenté par un et le client reçoit une notification de la validation. Les ensembles d'écriture sont étiquetés par le numéro de version qui était créé par la transaction de mise à jour correspondante.

Les transactions à lecture seule peuvent être traitées par n'importe quelle réplique en mode sérializable. Le planificateur est libre de décider sur quelle réplique à exécuter cette transaction. Si sur une réplique choisie la dernière version globale de la base de données produite n'est pas encore disponible, le planificateur peut retarder la création du cliché à lecture seule jusqu'à ce que tous les ensembles d'écriture nécessaires soient appliquées sur la réplique. Pour les clients qui n'acceptent aucunes attentes pour les transactions à lecture seule, il y'a deux choix : soient leurs requêtes sont envoyées vers la réplique maître réduisant ainsi la capacité disponible pour les transactions de mise à jour, ou le client peut donner un seuil de fraicheur. Un seuil de fraîcheur est par exemple un âge maximum du cliché requis ou la

condition pour que le client voie ses propres mises à jour. Le planificateur peut alors utiliser ce seuil pour choisir une réplique. Une fois que le planificateur ait choisi une réplique et que la réplique ait créé le cliché, toutes les opérations consécutives d'une transaction à lecture seule seront exécutées en utilisant ce cliché.

Du à sa s implicité, il n'y a aucun risque du planificateur RSI-PC à devenir le goulot d'étranglement dans le système. A la différence des autres programmes basés middleware, cet algorithme de planification n'invoque aucun parsing des instructions SQL ou de s opérations de contrôle de concurrence. En plus, aucun verrouillage au niveau des lignes ou des tables n'est fait au niveau du planificateur. La détection des conflits, qui par définition de l'isolation par cliché peut seulement se p roduire durant les mises à jour, est laissée à l'isolation par cliché de la base de données en cours d'exécution sur la réplique maître. En outre, RSI-PC ne met aucunes assomptions sur la charge transactionnelle, le partitionnement des données, l'organisation du schéma, ou les requêtes répondues ou non répondues.

Puisque seule une petite quantité d'informations d'état peut être gardée par le planificateur RSI-PC, il est même possible de construire des planificateurs travaillant en parallèle. Ceci aide à fournir une tolérance aux fautes. Par rapport aux systèmes traditionnels, où chaque réplique a son propre planificateur qui est au courant de l'état global, l'échange d'information d'état entre un petit nombre de planificateurs RSI-PC peut s'effectuer très efficacement. Même dans le cas où tous les planificateurs tombent en panne, il est possible de reconstruire l'état de la base de données entière : un planificateur de remplacement peut être utilisé et son état initialisé en inspectant toutes les répliques disponibles. Dans le cas d'une panne des répliques esclaves, le planificateur les ignore simplement jusqu'à ce qu'elles soient réparées par un administrateur. Cependant, dans le cas d'une défaillance d'un maître, les choses sont un peu plus compliquées. En élisant seulement un nouveau maître, le problème est qu'à moitié résolu. Le planificateur doit aussi s'assurer qu'aucune mise à jour des transactions validées ne soit perdue, garantissant ainsi la d urabilité de l'ACID. Cet objectif peut être atteint en envoyant seulement les notifications de validation aux clients après que l'ensemble des transactions de mise à jour ait été appliqué avec succès.

Ce système offre une disponibilité et une cohérence des données en utilisant la réplication. Mais le type mono maître utilisé pose un problème sur le temps de réponse des transactions si la charge transactionnelle est très grande. En plus le planificateur constitue un goulot d'étranglement pour le système ce qui constituera un inconvénient majeur pour son passage à l'échelle.

II. Architecture globale de notre système

L'architecture globale de notre système est représentée par la Figure 7. Il se présente avec un intergiciel (ou middleware en anglais) qui s'intercale entre les CNs (Client Nodes) et les DNs (Data Nodes). L'intergiciel est essentiellement constitué de TMs (Transaction Manager) et du Shared Directory (ou annuaire partagé).

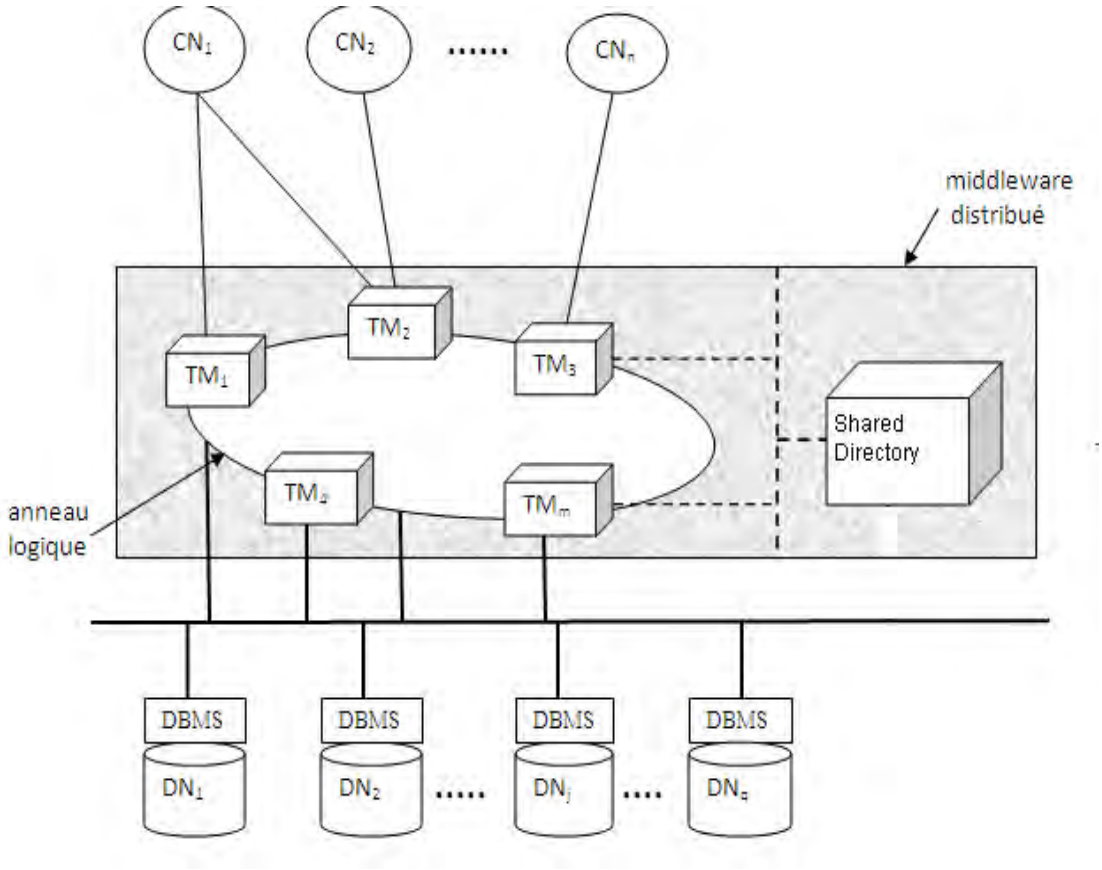

Figure 7 : Architecture globale du système

- **CN (Client Nodes) :** ils envoient les transactions à un Transaction Manager (TM). Un Client Node est connecté à un ou plusieurs Transaction Managers. Un Transaction Manager est choisi selon la méthode du tourniquet(ou Round Robin en anglais). Durant le routage, un nœud client affecte à chacune de ses transactions son identifiant (c'est-à-dire son adresse IP et son port). Ceci dans le but de permettre à un Data Node d'envoyer directement les résultats d'une transaction au CN qui l'avait soumise.
- **TM (Transaction Manager):** ils sont responsables du routage des transactions vers les nœuds de données. Les Transaction Managers se servent des métadonnées stockées dans le Shared Directory (ou annuaire partagé) pour effectuer un acheminement efficace. Ils sont organisés autour d'un anneau logique dans le but de faciliter la

détection collaborative des pannes qui n'est pas un objectif de ce stage. Un TM a aussi la possibilité d'envoyer des messages aux CNs dont il est responsable s'il se s ent saturé de transactions pour que ces derniers puissent choisir un autre TM dont il a la connaissance.

- **DN (Data Nodes) :** chaque Data Node utilise un Système de Gestion de Base de Données local pour stocker les données et exécuter les transactions venant des Transaction Managers. Après chaque exécution d'une transaction, les résultats sont retournés directement aux nœuds clients, et le DN informe aussi le Gestionnaire de Transaction ayant envoyé la transaction de la fin de l'exécution ceci dans le but de maintenir la cohérence de la base de données.
- **Shared Directory:** il contient les informations détaillées des transactions traitées sur les nœuds de données, c'est-à-dire pour chaque relation et pour chaque nœud de données, la liste des transactions exécutées et celle des transactions en cours d'exécution. Ces informations sont utilisées pour calculer la fraîcheur d'un nœud de données mais aussi pour déterminer les transactions de synchronisation si le nœud de donner demande à être rafraichir.

Remarque :

- L'objectif principal de notre étude étant le routage des transactions, nous faisons une abstraction de toutes défaillances probables dans le système.

III. Mécanismes de traitement des transactions

Face aux inconvénients des solutions décentralisées proposées qui ne prennent pas en compte les différentes caractéristiques des systèmes à large échelle qui sont très dynamiques et très hétérogènes, le besoin d'avoir des mécanismes de traitement les prenant en compte devient une nécessité.

1. Définition des transactions

Les transactions des Client Nodes sont envoyées au Transaction Manager qui doit les acheminer aux Data Nodes.

Chaque transaction arrive dans le système à une date donnée. Cette date servira d'estampille. Ainsi T_k sera considérée comme étant la transaction d'estampille k.

L'approche de routage des transactions adoptée dans notre système est l'approche « pull »[4] où ce sont les Data Nodes qui signalent au TM qu'ils sont disponibles pour exécuter une transaction. Dès qu'un Data Node déclare qu'il est disponible, le Transaction Manager doit lui envoyer immédiatement des transactions. On suppose qu'après déclaration de disponibilité, le DN assume la responsabilité d'exécuter toutes les requêtes qu'on lui envoie c'est-à-dire le DN va rester sur le réseau pendant un bon moment.

Pour envoyer une transaction, le TM se basera non seulement sur la *fraîcheur* des Data Nodes ((a) de la section 2) mais aussi sur *l'obsolescence tolérée* ((b) de la section 2) des transactions de sa file d'attente.

Avec la réplication symétrique asynchrone [3] utilisée dans notre système, nous pouvons distinguer entre autre trois catégories de transactions :

- les transactions de mises à jour qui sont composées d'une ou de plusieurs instructions SQL et qui mettent à jour la base de données.
- les requêtes étant des transactions à lecture seule et qui peuvent accepter des mises à jour manquantes. Dans ce cas les nœuds de données (DN) n'ont pas besoin d'être rafraîchies tout le temps.
- les transactions de rafraîchissement qui sont utilisées pour propager les transactions de mise à jour vers les autres répliques pour le rafraîchissement. Une transaction de rafraîchissement peut être faite soit en relançant la transaction initiale ou en propageant ses effets à la base de données comme une séquence d'opérations d'écriture.

2. Définition des concepts de base

Les travaux effectués dans [1] nous permettent de définir les termes de fraîcheur d'une réplique (ou Data Node) et d'obsolescence tolérée d'une transaction. Ces deux concepts dépendent de la quantité de changement d'une transaction.

a) Quantité de changement d'une transaction

Un changement consiste à faire des modifications sur les éléments d'une relation. Une transaction de mise à jour implique généralement des changements de relations. C'est ainsi

[4] Il existe aussi l'approche PUSH où c'est au Transaction Manager de déterminer le Data Node qui donnera le temps de traitement de transaction le plus optimal pour lui envoyer des transactions

que nous définissons la quantité de changements au niveau des relations. Supposons R_i une relation i et T une transaction de mise à jour qui modifie R_i.

Nous notons par ***Change(T,R_i)*** le nombre maximum de tuples que T doit modifier dans la relation R_i.

Ainsi, si la transaction accède à un ensemble de relations noté $Write_T$ alors la quantité de changement de T est définie par la quantité de changement de chaque relation R_i et qui est notée par:

$$Change(T) = \{(R_i, Change(T,R_i)) / R_i \in Write_T\} \qquad [1]$$

b) Fraîcheur d'un Data Node

Avec la réplication symétrique asynchrone adoptée dans notre système, plusieurs répliques (sur différents nœuds du système) peuvent avoir des états différents à un instant donné, car elles n'ont pas encore atteint l'état complètement cohérent, c'est-à-dire l'état obtenu après l'exécution correcte de toutes les transactions reçues.

La fraîcheur d'une réplique correspond à la quantité de changements (effectués sur les autres répliques de la même base de données) qui n'ont pas encore été appliqués à la réplique. Si cette quantité de changement vaut zéro, le nœud aura la fraîcheur maximale, c'est-à-dire que son état est complètement cohérent.

Ainsi, la fraîcheur de la réplique R correspond à la quantité de changements effectués sur toutes les autres répliques sauf R.

Soit T_R l'ensemble des transactions qui ont modifié ces autres répliques, la fraîcheur F de la relation R peut donc être définie comme étant la somme des changements effectués par T_R :

$$F(R)=\sum(Change(T_i,R)/ T_i \in T_R) \qquad [1]$$

Ainsi, la fraîcheur F d'un nœud N possédant la réplique d'une base de données de n relations est définie par

$$F(N)=\{(R_i,F(R_i)) / i=1,n\} \qquad [1]$$

c) Définition de l'obsolescence tolérée d'une transaction

L'obsolescence tolérée d'une transaction correspond à la valeur maximale de fraîcheur que la transaction accepte lors de la lecture des données sur un Data Node. Cette obsolescence tolérée de la transaction est exprimée par l'obsolescence tolérée ***F(R_i,T)*** de chaque relation R_i accédée par la transaction. L'obsolescence tolérée d'une relation est une valeur positive

définie par le client, qui exprime l'obsolescence tolérée maximale requise par le client pour sa transaction T, lors de la lecture de la relation R_i.

Supposant R_T l'ensemble des relations accédées en lecture par la transaction T, l'obsolescence tolérée F de T est définie par :

$$F(T)=\{(R_i,F(R_i,T)) / R_i \in R_T\} \qquad [1]$$

Si T est une transaction de mise à jour, cette valeur est toujours égale à zéro. En d'autres termes, les transactions de mise à jour ne tolèrent pas d'obsolescence, pour faire en sorte que les transactions conflictuelles soient exécutées sur chaque nœud dans le même ordre relatif.

L'obsolescence tolérée associée avec une transaction permet l'envoie d'une transaction sur un nœud même s'il n'est pas parfaitement frais.

3. Disponibilité d'un Data Node

On suppose que lorsqu'une transaction s'exécute, elle utilise presque toutes les ressources du Système de Gestion de Base de Données et qu'un Data Node traite les transactions les unes après les autres.

Sur chaque DN est définie une file d'attente ou buffer avec une taille donnée qui va contenir toutes les transactions envoyées sur ce nœud et qui ne sont pas encore en exécution en d'autres termes toutes les transactions en attente sur celui-ci.

La disponibilité du DN va dépendre d'un paramètre appelé ***degré de disponibilité*** correspondant au nombre de transactions restant pour que la file d'attente du Data Node soit pleine.

Si ce nombre est positif, alors le DN est considéré comme étant disponible et il envoie un message contenant son degré à un TM pour lui déclarer sa candidature à l'exécution d'une transaction. Ainsi, à sa déclaration le Transaction Manager utilise l'algorithme défini dans la quatrième partie de ce chapitre pour lui envoyer une ou des transaction à exécuter.

Nous définissons au niveau des DNs un temps appelé ***temps_disp***. C'est le temps pendant lequel un DN va attendre avant d'envoyer un nouve au message de disponibilité au Transaction Manager après avoir reçu des transactions.

Si le ***temps_disp*** est long et que le DN possède une puissance d'exécution très élevée, la présence de transactions à exécuter sur sa file d'attente sera très faible et le nombre diminue au cours du temps ce qui peut rendre le nœud inactif puisque sa file sera vide. Un ***temps_disp*** très court provoquerait aussi un nombre très élevé de message de disponibilité dans le système. Ce sont les raisons pour lesquelles nous supposons un temps moyen qui évite toute

surcharge du système mais aussi l'inactivité des nœuds de données. Ce temps sera donc choisit suivant la nature du système.

4. Gestion de la cohérence mutuelle des Data Nodes

Nous supposons dans notre système une seule base de données répliquée sur tous les nœuds de données et nous supposons aussi que toute transaction entrante peut être entièrement exécutée sur un unique nœud, c'est-à-dire que nous ne nous occupons pas des transactions distribuées.

Avec ce type de réplication, la cohérence mutuelle de la base de données peut être compromise par l'exécution non contrôlée de transactions conflictuelles sur les différents nœuds de données. Pour résoudre ce problème, les transactions de mis à jour sont exécutées sur les nœuds dans des ordres compatibles, produisant ainsi des états cohérents sur toutes les répliques de la base de données. Les requêtes sont envoyées sur tout nœud qui est assez frais et qui respecte les exigences de la requête. Ceci implique qu'une requête peut lire différents états de la base de données selon le nœud où elle est envoyée. Cependant, puisque les requêtes ne sont pas distribuées, elles lisent toujours un état cohérent de la base.

Pour assurer la cohérence mutuelle des répliques, les mises à jour doivent être propagées vers les répliques. Cette propagation doit s'effectuer en surchargeant le moins possible le système, donc il faut éviter les synchronisations inutiles. Une première méthode peut consister à ne propager les mises à jour que si cela est nécessaire pour traiter une nouvelle transaction sur un DN. Ceci peut entrainer un ralentissement du traitement de la transaction entrante. Si les DNs restent assez longtemps sans être mis à jour, leur fraîcheur à tendance à diminuer continuellement. De ce fait toute transaction exigeant une forte obsolescence tolérée, va déclencher une très grande synchronisation.

Ce faisant, les propagations des mises à jour se font par anticipation non pas par la méthode ***« push »*** mais par ***« pull »*** c'est-à-dire sur demande des Data Nodes. Sur un DN est défini un module qui détermine le temps adéquat pour envoyer un message contenant la dernière transaction de sa file d'attente au Transaction Manager pour lui demander l'état du système, i.e. demander les dernières transactions qui sont envoyées sur les autres répliques et qui ne sont pas encore acheminées vers lui. Le TM ayant reçu ce message détermine à partir du Shared Directory les transactions à propager sur ce DN si son état de cohérence n'est pas total.

IV. Notre algorithme de routage

Nous définissons sur le Transaction Manager une structure de données sous forme d'une file que nous appelons *file d'attente* où les transactions venant des Client Nodes seront ordonnées avant d'être acheminées. La discipline utilisée pour la sortie des transactions de la file est le FIFO (First In First Out)[5].

Les Data Nodes envoient au Transaction Manager pour déclarer leur disponibilité des messages contenant non seulement leur degré de disponibilité mais aussi la dernière transaction qui leur a été envoyée. Ainsi, un DN en se déclarant disponible, le Transaction Manager choisit des transactions à lui envoyer. Ce nombre va dépendre essentiellement de son degré de disponibilité. On suppose deux cas de disponibilité dans le système: le cas où un seul Data Node déclare sa disponibilité et le cas où plusieurs DNs se déclarent disponibles.

1. Cas où un seul DN déclare sa disponibilité

Un Data Node déclarant sa disponibilité à recevoir une transaction envoie aussi son degré de disponibilité. Ainsi, le TM ayant reçu ce message va utiliser cette procédure pour choisir les transactions à acheminer vers ce DN

```
Entrée : n (correspond au degré de disponibilité du Data Node)
Sortie : Seq (correspond à la séquence des transactions à envoyer au Data Node)
-------------------------------------------------------------------------------
F(N)=detFraicheur()  //déterminer la fraîcheur du Data Node
Tant que n>0
    F(Tk) = detFraicheurTolere() //déterminer l'obsolescence tolérée de la transaction
    Si F(N) > F(Tk) alors
        Seq = ajouter(Tk) // Tk est ajoutée à la séquence des transactions à envoyer
        decrementerUn(n)     // le degré de disponibilité est réduit de 1
        enlever(Tk)          // Tk est enlevée de la file du TM
    Sinon
        Tsync = determinerTsync() // l'ensemble des transactions à propager pour que
                                  // F(N) atteigne F(Tk)
        m=nombreTrans(Tsync)
        Si m<n-1 alors
            Seq= ajouter(Tsync+Tk) // Tsync et la transaction sont ajoutées à Seq
            decrementerM(n)        // le degré de disponibilité est réduit de m
            enlever(Tk)
        Sinon
            Seq=ajouter(Traf)  // Traf est une séquence de transactions appartenant
                               // à l'ensemble Tsync tel que le nombre soit égal à n
        Fin si
    Fin si
Fin tant que
 Seq=déterminerOrdre()  //la séquence est triée selon l'ordre de précédence
 envoyer(Seq)  // la séquence de transactions est envoyée au Data Node
```

[5] PAPS en français, Premier Arrivé Premier Servi

Le TM commence par déterminer la fraîcheur du Data Node. Toute transaction envoyée et exécutée sur un DN est enregistrée dans le Shared Directory. Ainsi, pour évaluer la fraîcheur le TM consulte l'annuaire. Puisque le Shared directory maintient un graphe appelé ***graphe global*** qui contient toutes les transactions exécutées sur les DNs, le TM détermine l'ensemble des transactions exécutées se trouvant entre la dernière transaction de la file d'attente et celle envoyée en dernière lieu sur le Data Node. Connaissant cet ensemble de transactions non appliquées au DN, le TM détermine la quantité de changements de chacune d'elle. Par la suite il détermine la fraîcheur du DN.

Puisque c'est le seul DN disponible, le TM cherche à lui envoyer une séquence de transactions équivalente à son degré de disponibilité.

Le TM après avoir déterminer l'obsolescence tolérée de la transaction se trouvant à la tête de sa file, la compare avec la fraîcheur du DN. Si le DN est assez frais (i.e. sa fraîcheur est supérieure à l'obsolescence tolérée) pour recevoir la transaction alors celle-ci est ajoutée à la séquence des transactions à envoyer. Par contre, si le DN n'est pas frais, le TM détermine l'ensemble des transactions à propager pour que la fraîcheur du DN atteigne l'obsolescence tolérée de la transaction. Pour voir si le DN pourra recevoir la transaction et les transactions de cet ensemble, le TM vérifie si le nombre est inférieur au degré de disponibilité réduit de un. Si tel est le cas alors cet ensemble est ajouté à la séquence avec la transaction. Sinon, i.e. il y'a un nombre de transactions à synchroniser plus élevé que le degré de disponibilité, le DN sera rafraîchi avant que le TM ne puisse lui envoyé une nouvelle transaction de sa file.

2. Cas où plusieurs Data Node déclarent leur disponibilité

Supposons que ***m*** (avec ***m***>1) Data Node déclarent au Transaction Manager qu'ils sont disponibles à recevoir une transaction. Dans ce cas le TM utilise la procédure suivante pour le routage des transactions

```
Entree : m (correspond au nombre de Data Node disponible)

// déterminer la fraîcheur de chaque DN
Pour chaque DNi faire
     F(Ni) = detFraicheur()
Fin pour

Tant que m > 0
   F(N) = detDnPlusFrais() //déterminer le Data Node le plus frais
   F(Tk) = detFraicheurTolere() //déterminer l'obsolescence tolérée de la transaction
   Si F(N) > F(Tk) alors
       envoyer(Tk)  // Tk est envoyée au Data Node
       enlever(Tk)          // Tk est enlevée de la file du Transaction Manager
       decrementer(m)     // le nombre de DN dispo est réduit
   Sinon
      Tsync = determinerTsync() // l'ensemble des transactions à propager pour que
                                // F(N) atteigne F(Tk)
      nb_Trans = nombreTrans(Tsync)
      Si nb_Trans < n-1 alors   // n étant le degré de disponibilité du Data Node
           envoyer(Tsync+Tk)   // Tsync et la transaction sont envoyées DN
           enlever(Tk)
           decrementer(m)
      Sinon
           envoyer(Traf)  // Traf est une séquence de transactions appartenant
                          // à l'ensemble Tsync tel que le nombre soit égal à n
           decrementer(m)
      Fin si
   Fin si
Fin tant que
```

Le Transaction Manager ayant reçu le message de disponibilité de ***m*** Data Nodes détermine la fraîcheur de chacun d'eux.

Le Transaction Manager essaie d'envoyer à chaque DN une ou des transactions. Le nombre de transactions va dépendre non seulement du degré de disponibilité du DN qui est contenu dans le message mais aussi de la fraîcheur de ce DN.

Après avoir déterminé leur fraîcheur, le TM choisit le plus frais et compare sa fraîcheur avec l'obsolescence tolérée de la transaction se trouvant à la tête de sa file d'attente. Si le Data Node est assez frais pour recevoir la transaction alors celle-ci lui est envoyée. Dés qu'une transaction est envoyée à un DN, elle est enlevée de la file pour qu'à la prochaine itération la transaction qui la suit soit à la tête de la file. Par contre si le DN n'est pas assez frais pour recevoir la transaction, le TM détermine l'ensemble des transactions à propager sur le DN pour que sa fraîcheur atteigne l'obsolescence tolérée de la transaction.

Le TM compare le nombre de transactions de cet ensemble avec le degré de disponibilité du DN. Si ce nombre est inférieur au degré de disponibilité moins un alors l'ensemble des transactions sont envoyées avec la transaction. Sinon, i.e. il y'a beaucoup plus de transactions

à synchroniser que de places disponibles dans la file du Data Node, le TM choisit un nombre de transactions égal au degré de disponibilité dans cet ensemble pour l'envoyer au Data Node afin de le rafraîchir.

Conclusion

Dans ce ch apitre, nous avons essayé de présenter un mécanisme pour le traitement de transactions dans les systèmes à l arge échelle. Contrairement aux algorithmes existants comme celui présenté dans [2], notre algorithme effectue un routage sans connaître à priori le temps d'exécution d'une transaction. Les différents concepts sur lesquels il se base sont surtout la fraîcheur des différents nœuds de données du système, l'obsolescence tolérée des transactions des clients mais aussi de la disponibilité des nœuds de données.

Chapitre 3

Validation de notre algorithme de routage

Ce chapitre est consacré aux simulations que nous avons effectuées pour valider notre algorithme de routage. En effet après avoir mis en place l'algorithme de routage de transactions, nous nous sommes posé un certain nombre de questions concernant surtout son passage à l'échelle et l'équilibrage de charge.

Puisqu'il est difficile voire même impossible de le mettre en œuvre dans un réseau à large échelle réel, nous avons utilisé un outil réseau P2P qui nous a permis de simuler le comportement du système.

Durant nos expériences, nous avons utilisé le simulateur FreePastry [4] que nous décrirons brièvement, puis nous présenterons les différents modules d'implémentation de notre algorithme avant de terminer par donner et commenter les graphes résultant des simulations effectuées.

I. Le simulateur réseau FreePastry

FreePastry [4] permet l'implémentation d'un overlay Pair-à-Pair. Il est *open source* et son code écrit en Java lui assure une grande modularité mais aussi une forte extensibilité. FreePastry permet de simuler le mode de fonctionnement d'un réseau à grande échelle avec des milliers voir même des centaines de milliers de nœuds.

Il est aussi un simulateur à événement discret. Un ensemble de messages est défini et des protocoles sont exécutés selon les messages survenus. FreePastry permet l'exécution d'un ensemble d'applications sans modification du code source.

L'outil FreePastry présente un ensemble de package contenant chacun des classes et des interfaces qui permettent de mettre en œuvre une simulation. Parmi ces packages nous trouvons ***rice.p2p.commonapi*** qui est l'interface universelle des overlays structurés et qui comporte la plupart des interfaces utiles pour nos implémentations. Les interfaces les plus importantes sont :

- ✓ ***Application***: permet de créer les différentes applications qui s'exécutent sur les nœuds du système. C'est une interface que toute application tournant sur un nœud doit importer. Elle permet aussi à un nœud de transmettre des messages, d'informer les applications du passage des messages et des changements effectués sur les

nœuds voisins. Il est possible d'avoir plusieurs applications sur un nœud, et il est aussi possible d'avoir plusieurs instances d'une application sur le même nœud.

La méthode ***deliver()*** de cette interface permet de simuler le fonctionnement d'un système. Elle est appelée sur le nœud destinataire à chaque fois que celui-ci reçoit un message.

- ✓ ***Message :*** cette interface représente l'abstraction d'un message dans l'API de FreePastry. Ainsi, les messages envoyés vers les autres nœuds doivent obligatoirement implémenter cette classe.

FreePastry possède un autre package très important ***rice.pastry.direct*** contenant des classes et interfaces qui permettent de créer un réseau P2P et les différents nœuds le constituant. Selon leur utilité nous pouvons citer:

- ✓ celles qui permettent de créer le réseau. Il existe différentes topologies de réseau et elles sont toujours spécifier grâce aux classes: ***EuclideanNetwork*** qui permet d'avoir un réseau avec une topologie plane et qui ne prend pas en compte des latences entre les différent nœuds du système, et la classe ***GeneticNetwork*** qui permet d'importer une matrice des latences dans le but de simuler un modèle de topologie réelle
- ✓ celles qui permettent de définir la manière dont les nœuds seront liés dans le réseau : par exemple le ***DirectPastryNodeFactory*** qui permet d'avoir des liens directs entre les nœuds
- ✓ et enfin la classe qui permet de créer les nœuds : la classe ***PastryNode.***

II. Implémentation de notre algorithme

Notre objectif est de faire un routage efficace des transactions. Ce routage doit s'effectuer tout en assurant l'équilibrage des charges mais aussi la cohérence des bases de données. Dans notre système, nous avons une seule base de données qui est répliquée totalement sur les différents nœuds de données. Nous avons utilisé un mode de réplication asynchrone, qui tolère un certain niveau de divergence entre les répliques de la base se trouvant sur les Data Nodes. Une réplique est obsolète lorsqu'elle n'a pas encore reçu toutes les transactions qui lui sont destinées.

Il y'a trois types de transactions dans le système, les transactions à lecture seule (ou requêtes), les transactions de mise à jour (ou transaction) et les transactions de rafraîchissement. Les requêtes acceptent un certain niveau de divergence (appelé

obsolescence tolérée) alors que les transactions doivent toujours s'exécuter sur une base de données totalement fraîche. Ainsi, à l'exécution d'une transaction demandant une certaine fraîcheur, le rafraîchissement est obligatoire.

L'exécution des transactions peut se résumer comme suit :

i. Les Client Nodes envoient soit une requête ou une transaction au Transaction Manager (TM) avec une obsolescence tolérée. Pour rappel l'obsolescence tolérée d'une transaction de mise à jour est toujours égale à zéro

ii. Celles-ci sont mises dans la file d'attente des TMs avant d'être routées. Les Data Nodes envoient leur disponibilité à l'exécution d'une transaction au TM qui leur envoie en revanche un ensemble de transactions correspondant à leur degré de disponibilité tout en vérifiant leur degré de fraicheur et l'obsolescence tolérée des transactions de la file des TMs.

iii. Un DN après avoir exécutées une transaction envoie la réponse directement au Client Node et un accusé sur l'état d'exécution au TM pour que ce dernier fasse le nécessaire au niveau des métadonnées en vue de garder la cohérence de la base.

Pour simuler le fonctionnement de notre système nous avons développé les cinq classes suivantes: MyApplication, MyMessage, SharedDirectory, Couple, et MySimulation. Chacune de ces classes comporte un certain nombre d'attributs et de méthodes.

1. La classe MyApplication

C'est la classe qui permet de définir les différents types de nœuds du système. C'est aussi la classe principale où notre algorithme sera implémenté. L'envoi et la réception des transactions s'effectuent à l'aide des méthodes qui y sont définies. Parmi ses méthodes nous pouvons citer entre autre la méthode *routeMyMsgDirect(NodeHandle node, String message)* et la méthode *public void deliver(Id id, Message message)*. MyApplication implémente l'interface ***Application*** du package universel de FreePastry ***commonAPI***.

• Les attributs

public static Vector<String> file_attente = new Vector<String>(): représente la file d'attente des Transaction Managers qui contiendra les transactions envoyées par les Client Nodes avant qu'elles ne soient acheminées vers les Data Nodes.

public static Vector<Long> temps = new Vector<Long>() : stocke le temps d'exécution des transactions validées dans le système

public static HashMap<String, Long> listTrans = new HashMap<String, Long>(): permet de faire la correspondance entre une transaction avec son début d'exécution

protected Long debut : représente le début global d'une transaction. Il correspond au moment où la transaction est envoyée vers un Data Node

protected Long fin : correspond à la fin d'exécution d'une transaction. Il est évalué dès qu'un TM reçoit un accusé d'exécution pour la transaction.

protected Long tempsTotal: c'est le temps d'exécution d'une transaction, équivalent à la différence des deux temps calculés précédemment

protected long puissance: correspond à la puissance d'un Data Node

- **Les méthodes**

public MyApplication(Node node, String type, long puissance) : c'est le constructeur de la classe. Elle permet d'initialiser les paramètres d'une application s'exécutant sur un nœud du système tels que le type (CN, TM ou DN) et la puissance ;

public String getType(): renvoie le type de nœud ;

public void setType(String type): définit le type de nœud (i.e. CN, TM ou DN) ;

public long getPuissance(): renvoie la puissance d'exécution d'un Data Node ;

public void routeMyMsgDiect(NodeHandle node, String message): cette méthode permet d'envoyer directement un message à nœud. Elle utilise la méthode *route* de l'API FreePastry ;

public void deliver(Id id, Message message)*:* c'est la méthode qui est appelée à chaque fois qu'un nœud reçoit un message. Selon le type de l'application (CN, TM ou DN) qui tourne sur ce nœud, il y'a un comportement spécifique avec le type de message reçu.

2. La classe MyMessage

Elle permet de définir le type de message qui sera envoyé par un nœud. La nature du message dépend du type de nœud qui doit l'envoyer. Cette classe implémente l'interface ***Message*** du package ***commonAPI*** de FreePastry.

3. La classe Couple

La correspondance entre les transactions exécutées sur un Data Node est effectuée à l'aide de cette classe puisqu'on a toujours un couple constitué de (numeroTransaction, nomDn). Elle est surtout utilisée pour l'évaluation de la fraîcheur d'un Data Node.

4. La classe SharedDirectory

Cette classe joue le rôle d'un annuaire puisque l'historique des transactions exécutées sur les DNs y est stocké. Il est défini dans cette classe des méthodes permettant de faire cet historique.

- **Les attributs**

public static Vector<Couple> sd = new Vector<Couple>(): vecteur permettant de stocker les couples de transactions exécutées sur les Data Nodes.

- **Les méthodes**

public static void insert(Couple c)*:* insère chaque couple dans le vecteur ;

public static Stringbuffer detTransDn(String dn, Vector<Couple> vect)*:* elle permet de déterminer l'ensemble des transactions envoyées sur les autres Data Nodes et qui ne se sont pas encore propagées sur le DN donné en paramètre ;

public static Stringbuffer sequenceTrans(String Dn, Vector<Couple> vect)*:* détermine l'ensemble des transactions envoyées sur le Data Node ;

public static int fraicheurDn(String dn): détermine la fraîcheur d'un Data Node ;

public static void ecrireResult(String tps): cette méthode permet d'écrire les résultats des temps d'exécution des transactions dans un fichier ***.CSV***.

5. La classe MySimulation

Elle représente la classe principale de la simulation. La création du réseau et des nœuds du système y sont effectuées. De même l'initialisation des paramètres des nœuds tels que le type, la puissance d'exécution, etc. C'est aussi dans cette classe que l'exécution de la simulation sera démarrée puisque la méthode principale (main) s'y trouve.

III. La simulation

Dans cette simulation nous avons comme principal objectif de vérifier le passage à l'échelle de notre algorithme :

i. dans les premières expériences, nous allons montrer l'impact du nombre de Data Nodes sur le temps moyen d'exécution des transactions ;
ii. ensuite, nous allons montrer que notre algorithme permet de faire un équilibrage de charge lors de l'exécution des transactions des Client Nodes.

iii. et enfin nous allons terminer par comparer notre approche avec l'approche Round Robin afin de montrer l'apport de notre algorithme par rapport à celui du Round Robin.

1. Spécifications

- ✓ Pour prendre en charge de l'hétérogénéité dans le système, dans nos différentes expériences nous avons des Data Nodes qui ont des puissances d'exécution et des capacités de file d'attente différentes.
- ✓ Les transactions et les requêtes simples sont mélangées et nous avons supposées que le temps d'exécution d'une transaction et celui d'une requête simple sur un DN donné est le même.
- ✓ Chaque transaction (mise à jour ou requête simple) est soumise avec une obsolescence tolérée. Une obsolescence qui est toujours égale à zéro pour les transactions de mises à jour.

2. Impact du nombre de nœuds sur le temps moyen d'exécution

La première expérience vise à déterminer le temps moyen d'exécution des transactions émises par les clients et regarder son évolution en augmentant les Data Nodes du système. Nous fixons le nombre de Client Nodes à 10 et chacun d'eux envoie 100 transactions (soit des mises à jour ou des requêtes simples), et nous faisons varier le nombre de Data Nodes de 50 à 750 et nous évaluons à chaque fois le temps moyen global d'exécution des transactions.

Comme le montre la Figure 8, l'augmentation du nombre de Data Node a une influence considérable sur le temps moyen d'exécution. Nous observons une diminution du t emps quantifié ; ceci s'explique par le fait que le traitement est partagé entre les différents Data Nodes du système. A 750 DNs, nous avons une diminution de 67% du temps d'exécution global.

Toujours dans l'optique de montrer l'impact du nombre de DNs sur le temps d'exécution, dans une seconde expérience, nous fixons le nombre de Data Nodes et augmenter les transactions envoyées par les Client Nodes (i.e. augmenter le nombre de clients dans le système). Dans la Figure 9, nous faisons varier le nombre de transactions entre 100 et 1000. Nous effectuons trois expériences où le nombre de DNs est fixé respectivement à 250, 500 et 750 et nous évaluons pour chacune de ces expériences le temps moyen global d'exécution.

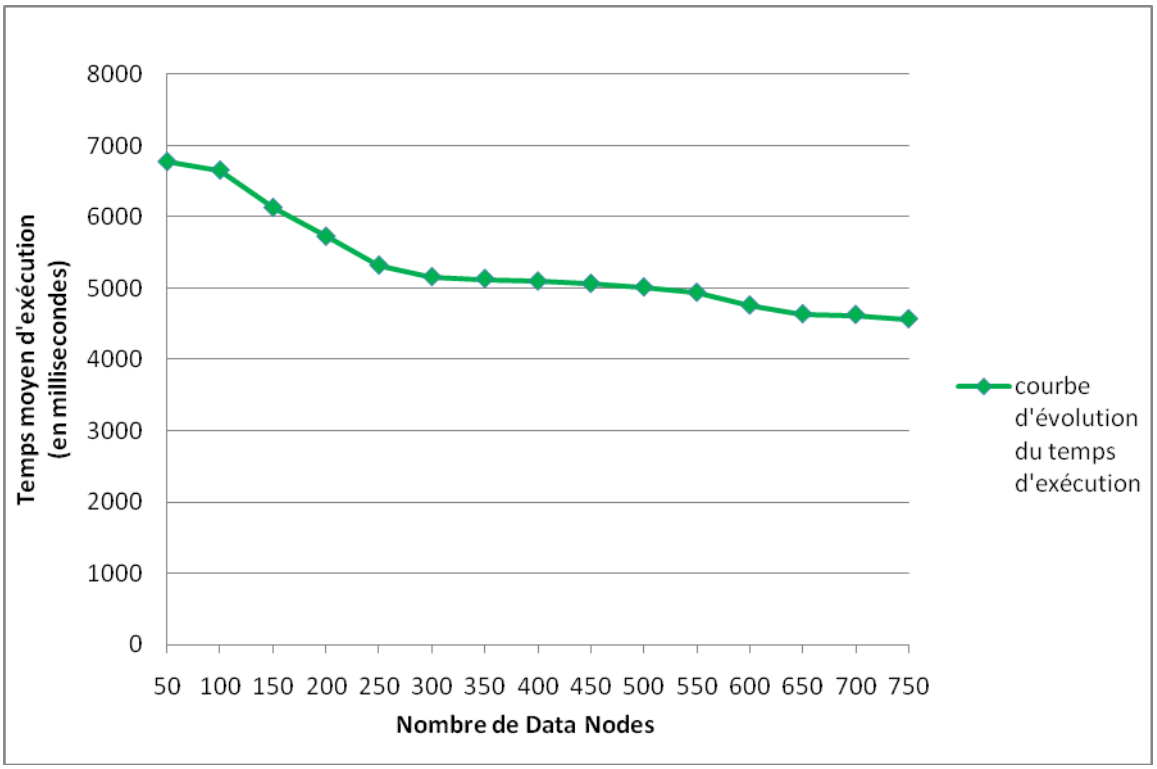

Figure 8 : Impact du nombre de DNs sur le temps d'exécution en fixant le nombre de Client Nodes

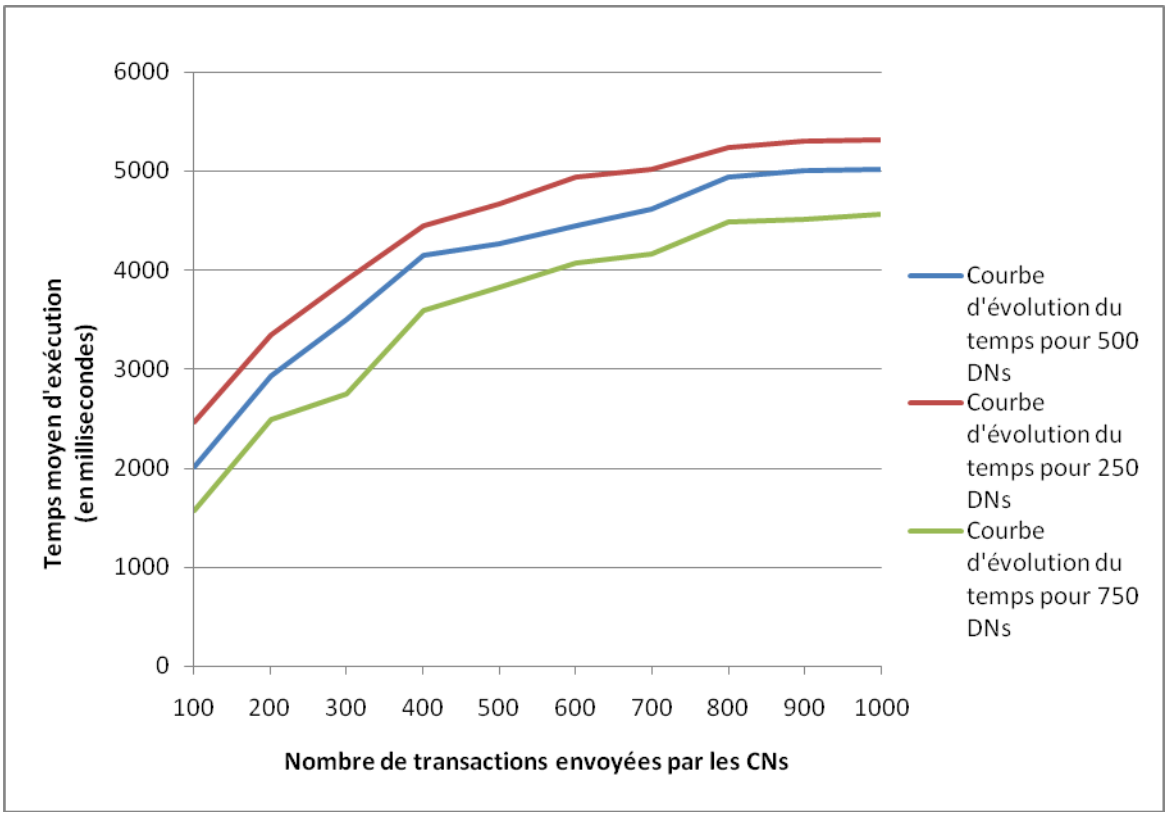

Figure 9: Impact du nombre de DNs sur le temps d'exécution en variant le nombre de Client Nodes

Nous observons dans cette Figure 9 que le temps d'exécution augmente de manière considérable quand nous augmentons le nombre de transactions envoyé par les clients. Nous remarquons aussi dans cette Figure 9 que l'impact du nombre de DNs sur le temps moyen d'exécution des transactions est le même qu'à la Figure 8. En effet, pour un même nombre de transactions, plus le nombre de DNs est important plus le temps d'exécution est petit.

Avec ces deux expériences nous pouvons voir l'impact majeur que les Data Nodes ont sur le traitement des transactions des clients.

Après avoir montré l'influence du nombre de Data Nodes sur le temps d'exécution moyen global des transactions, nous regardons dans une autre expérience comment ceux-ci sont utilisés dans l'exécution des transactions. Pour ce faire, nous faisons varier les DNs tout en fixant le nombre de transactions envoyées et évaluer le pourcentage de DNs utilisé parmi l'ensemble déclaré.

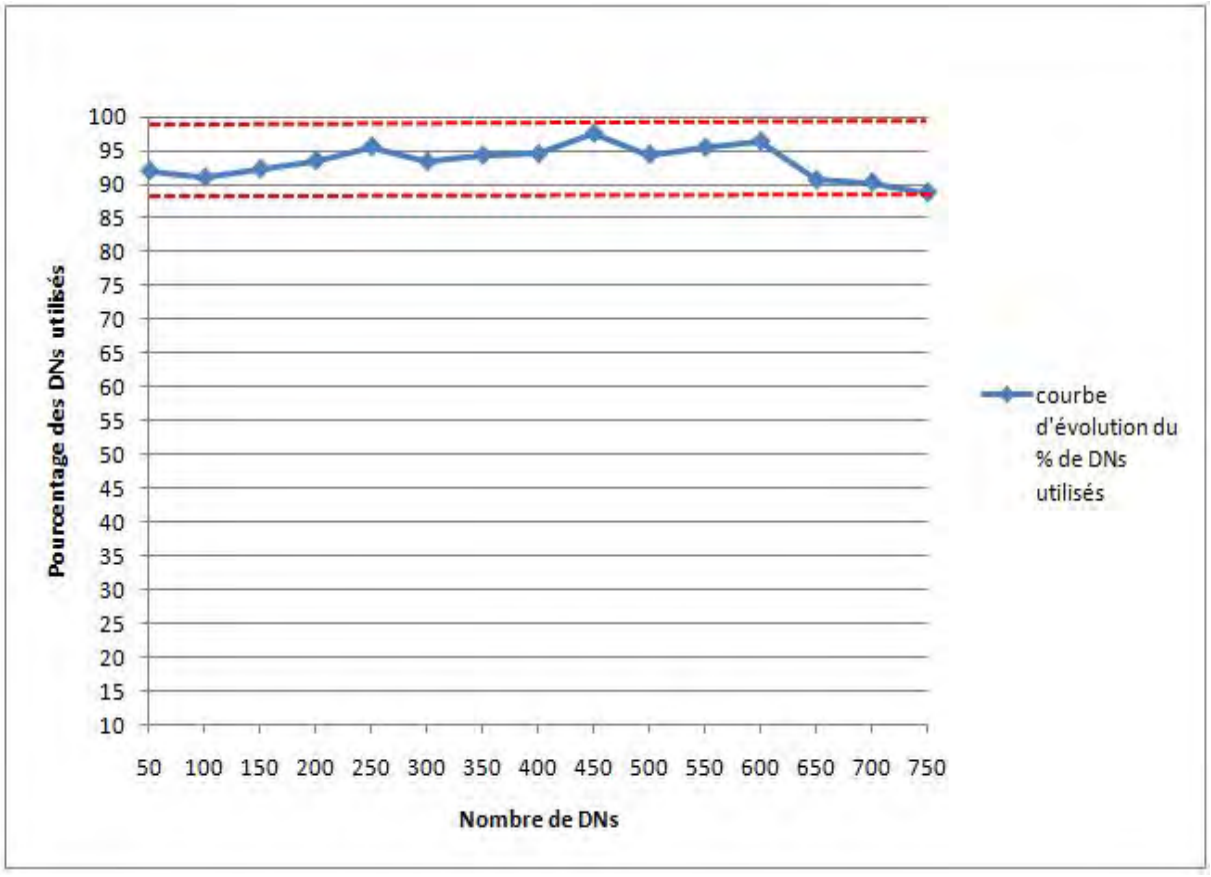

Figure 10: Evaluation du pourcentage des Data Nodes utilisés

À la Figure 10, nous constatons à chaque fois qu'au moins 85% des DNs sont utilisés, ce qui explique le fait que le temps moyen d'exécution global diminue de façon considérable quand on augmente les Data Nodes. Nous remarquons aussi que tous les DNs déclarés ne sont pas utilisés ; ceci peut s'expliquer par deux choses. La première peut être due par le fait qu'un

Data Node peut envoyer une disponibilité alors qu'il n'était pas assez frais en ce moment. Puisque son état est obsolète, seules des transactions de rafraîchissement lui seront envoyées. La seconde cause peut être due par le fait qu'à chaque fois qu'il est disponible, il n'y a pas de transactions dans la file du TM. De ce fait il sera rafraichi si son état est obsolète.

3. Equilibrage de charge

Pour vérifier l'équilibrage de charge, nous utilisons la variable appelée ***coefficient de variation*** qui représente le rapport entre l'écart type et la moyenne d'une distribution. Pour ce faire, pour chaque expérience nous déterminons le nombre de transactions exécutées par chaque Data Node. Avec cette distribution de valeurs obtenues, l'écart type qui permet de voir la largeur de la distribution est calculé, de même nous calculons la moyenne de la série. Connaissant ces deux constantes, nous effectuons leur rapport.

Généralement, un coefficient de variation petit signifie que l'écart type est petit par rapport à la moyenne, ce q ui permettra de dire que les différentes valeurs de la distribution sont regroupées autour de la moyenne. En d'autres termes, plus le coefficient de variation est petit, plus la distribution est uniforme.

Figure 11: Variation des coefficients de variation en fonction des Data Nodes

Durant cette expérience, le nombre de Data Nodes varie entre 50 et 750 et le nombre de Client Nodes est fixe. Comme le montre la Figure 11, nous remarquons une constance du coefficient de variation avec des valeurs très petites puisqu'elles se situent entre 0,36 et 0,47. Avec de telles valeurs, nous pouvons dire que les transactions sont quasi-uniformément réparties sur les Data Nodes. Ces résultats montrent que la différence du no mbre de transactions exécuté par chaque DN n'est très élevé ; ce qui nous a permis de dire que l'équilibrage de charge est assuré dans notre système où notre algorithme est implémenté.

4. Comparaison de notre approche avec Round Robin

Afin de montrer la performance de notre approche, nous allons comparer le temps d'exécution global des transactions par rapport au temps d'exécution global donné par l'approche du tourniquet ou Round Robin. Avec ce dernier, les Data Nodes sont choisis à tour de rôle pour l'exécution des transactions se trouvant dans la file d'attente du Transaction Manager. Pour effectuer la simulation, nous nous mettons dans les mêmes contextes que lors des expériences d'évaluation du temps global de notre approche.

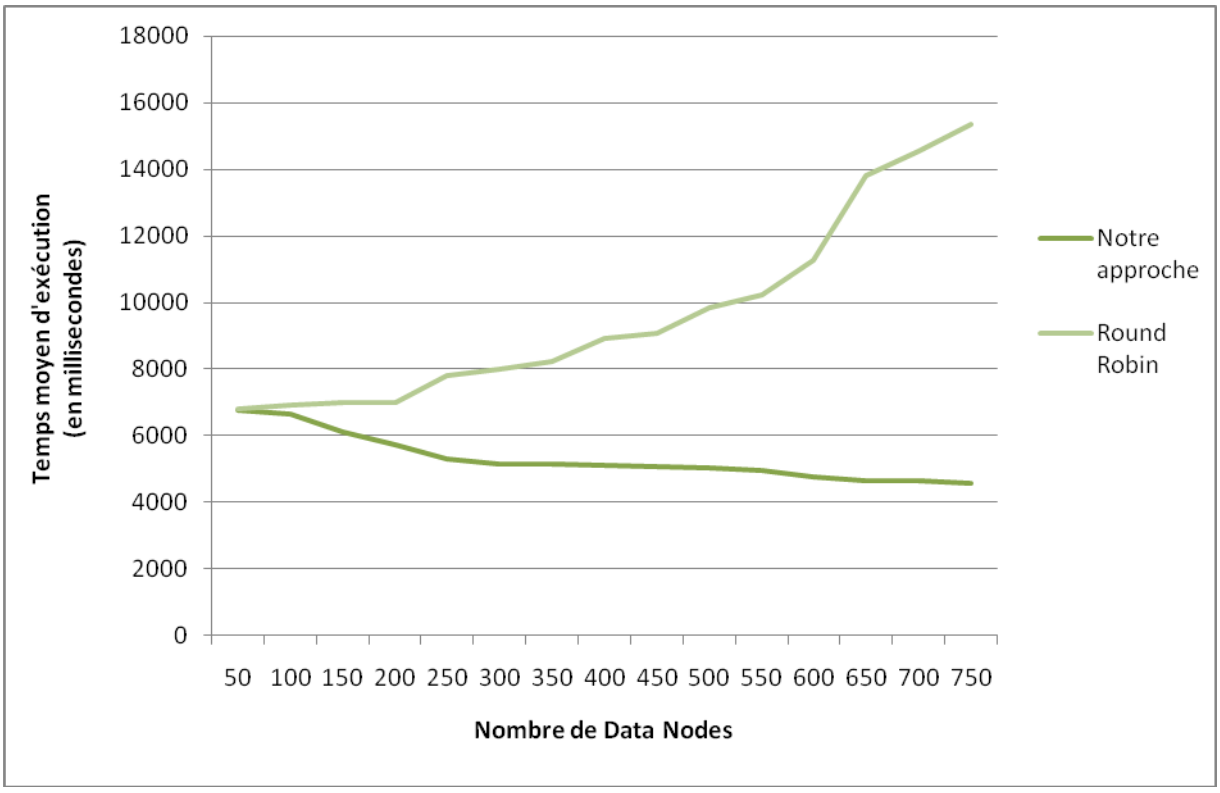

Figure 12: Notre approche vs Round Robin

Dans cette figure 12, nous constatons un temps d'exécution plus élevé pour Round Robin par rapport à notre approche si le nombre de DN dépasse 150. Nous observons dans le cas du

Round Robin un temps qui s'accroit de façon remarquable lorsque le nombre de nœuds DN augmente. Ceci s'explique par le fait que, quand un Data Node est élu pour l'exécution d'une transaction alors qu'il y'a des transactions qui précédent cette dernière, alors celles-ci sont envoyées en premier lieu, augmentant ainsi son temps d'exécution. Notons que dans cette stratégie le Data Node est rafraîchi lors de l'exécution des transactions.

Durant la phase de simulation, nous avons pu voir que la plupart des caractéristiques des P2P sont pris en compte par notre système. Les Client Nodes, les Transaction Managers comme les Data Nodes effectuent leurs opérations de manière autonome ce qui donne à notre système une certaine autonomie. Les Figures 8 et 9 montrent la scalabilité et la qualité de service du système puisque les ressources des nœuds sont très bien utilisées donnant ainsi des temps de réponse très acceptables. L'équilibrage de charge s'effectue aussi avec notre système.

Conclusion

Nos différentes expériences nous ont permis de montrer que notre algorithme passe bien à l'échelle, car pouvant être adapté dans un contexte où il y'a plusieurs nœuds. Elles nous ont aussi permis de mettre en évidence l'impact du nombre de nœuds sur le temps moyen d'exécution qui diminue de manière considérable avec l'augmentation du nombre de Data Nodes, mais aussi de montrer l'équilibrage de charge. Ces différentes expériences ont eu à mettre en évidence l'apport de notre approche par rapport à l'approche du Round Robin(ou Tourniquet).

Conclusion et Perspectives

Tout au long de ce stage de DEA, nous nous sommes consacrés à l'étude du traitement de transactions dans les bases de données à large échelle.

Nous avons commencé par faire une étude des systèmes à large échelle. Elle nous a permis de mieux comprendre leur fonctionnement mais aussi de déterminer les avantages qu'ils présentent dans la gestion des données.

Vu leurs importances dans la gestion et le stockage des données, définir des mécanismes qui permettent d'exploiter efficacement ces données constitue un défi majeur. Cela consiste surtout à définir des algorithmes de routage des différentes opérations de lecture et d'écriture sur les données stockées qui permettent de garantir un accès rapide et cohérent des données dans ces environnements.

Certes des solutions ont été proposées mais certaines d'elles s'appuient sur des hypothèses supposées très fortes ou qui ne prennent pas en compte l'hétérogénéité des différents composants des systèmes.

Nous avons proposé un algorithme pour le routage des transactions qui prend en compte la différence qu'il y'a dans les puissances de traitement des nœuds constituant le système. En effet, c'est un algorithme qui effectue le routage sans connaître à priori le temps d'exécution d'une transaction. De même, la décision pour l'exécution de transactions vient des nœuds de données qui s'appuient sur un paramètre appelé ***degré de disponibilité.*** L'algorithme s'appuie aussi sur certains concepts de [2] tels que l'obsolescence tolérée des transactions et la fraîcheur courante des nœuds de données afin d'améliorer l'équilibrage de charge.

En utilisant le simulateur réseau FreePastry, nous avons validé notre approche en effectuant plusieurs expériences. Les résultats obtenus ont montré que notre approche s'adapte bien aux systèmes à large échelle. Ils montrent l'impact de la taille du système sur le temps d'exécution des requêtes soumises par les clients. Ces résultats montrent aussi l'augmentation de la qualité de traitement du système avec l'équilibrage de la charge. De même, ils nous ont aussi permis de voir ce que notre approche a apporté par rapport à la solution Round Robin ou Tourniquet.

Tout au cours de notre travail, les Transaction Managers avant d'acheminer une transaction consulte un seul annuaire où les opérations effectuées sur les nœuds de données y sont enregistrées. Par conséquent, il serait intéressant, pour un travail futur, définir un système où chaque Transaction Manager garde les informations des transactions qu'il a acheminé ; établir des liens de communication entre ces TMs pour qu'ils puissent échanger leurs informations. Nous envisageons aussi, dans un futur très proche, effectuer des expériences sur le ***temps_disp*** qui correspond à la durée que le Data Node attend pour envoyer de nouveau sa disponibilité ; pour voir son influence sur le temps de traitement des transactions afin de pouvoir déterminer le ***temps_disp*** moyen pour les nœuds de données.

Bibliographie

[1] Stéphane Gançarski, Hubert Naacke, Esther Pacitti and Patrick Valdurez. *The Leganet System: Freshness-Aware Transaction Routing in a Database Cluster*. Information Systems Journal 32(2), pp. 320-343, 2006

[2] Idrissa Sarr, Hubert Naacke and Stéphane Gançarski. *Routage Décentralisé de Transactions avec Gestion des Pannes dans un Réseau à Large Echelle*.

[3] Gardarin G., Gardarin O.. *Le Client Serveur*. Editions Eyrolles, Paris, 1997

[4] http://www.freepastry.org/FreePastry/

[5] Ian Foster and C. Kesselman, editors. *The Grid: Blueprint for a N ew Computing Infrastructure.* Morgan-Kaufmann, 1999.

[6] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. *Grid'5000: A Large Scale, Reconfigurable, Controlable and M onitorable Grid Platform*. In Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05), pages 99–106, Seattle, Washington, USA, November 2005.

[7] Rob Pennington. *Terascale Clusters and the TeraGrid*. In Proceedings of 6th International Conference/Exhibition on High Performance Computing in Asia Pacific Region, pages 407–413, Bangalore, India, December 2002. Invited talk.

[8] Ian Foster and Carl Kesselman. Globus*: A Met computing Infrastructure Toolkit*. The International Journal of Supercomputer Applications and High Performance Computing, 11(2):115–128, Summer 1997.

[9] Andrew S. Grimshaw, William A. Wulf, and the Legion team. *The Legion vision of a worldwide virtual computer*. Communications of the ACM, 1(40):39–45, January 1997.

[10] Site officiel du projet DataGrid: http://www.eu-datagrid.org

[11] LIANG (J.), KUMAR (R.) et ROSS (K.W.). « Understanding KaZaA », 2004.

[12] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steven Lim. *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes*.

[13] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp et Scott Shenker. *A Scalable Content-Addressable Network.*

[14] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.*

[15] C. Plattner and G. Alonso. *Ganymed: scalable replication for transactional web applications.* In Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, pages 155–174, 2004.