*gestion des transactions  dans un environnement*

L'étude de la gestion des transactions dans un environnement P2P requiert d'abord la compréhension de quelques concepts de base que nous spécifierons tout d'abord. Il s'agit particulièrement des concepts de P2P. Nous positionnerons par la suite la problématique de notre thème avant d'en venir au traitement proprement dit des transactions dans un système P2P.

## *I. Concepts de base des réseaux peer to peer*

### I.1. Définition du concept pair

Le terme " pair à pair " fait référence à plusieurs termes apparentés tels que " poste à poste ",  " égal à égal ", " peer to peer " (en anglais), ou encore les abréviations comme " P to P " ou " P2P ".
Le terme " pair à pair " signifie littéralement une relation d'échange réciproque qui unit directement deux acteurs de même statut.
Ce terme désigne, à l'origine sur Internet, une technologie d'échange de fichiers entre internautes permettant à deux ordinateurs de communiquer l'un avec l'autre sans passer par un serveur central qui redistribue les données.

### I.2. Réseaux P2P

En général, un système P2P est composé d'un ensemble de mécanismes  permettant la communication entre pairs, d'algorithmes pour localiser les ressources et d'une interface sur l'environnement distribué. La figure 15 décrit en détail la pile de protocoles des P2P. Notons que toutes les couches P2P sont au niveau de la couche application du modèle OSI.

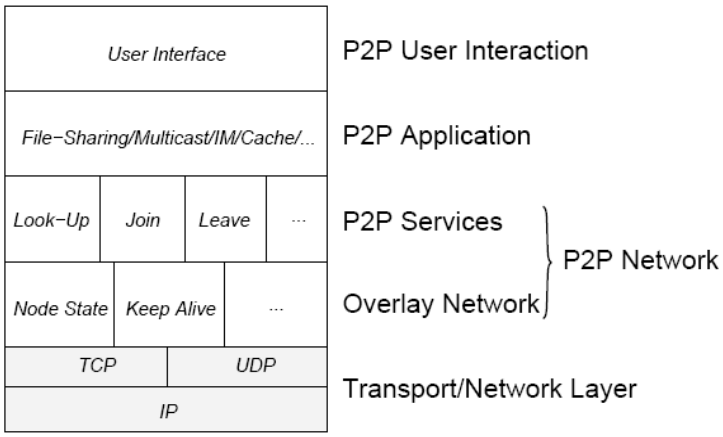


Figure 15 : Pile de protocole des P2P

Ce qu'on appelle réseaux P2P, est composé de la configuration du réseau (Overlay network) et des services P2P.

**a. Configuration du réseau**

Il est souvent représenté par un graphe direct ($N$, $L$) où $N$ représente l'ensemble des nœuds et $L$ les liens entre les différents nœuds. Le routage des paquets est effectué au niveau de la couche réseau du modèle OSI, cependant la décision de routage se fait au niveau de l'application.

**b. P2P Services**

Ce sont les différentes fonctionnalités fournies à la couche P2P application : comme la localisation des informations ou l'adjonction d'un nouveau nœud sur le réseau.

### I.2.1. Caractéristiques

Les caractéristiques principales de ces types de réseaux sont :

Chaque ordinateur peut prendre part au réseau : Il paye sa participation en autorisant l'accès sur une partie de ses ressources.

- Chaque pair peut être client ou bien serveur.
- L'inexistence d'un coordinateur central source de goulot d'étranglement
- L'autonomie des nœuds qui n'ont aucune vision globale du système mais ils interagissent pour faire émerger le comportement global.
- Les nœuds comme les connections ne sont pas fiables.

### I.2.2. Types de réseaux P2P

Généralement les réseaux P2P se rangent suivant deux grandes familles:

- les P2P structurés
- les P2P non-structurés

La différence principale entre les deux repose sur l'algorithme de recherche utilisé pour trouver des ressources dans le réseau P2P.

### I.2.2.1. Les P2P non-structurés

Les réseaux P2P non-structurés ne font pas correspondre des pairs et des ressources de façon stricte, du à l'absence d'un espace commun d'identificateurs, ou tout simplement à l'absence des identificateurs au sens propre. L'algorithme de recherche est aléatoire dans le sens où la recherche d'une donnée n'a pas un but préétabli qui puisse définir sa finalisation. Les connexions entre les pairs sont aussi aléatoires, parce qu'elles sont établies par convenance ou sans aucune règle, n'ayant pas *a priori* de relations prédéfinies ou nécessaires entre les pairs.

On distingue plusieurs types de P2P non-structurés [16] en fonction de leur mode de fonctionnement.

- **P2P hybrides (Napster)**

Dans cette architecture, un index central est utilisé pour retrouver d'abord le nœud (son adresse IP et le port de communication) contenant un objet donné dans le réseau. Par la suite, la transmission des données, entre le nœud requérant et le nœud sollicité, s'effectue sans l'intervention de l'index. Un nœud, dés qu'il se connecte sur le réseau obtient l'adresse du nœud stockant l'index central. On parle d'architecture centralisée. Ce type de réseau est très sensible aux pannes et ne passe pas à l'échelle.

- **P2P purs (Gnutella)**

C'est une architecture décentralisée très robuste où chaque nœud construit son propre index de ses données. Tous les nœuds sont équivalents en termes de comportement. La recherche d'une information est simple : elle s'effectue par diffusion des requêtes à travers le réseau, chaque pair exécute la requête puis la transmet à ses voisins. La diffusion des messages lors des requêtes surcharge le réseau et empêche cette architecture de passer à l'échelle.

- **P2P hiérarchiques ou « Super-peers » (KaZaa)**

C'est une architecture décentralisée qui prend en compte l'hétérogénéité de la bande passante des différents nœuds. Ainsi tous les nœuds ne jouent plus les mêmes rôles. En effet les nœuds dotés d'une très grande bande passante sont organisés en P2P purs : les **super-peers.**

Les nœuds avec faible bande passante sont rattachés en mode client-serveur à un super-peer (**cluster**). Chaque super-peer possède un index des ressources de son cluster.
Cela ne va pas sans problème car il introduit de nouveau la sensibilité aux fautes : si un super-peer est en panne cela entraîne l'inaccessibilité de tous les nœuds de son cluster. Mais ce problème est facilement résolu en utilisant une redondance des super-peers qui diminue la charge agrégée des clusters et accélère les recherches.

L'avantage principal des systèmes non structurés est la simplicité de leur mise en œuvre. Cependant les recherches s'effectuent avec un O (n) messages vers les autres nœuds.

### I.2.2.2. Les P2P structurés

Les systèmes P2P structurés possèdent un algorithme de recherche pour lequel une ressource donnée est assignée de façon univoque à un pair, (pour un état donné du système.) L'assignation est faite en minimisant une métrique prédéfinie sur un espace numérique d'identificateurs partagé par les pairs et les ressources. L'algorithme de recherche est complètement déterministe, et les liens entre les pairs sont faits suivant des règles bien définies. L'infrastructure de recherche marche comme un Tableau de Hachage Distribué ou THD (*Distributed Hash Table*, DHT) implémenté à travers les pairs. Les deux types de P2P structurés les plus utilisés actuellement sont Chord [14] et CAN [15] que nous allons décrire brièvement.

- **Chord**

C'est un protocole qui utilise une table de hachage distribuée (Distributed Hash Table : DHT). Les nœuds sont répartis virtuellement sur un anneau. Il est caractérisé par sa simplicité, sa performance et son exactitude. Chord n'utilise que O (log n) nœuds pour router une requête et O (log n) informations à propos de ses voisins pour d'efficients routages.
Chord fonctionne en associant une clé à chaque objet. Alors étant donné une clé, il stocke le couple (clé, objet) sur un nœud spécifique grâce à une fonction de hachage. De la même manière on peut retrouver le nœud contenant un objet donné.

- **CAN**

Il s'appuie sur un espace cartésien virtuel à **d**-dimensions divisée en plusieurs cellules (zones). Chaque zone est contrôlée par un noeud. L'espace virtuel est utilisé pour stocker tout couple (clé,

objet). Pour stocker ($K_1$, $V_1$), $K_1$ est mappé sur un point $P$ de l'espace en utilisant une fonction de hachage uniforme. Le couple correspondant ($K_1$, $V_1$) est alors stocké sur le nœud maintenant la zone où se trouve le point $P$. Pour retrouver un objet donné, on utilise la même fonction pour localiser le point $P$ sur lequel est stocké la clé correspondante. Chaque nœud maintient une table de routage contenant l'adresse IP et les coordonnées virtuelles des zones de ses voisins. Ceci pour router les requêtes à la bonne destination, ce qui ne demande qu'un parcours de O ($d\ n^{1/d}$) nœuds.

En résumé, même s'ils sont très complexes à mettre en œuvre, les P2P structurés passent facilement à l'échelle et donne de meilleures performances lors des recherches d'information.

## *II. Traitement des Transactions dans les P2PDB*

***Définition***

On définit les systèmes de bases de données P2P (P2P Data Base : P2PDB) comme un ensemble de pairs acceptant de partager des informations et des ressources de traitement des informations. Chaque pair contient une base de donnée locale plus un logiciel (middleware) permettant effectivement le partage d'informations.
Pour traiter cette partie, on va faire l'état de l'art avant de s'attaquer au traitement de transactions.

### II.1. Etat de l'art

Plusieurs travaux ont été effectués pour améliorer le traitement des transactions, la performance, la scalabilité et l'équilibrage des charges dans les bases de données distribuées. Cependant peu de sujets se sont orientés particulièrement au traitement des transactions dans les P2PDB. Néanmoins, nous donnerons l'état de l'art de ce domaine avec le peu de littérature disponible pour le moment. Nous présenterons aussi quelques algorithmes d'équilibrage de charges existants et des travaux qui ont été effectués sur le relâchement de la cohérence pour traiter des transactions.

#### II.1.1. Relâchement de la cohérence

La première forme de relâchement de la cohérence a été introduite avec le concept de *snapshot isolation* [16] où une nouvelle transaction ne voit que la dernière version de *snapshot* produit par l'ensemble des transactions qui ont validé avant son début. Une transaction ne peut

valider que si ses mises à jour ne rentrent pas en conflit avec les transactions concurrentes. *Snapshot* est bien utilisé car il est simple et améliore les performances du traitement en évitant les interblocages et les reprises de requêtes de lecture seule dans les bases de données répliquées. L'algorithme RSI-PC [17] utilise aussi le relâchement de la cohérence. C'est une solution, basée sur le concept de copie primaire, qui discerne les transactions de mises à jour des transactions en lecture seule. Les transactions de mises à jour sont toujours routées vers la réplique maîtresse tandis que les transactions en lecture seule sont dirigées vers les autres répliques qui se comportent comme des copies en lecture seule. Avec ces mécanismes de traitement des transactions, plus connus sous le nom de réplication asynchrone, l'incohérence persiste jusqu'à la réconciliation, période durant laquelle on fait converger l'ensemble des modifications qui ont été effectuées sur les données. Les spécifications du relâchement de la cohérence, sont largement étudiées dans la littérature, et peuvent être divisées en deux dimensions : spatiale et temporelle [18]. Un exemple de la dimension temporelle peut être trouvé dans [19] où l'accès à une copie cache (image) est tributaire d'une condition temporelle : par exemple l'intervalle de temps séparant la date de dernière modification de la copie accédée et celle de la copie maître. La dimension spatiale consiste à accorder une quantité de modifications qui peut séparer les données accédées des données réellement stockées sur, par exemple, la copie maître. Dans le modèle de cohérence continue présenté dans [20], toutes les deux dimensions sont contrôlées. Chaque nœud propage ses modifications vers les autres nœuds, ce qui leur permet de maintenir un niveau de cohérence prédéfini pour chaque dimension. Ce faisant, chaque requête peut être envoyée sur un nœud satisfaisant le niveau de cohérence dans l'optique d'équilibrer les charges.

Le fonctionnement de ces algorithmes, qui viennent d'être présentés, ne laisse pas intact l'autonomie des applications et des bases de données. L'environnement de travail est quasi statique. Notre étude veut faire bénéficier les performances du relâchement de la cohérence aux systèmes P2P. Pour ce faire des mutations seront effectuées dans l'ordre de repousser leurs limites et de prendre en compte l'autonomie et le dynamisme des nœuds ainsi que leur nombre élevé.

### II.1.2. Equilibrage des charges et routage dans les P2P

L'équilibrage des charges dans les systèmes P2P est, au même titre que la localisation des données (objets), l'un des sujets qui a intéressé bon nombre de chercheurs. Chord [14] utilise une

fonction de hachage distribuée pour stocker les différent nœuds du système sur un ring ainsi que les données. Ceci garantit de manière naturelle un certain niveau d'équilibrage des charges. Chord assure, avec une très grande probabilité, que le nombre d'objets par nœud ne dépasse un certain seuil. Mais ceci requiert que tous les nœuds soient homogènes, que tous les objets aient la même taille et que les identifiants des objets soient répartis uniformément sur l'espace d'adressage. CFS [21] prend en compte l'hétérogénéité des nœuds en allouant à chacun d'entre eux un nombre de *virtual server* (découpage logique de la charge d'un nœud) proportionnellement à leur capacité de stockage. Si un nœud devient surchargé, CFS lui retire quelques *virtual server* afin de garantir l'équilibre des charges. [22] utilise le même concept de *virtual server* et propose trois algorithmes qui se diffèrent principalement de la quantité d'information utilisée pour décider comment rééquilibrer la charge. Le premier algorithme nommé One-to-One, choisit par hasard deux nœuds dans le système. Si l'un d'entre eux est surchargé alors que l'autre a une charge en dessous d'un seuil, on initie un transfert de virtuel server. Contrairement au premier, le deuxième One-to-Many permet à un nœud surchargé d'effectuer un transfert vers plusieurs autres nœuds moins chargés. Alors que le dernier, Many-to-Many permet à plusieurs nœuds surchargés de s'échanger avec plusieurs nœuds de charges faibles. Le partitionnement horizontal par intervalle d'une relation dynamique dans les systèmes P2P permettent de répondre plus rapidement aux requêtes par intervalle. Si la répartition des charges et des requêtes est uniforme, les nœuds stockeront la même quantité d'information. Dans le cas contraire, l'équilibrage des charges devient indispensable. Ainsi [13] utilise principalement deux opérations *Nbradjust* et *Reorder* pour garantir l'équilibrage des charges dans de tel cas. L'algorithme que propose [13] est asymptotiquement efficient et utilise une structure de données nommée *skip graph* [26] pour faciliter le routage des informations. Ce routage, joue un rôle essentiel dans les algorithmes d'équilibrage, c'est pour cela des études lui ont été consacré.

En fait, le routage des informations dans un système P2P est étroitement lié avec la structure du système. Dans les P2P non structurés les informations sont routées par diffusion, ce qui génère un très grand trafic. Pour éviter ce problème, les P2P structurés utilisent des tables de hachages distribuées pour localiser les nœuds et y router les données. Il s'agit de Chord où chaque nœud maintient une table de routage avec O(log N) et de CAN dont la table contient O(d) entrées avec d la dimension de l'espace d'adressage. De même Pastry [27] présente un algorithme de localisation et de routage générique des données. Pastry se repose entièrement sur une auto-

organisation des nœuds. Pastry est complètement décentralisé, scalable et fiable. Le routage d'une information se fait suivant l'ordre de O(log N) avec N le nombre de nœud du réseau. Tous les algorithmes d'équilibrage que nous avons présenté, garantissent l'uniformité de la répartition des charges en faisant déplacer des données d'un nœud supposé surchargé vers d'autres nœuds moins chargés. Ces algorithmes attendent d'abord que le problème de déséquilibrage des charges se pose pour solutionner ce dernier, on peut les qualifier de «médecins après la mort ». Nous allons, par contre spécifier un algorithme qui permet de prévenir ce déséquilibre des charges en répartissant le traitement des requêtes sur les différents nœuds du système. En d'autres termes nous allons prévenir les *skews execution* [13] (mal répartition des tâches de traitements sur les différents pairs) qui engendrent les *skews data* [13] (déséquilibre des charges). D'autres parts, les mécanismes de routage utilisés par ces algorithmes s'appuient en général sur la technique de hachage distribuée qui fonctionne que sur les P2P structurés. Notre algorithme se veut plus générique en utilisant un routage qui s'appuie sur la sémantique des données que stockent les nœuds et la sémantique des requêtes à exécuter sur ces données.

### II.1.3. Gestion de l'acidité des transactions

Classiquement, garantir l'acidité des transactions revient à gérer l'atomicité d'exécution ou contrôle de concurrence et l'atomicité en cas de panne ou la consistance.

Les travaux de [2] se sont orientés particulièrement à cette fin en proposant un protocole qui assure la préservation des propriétés ACID dans les P2PBD.

L'essentiel du travail de [2] s'appuie sur l'algorithme *epidemic* [3]. La durabilité est assurée grâce à la *réplication dynamique* [2], l'atomicité via la version hiérarchisée du protocole *3-PC*, l'isolation par l'algorithme *Dynamic voting* [5] et enfin la cohérence par un nombre suffisant de répliques permettant la vérification des contraintes d'intégrité.

Force est de reconnaître, que les mécanismes utilisés peuvent être améliorés, ou même remplacés par d'autres, dans l'objectif de rendre plus optimale, l'exécution des transactions dans un environnement P2P. Par exemple plusieurs études ont été menées dans [6] et [11] en vue d'évaluer et d'augmenter les performances de l'algorithme *Dynamic voting.* De même, dans [8] propose un algorithme pour garantir l'atomicité en cas de panne (atomicité et durabilité) en structurant sous forme d'arborescence les nœuds actifs afin de faciliter leur communication : il

agrège les acquittements, en opposition avec la plupart des protocoles de validation atomique (2-PC, 3-PC, etc.) qui exigent que chaque action, exécutée par un participant, soit aussitôt acquittée.

Comme on vient de le voir, le fonctionnement de ces protocoles requiert une forte synchronisation des nœuds impliqués dans une transaction durant toute son exécution. Ceci, dans un environnement P2P où la connexion est assez instable, cause de sérieux problèmes à savoir des reprises incessantes de transactions. Nous introduisons la réplication optimiste [9, 10] dans laquelle les transactions sont validées localement, ainsi la forte synchronisation entre copies durant l'exécution de la transaction n'est plus nécessaire. Toutefois les répliques peuvent avoir des valeurs différentes jusqu'à la réconciliation où l'état cohérent du système est rétabli. Mais cette divergence doit être contrôlée pour au moins deux raisons. Premièrement, puisque la synchronisation consiste à produire une séquence de traitements unique à partir de plusieurs séquences divergentes, plus la divergence est importante, plus la réconciliation est difficile. La deuxième raison est que les applications n'ont pas toujours nécessairement besoin de lire des données parfaitement cohérentes, et peuvent tolérer des incohérences.

La réplication optimiste peut utiliser un modèle client/serveur [24] ou un modèle P2P [23]. Le modèle client serveur coïncide en fait avec la réplication asymétrique décrite dans le chapitre 2. Dans ce modèle la panne du serveur maître entraîne une paralysie totale du système. Pour pallier à cette insuffisance, on peut répliquer le serveur maître mais ceci ne résout pas totalement le problème. Par contre dans le modèle P2P, tout nœud peut communiquer ses mises à jour directement à tout autre nœud. Ce modèle ne restreint nullement l'autonomie du système. Par conséquent on peut utiliser n'importe quelle stratégie pour router les mises à jour et que les nœuds peuvent se défaire du système selon leur convenance. Les résultats de la simulation dans [25] démontre que le modèle P2P augmente la vitesse de propagation des mises à jour à travers les différentes répliques mais diminue aussi la probabilité de tomber sur une version des données périmée.

Le reste du document est organisé comme suit. Nous définissons d'abord les concepts principaux que sont la fraîcheur d'une réplique, la politique d'une transaction et le plan d'exécution d'une transaction. Ensuite nous présentons l'architecture de routage, puis nos algorithmes pour le routage des transactions. Les résultats expérimentaux valideront enfin l'efficacité de nos algorithmes de routage.

## II.2. Traitement des transactions P2P

### II.2.1. Concepts de base

L'objectif principal de notre étude est d'équilibrer les charges des mises à jour ; ce qui doit se faire sans modifications des caractéristiques de base des P2P. Pour cela on réplique toute la base de données dans plusieurs nœuds du système, autrement dit on ne peut répliquer une tierce partie de la base.

Avec la réplication optimiste, plusieurs répliques peuvent avoir des états différents à un instant donné, car elles n'ont pas encore atteint un état complètement cohérent, c'est-à-dire l'état obtenu après la réconciliation. De plus, les transactions à traiter sur le système peuvent avoir des exigences de cohérence différentes. Afin de pouvoir router les transactions en contrôlant la précision, nous nous référons à l'algorithme de routage décrit dans [1] pour définir la précision d'une réplique de la base de données, puis les concepts de politique de transaction et de plan d'exécution.

#### a. Fraîcheur d'une réplique

Intuitivement, 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. Alors la fraîcheur est dite maximale si la quantité de changements vaut zéro, c'est-à-dire que son état est complètement cohérent. Un changement consiste à mettre à jour une relation. Par conséquent, nous définissons la quantité de changements au niveau d'une relation. Soit *R* une relation modifiée par *T*, nous appelons *Change (T,R)* le nombre maximum de n-uplets modifiés par *T*. La fraîcheur d'une réplique *R* est représentée par la quantité de changements effectuée sur toutes les autres répliques sauf *R*. Soit *TR* l'ensemble des transactions qui ont modifié ces autres répliques, la fraîcheur *F* de R peut donc être définie comme étant la somme des changements effectués par *TR* :

$F(R) = \sum i\ (Change(Ti,R)\ |Ti \in TR)$

Ainsi, nous définissons la fraîcheur *F* d'un noeud *N*, possédant la réplique d'une base de données avec *n* relations, par :

$F(N) = \{\ (Ri,F(Ri))\ |i=1,n\}$

#### b. Politique d'une transaction

Le routeur utilise la politique d'une transaction pour contrôler l'exécution d'une transaction avec un niveau de fraîcheur requis. Afin de préserver l'autonomie d'une base de données, nous définissons la politique d'une transaction basée seulement sur les transactions et les relations. Nous ne pouvons pas utiliser un niveau plus fin de granularité car les bases de données et les applications ne divulguent pas au routeur le code source détaillé des transactions. La politique d'une transaction décrit les exigences d'une transaction et ses effets sur la base de données. La politique d'une transaction décrit tout d'abord l'état requis du système P2PDB avant d'exécuter une transaction. Nous définissons l'état du système pour une transaction *T* en fonction de la fraîcheur des données du P2PDB, et de l'exécution des transactions conflictuelles avec *T*. L'exigence de fraîcheur est la fraîcheur minimale de toutes les relations accédées par la transaction. Soit l'ensemble des relations accédées par T, nous définissons la fraîcheur *F* de T par:

$F(T) = \{(Ri, F(Ri)) \mid Ri \in RT\}$

L'exigence concernant les transactions en conflit avec T a pour but d'empêcher l'occurrence de conflits insolubles. Soit *Precede(T)* l'ensemble des transactions qui doivent précéder la transaction, on a :

$Precede(T) = \{Ti \mid (Ti, T) \text{ non commutatives}\}$

La politique d'une transaction décrit aussi les changements effectués par la transaction pour toutes les relations modifiées par T. Soit l'ensemble des relations modifiées par *T*, nous définissons les changements de *T* par :

$Change(T) = \{(Ri , Change(T, Ri)) \mid Ri \in R'T\}$

Ainsi, nous définissons la politique d'une transaction (TP) par :

$TP(T) = (F(T), Precede(T), Change(T))$

**c. Plan d'exécution d'une transaction**

Un plan d'exécution d'une transaction décrit comment le routeur exécute une transaction *T* dans le système. Le plan décrit la réplique *N* où *T* sera exécutée ainsi que les traitements pré requis nécessaires pour atteindre l'état du système requis par *T*. Soit *Sync* la séquence de toutes les transactions devant précéder T et celles qui rendront la réplique suffisamment fraîche, nous définissons le plan d'exécution, d'une transaction (TEP) par :

$TEP(T) = (N, Sync(T))$

**Exemple**

Pour illustrer comment le routage des transactions en contrôlant la fraîcheur améliore l'équilibrage de charge, nous reprendrons l'exemple décrit dans [1] où l'on considère la relation Stock(item, quantity, threshold).

La transaction *D* décrémente la quantité d'article *id* par *q* :

**procedure** D(id, q): **update** Stock **set** quantity = quantity – q **where** item = id;

Nous supposons maintenant la requête *Q* donnant la liste des articles à renouveler :

**select** item **from** Stock **where** quantity < threshold

On mesure les changements effectués par une transaction ainsi que la fraîcheur comme étant le nombre de n-uplets modifiés. *D* modifie un seul n-uplet et requiert une fraîcheur de 2 (c'est-à-dire que D peut être exécutée sur une réplique qui n'a pas encore reçu les 2 dernières transactions D exécutées sur les autres répliques), et Q n'a aucune exigence de fraîcheur. Soient une application de vente de produits qui exécute une séquence de transactions D, et une application de réapprovisionnement qui exécute une séquence de requête *Q*. La charge est composée de deux applications de vente et deux applications de réapprovisionnement s'exécutant en concurrence. La relation Stock est répliquée sur les noeuds N1 et N2. La figure 1 illustre les transactions à exécuter.

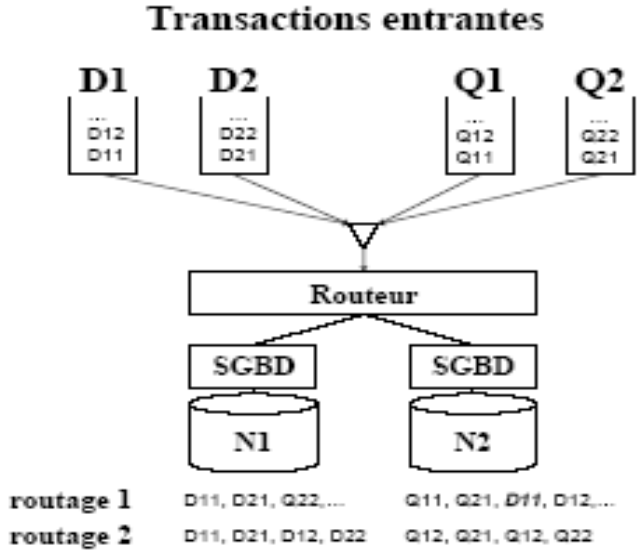


Figure 16 : Exemple de routage de transaction

En supposant que N1 et N2 peuvent traiter une requête en même temps, on spécifie deux stratégies de routage. La première stratégie consiste à choisir le noeud le moins chargé pour exécuter chaque nouvelle transaction. Donc, en moyenne, N1 et N2 traitent autant de transactions

D et Q. Par exemple, au départ, D11 est exécutée sur N1, puis Q11 sur N2, puis D21 sur N1 concurremment avec D11 en cours d'exécution, puis Q21 sur N2 concurremment avec Q11 en cours d'exécution. A cause de l'exigence de fraîcheur de D, lorsque D12 arrive, si le noeud le moins chargé est N2, le routeur commence par propager D11 sur N2 avant de traiter D12 afin de rendre N2 suffisamment frais. Après un certain temps, les deux noeuds seront complètement frais au surcoût de traiter toutes les ventes D sur tous les deux noeuds. Cette stratégie produit des temps de réponse élevés à cause de la surcharge pour traiter toutes les ventes deux fois. Pour éviter cette surcharge, la deuxième stratégie consiste à choisir le noeud dont la fraîcheur est aussi proche que possible des exigences de la transaction. Donc, lorsque D12 arrive, le routeur choisit N1 car il est suffisamment frais pour traiter D12, tandis que le traitement sur N2 aurait nécessité de propager D11. Par conséquent, après un certain temps, N1 se retrouve dédié aux ventes D et N2 aux réapprovisionnements Q. Donc, chaque vente est traitée une seule fois et le temps de réponse de D et Q est amélioré par rapport à la stratégie précédente. L'inconvénient secondaire est que N2 devient de moins en moins frais car il n'a reçu aucune vente D. Cet exemple illustre l'importance de connaître le coût de rafraîchissement d'un noeud, afin d'effectuer le routage adéquat.

**Remarque :** Dans [1] les nœuds du système sont supposés assez fiables, pour ne pas tomber en panne durant toute l'exécution de la séquence de transactions. Dans un contexte P2P, l'instabilité de la connexion peut engendrer des problèmes insolubles. Par exemple, avec la seconde stratégie, si le noeud N1 tombe en panne après avoir exécutée D12, alors D22 ne peut être exécutée car il n'y a pas de nœud suffisamment frais et que le seul nœud N1 susceptible de remettre la fraîcheur requise (par propagation des dernières mises à jour) est injoignable. De l'autre coté, l'application de réapprovisionnement risque de générer de faux résultats.

Une solution à ce problème est de prévoir la panne d'un nœud, recevant des mises à jour, en contrôlant sa durée de vie. En fait, comme la durée de vie moyenne (en anglais Time To Life : TTL) d'un pair peut être approximée (2.9 heures pour Gnutella et 13.8 heures pour SETI@home), on définit un intervalle de temps pendant lequel, le nœud peut exécuter une transaction. Si cet intervalle de temps est épuisé, alors le routeur ne doit plus le considérer comme candidat pour le traitement d'une nouvelle transaction. En outre, le nœud en coopération avec le routeur doit propager toutes les mises à jour récentes par la méthode ***push.*** On augmente ainsi, la tolérance aux pannes du système.

### II.2.2. Description de l'architecture de routage

Dans les systèmes P2PDB, chaque pair est libre d'émettre un nombre illimité de transactions simultanées, du fait de son autonomie. Les applications initiatrices de ces transactions, ignorent à priori tous mécanismes d'équilibrage de charge. Il est indispensable dans ce cas de définir au niveau du middleware une logique qu'on appellera routeur afin d'assurer cet équilibrage.

Les principaux rôles du routeur sont :

- Déterminer quels sont les nœuds capables de traiter une transaction, éviter les conflits insolubles entre transactions concurrentes et assurer une bonne cohérence des données.
- Si plusieurs nœuds peuvent traiter une transaction, le routeur doit choisir celui qui permet l'exécution la plus efficace.
- A cause de la réplication, le routeur doit propager les mises à jour sur les autres répliques. Cette synchronisation doit interférer le moins possible avec le traitement effectif des transactions afin de réduire la charge.
- Si le TTL d'un nœud est épuisé alors, le routeur doit être à même de suspendre temporairement le traitement des transactions, et diffuser ses derniers mises à jour.

Afin de mieux décrire notre routage, on spécifiera d'abord quelques hypothèses avant de présenter l'architecture de notre routeur ainsi que les différents modules qui le composent. Nous donnerons par la suite les différents protocoles utilisés pour assurer le traitement des transactions tout en prenant en compte les caractéristiques de l'environnement P2P.

#### II.2.2.1. Spécifications

- **Les données stockées sur chaque pair sont connues**

Chaque nœud peut indiquer la liste des relations qu'il contient, mais un nœud ne connaît pas les données des autres nœuds. Alors on fait la distinction entre une réplique accessible en lecture seule (*read only*) et une réplique accessible en lecture et écriture (réplique *read-write* appelée réplique maître). Une réplique en lecture seule ne sert que pour traiter des requêtes. On suppose qu'un nœud contenant une réplique maître connaît tous les autres nœuds qui possèdent aussi une réplique maître, mais il ne connaît pas tous les nœuds possédant une réplique en lecture seule.

- **Certaines transactions ne font que des lectures (requêtes)**

Distinguer les requêtes des transactions, permet de mieux gérer les potentialités du système. Par exemple on peut sélectionner un nœud dont son TTL tend vers 0 ou bien un nœud libre (sans transaction) pour exécuter une requête. On ne routera alors les requêtes que vers les *read only*, en réservant les *read-write* pour les opérations de mises à jour.

- **Les données accédées par une transaction sont connues**

Ceci permet au routeur, en se basant sur la première hypothèse, de sélectionner facilement un nombre limité de pairs candidats pouvant exécuter la transaction. Ce qui peut diminuer largement le temps d'exécution totale des transactions.

- **Les transactions accèdent souvent à des données disjointes**

Ceci permet de réduire la synchronisation avant le début d'une transaction car *Precede(T)* est minimal. Par ailleurs les risques d'incohérences sont diminués.

- **Chaque pair peut traiter plusieurs transactions simultanées**

Avec la précédente hypothèse, on peut avoir un grand nombre de transactions simultanément (maximum N) sur un nœud afin de réduire le temps moyen d'exécution d'une transaction.

### II.2.2.2. Architecture du routeur

Pour permettre à notre architecture de routage de passer à l'échelle en évitant des points de congestion, notre routeur est réparti sur l'ensemble des nœuds. Ainsi chaque nœud va implémenter une instance du routeur (Router Instance : *RI*).

Selon le niveau d'abstraction que l'on se donne, notre architecture se présente sous deux formes.

Selon les utilisateurs (applications), le routeur est une entité globale via laquelle toutes les transactions transitent pour être exécutées. La figure 17 illustre l'architecture globale du routage.

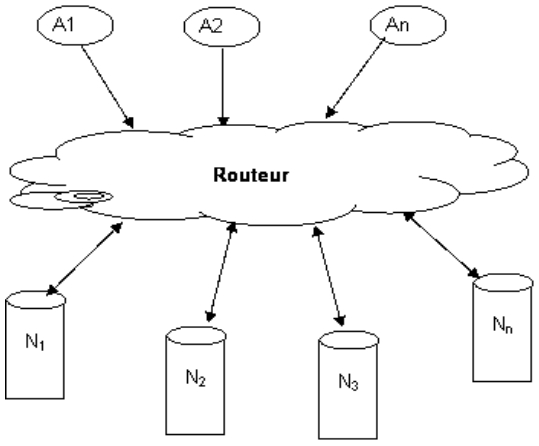


Figure 17 : Architecture globale du routage

A un niveau d'abstraction plus élevé (du point de vue des pairs), le routeur est un ensemble de composants qui coopèrent pour assurer le bon traitement des transactions. Par exemple soit une classe Router, chaque pair va instancier la classe Router dans son environnement. Les pairs peuvent se communiquer et donner leur état (disponible ou non pour traiter une transaction), ce qui génère le comportement global du système.

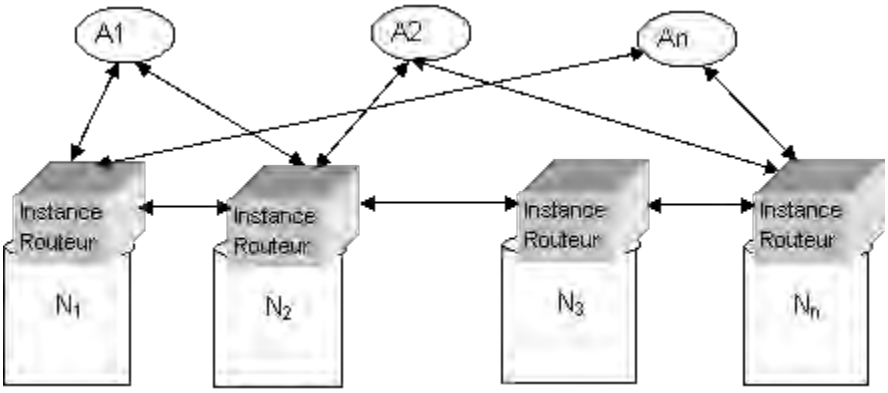


Figure 18 : Architecture détaillée du routage

Chaque *RI* d'un nœud stockant une relation *R* doit permettre l'accès aux autres répliques de *R*, ainsi il gère une table de routage associée à chaque relation, structurée comme suit :

| $N_1$ | $N_2$ | $N_k$ | ..... | $N_{n-1}$ | $N_n$ |
|---|---|---|---|---|---|
| (U,R) | (U,R) | (U,R) | ..... | (R,R) | (R,R) |

Figure 19 : Structure d'une table de routage

Les Ni désignent les nœuds stockant une réplique de la relation R et on suppose que le nombre de répliques est au maximum N. Les couples (U,R) représentent les répliques maîtres et les (R,R) représentent les répliques à lecture seule. La figure suivante montre l'architecture d'un routeur interface avec ses différentes composantes.

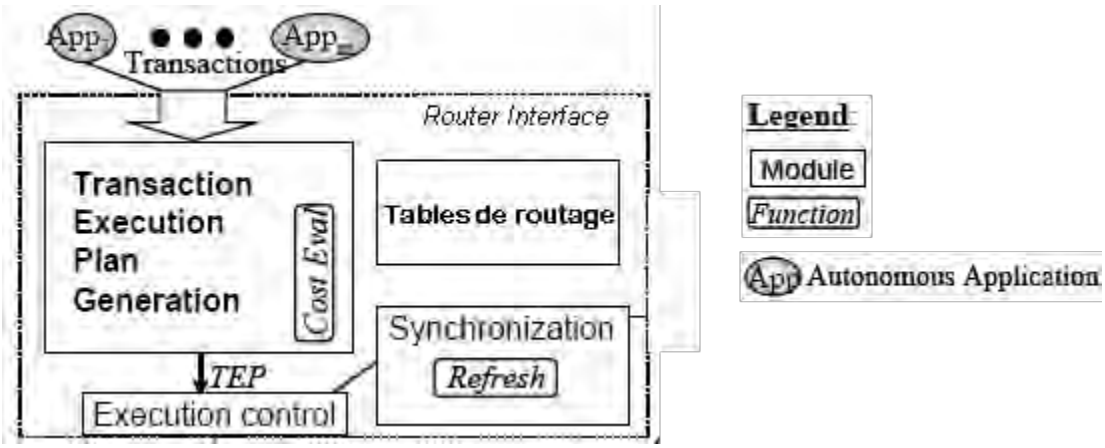


Figure 20 : Architecture de RI

**a. Gestionnaire des politiques de transactions**

Pour chaque nouvelle transaction T, le routeur génère une politique de transaction (TP). Après avoir déterminer les exigences initiales des transactions, à partir des règles d'exécution, il calcule la fraîcheur minimale requise. Par la suite, il recherche les transactions devant précéder T pour éviter les conflits insolubles et calcule via Change (T) les effets de l'exécution de T

**b. Module de synchronisation**

Il détermine le moment et la manière dont les mises à jour sont propagées vers les autres répliques. Cette propagation doit s'effectuer en surchargeant le moins possible le système, donc il faut éviter les synchronisations inutiles. Une première approche consiste à ne propager les mises à jour que si cela est nécessaire pour traiter une nouvelle transaction (Refresh). Ceci peut entraîner deux problèmes. Premièrement, si les nœuds restent assez longtemps sans recevoir des mises à jour, leur fraîcheur a tendance à diminuer continuellement. De ce fait toute nouvelle transaction exigeant une forte fraîcheur, va déclencher une très grande synchronisation, ce qui va ralentir le traitement de la transaction entrante. Deuxièmement, un nœud restant longtemps sans diffuser ses mises à jour, peut tomber en panne avant une synchronisation, ce qui peut entraîner une perte de données favorisant des risques intenses d'incohérence.

Ce faisant, les propagations sont déclenchées selon l'un des trois critères suivant :

1. si la fraîcheur d'un nœud est en dessous d'une certaine limite
2. si la charge d'un nœud est en dessous d'un certain seuil
3. si le TTL d'un nœud tend vers 0.

Les valeurs des seuils qui déclenchent la synchronisation dépendent de la charge du système et des applications.

### II.2.3. Algorithmes de routage des transactions

Dans cette partie nous allons proposer un ensemble de protocoles permettant de gérer les transactions mais aussi les interactions entre les différents RI du système. Nous ne manquerons point aussi de mettre en œuvre l'équilibrage des charges.

#### II.2.3.1. Routage des transactions et des requêtes

Lorsqu'un nœud reçoit une transaction *T*, celle-ci est interceptée par l'instance du routeur (*RI*) se trouvant sur la même machine que l'application et qui procède comme suit :

a. Grâce à son gestionnaire de politiques de transaction, le *RI* génère une politique : il s'agit de décrire les exigences de la transaction (Fraîcheur, Precede (T), Change (T))
b. L'instance sélectionne les nœuds contenant les données accédées par T en se basant sur la table de routage. Soit $R_T$ la relation accédée par *T*, *RI* exécute la fonction fing($U$,$R_T$) (ou fing($R$,$R_T$) s'il s'agit d'une requête) qui renvoie tous les nœuds contenant le couple ($U\ R_T$) (ou le couple ($R$,$R_T$) pour les requêtes).
c. Pour chaque nœud renvoyé par fing(), *RI* du nœud initiateur effectue les opérations suivantes :
    - Vérifier si les transactions, qui doivent précéder T, sont appliquées. Soit *Tprec*, l'ensemble des transactions de *Precede(T)* à exécuter sur N, si *Tprec* n'est pas vide, alors valider *Tprec* sur *N*.
    - Vérifier la fraîcheur du nœud. Si N n'est pas suffisamment frais, on ajuste la fraîcheur. Soit *Tsync*, l'ensemble minimal des transactions à exécuter sur *N* tel que *F(N)* atteigne *F(T) ;* Pour déterminer *Tsync*, on calcule d'abord *Ns*, l'état du nœud après la synchronisation. Premièrement, initialiser *Ns* avec l'état de *N*. Puis, itérer sur l'ensemble des exigences de fraîcheur. Pour chaque exigence, (1) rechercher la fraîcheur *f* de *Ns* qui n'atteint pas l'exigence, (2) tant que la fraîcheur de *Ns* demeure inférieure à l'exigence, (2.1) trouver une transaction *ts* à synchroniser pour améliorer *f*, (2.2) ajouter *ts* à *Tsync*, et mettre à jour la fraîcheur de *Ns* pour refléter la propagation de *ts* sur *Ns*. *Tsync* est ensuite ajoutée à *Tprec* pour obtenir le plan candidat : *TEP(T) = (N, Tprec U Tsync)*.

d. Après avoir généré tous les TEP, on utilise la fonction de coût définie dans [1] pour en choisir un.

### II.2.3.2. Gestion de la dynamicité

Afin de prévoir certaines pannes chaotiques causées par le départ d'un nœud dans une opération de synchronisation, on peut définir notre fonction fing() de sorte qu'elle ne renvoie que les jeunes nœuds (TTL > 0). On peut, soit envoyer un message aux répliques pour qu'elles fournissent leur TTL soit ajouter une troisième entrée dans le tableau de routage qui contiendra les dates de connexions des différentes répliques.

Cependant, si le nœud quitte le réseau sans diffuser les récentes mises à jour alors ces dernières sont indisponibles jusqu'à ce que le nœud rejoigne le réseau. Alors essayer de propager ces modifications sur les autres nœuds peut entraîner des cas insolubles car plusieurs transactions peuvent s'exécuter durant son absence. De ce fait, une possibilité est d'obliger le nœud à essayer de restaurer la plus récente sauvegarde sans les transactions précédant sa panne. Par la suite via la méthode pull il essaie de se connecter à un autre nœud dans l'optique de se conformer avec l'état actuel du système. Ceci est effectué par l'administrateur du nœud.

Néanmoins on peut essayer d'éviter qu'un nœud quitte le réseau alors qu'il n'a pas encore diffusé les transactions reçues. Pour ce faire, si le TTL d'un nœud tend vers zéro, ce dernier déclenchera une propagation des récentes transactions.

Par exemple on peut supposer que si la durée de vie restante d'un nœud vaut 1/3 * TTL, alors le nœud ne peut plus exécuter une nouvelle transaction et commence la diffusion des mises à jour vers les nœuds libres. Ceci va diminuer la probabilité d'avoir le premier cas de figure.

### II.2.3.3. Gestion de la fraîcheur des noeuds

Quand le *RI* initiateur vérifie la fraîcheur d'un nœud N, il envoie un message get(F). Le *RI* de ce dernier va se charger de trouver la fraîcheur en s'appuyant sur sa table de routage. Pour cela on peut avoir deux possibilités :

Soit on demande à toutes les répliques maîtres d'envoyer leurs mises à jour au nœud N. Ce qui peut générer un grand trafic dans le système.

Soit on contacte un seul nœud, qui doit être informé de toutes modifications de la relation R. Ceci fonctionne si le nœud reste connecté à tout moment. S'il tombe en panne toutes les modifications qu'il avait reçues sont perdues.
Pour éviter cette perte de modifications nous allons adapter la première stratégie dans le reste de notre travail.
Si la fraîcheur est trouvée le nœud N renvoie le résultat au nœud émetteur du message get(F).

### II.2.3.4. Gestion de la communication entre RI

Le processus de communication entre *RI* est un peu semblable à celui du protocole 2PC. Le *RI* d'un nœud initiateur d'une transaction joue le rôle de coordinateur durant toute l'exécution. Les nœuds stockant la relation accédée par la transaction jouent le rôle de participant. Le coordinateur envoie des messages aux participants pour demander des services, ces derniers les réalisent et lui envoient les résultats qu'il peut assembler et prendre la bonne décision. On peut utiliser des mécanismes de *timeout* pour limiter le temps d'exécution global.
En utilisant la table de routage, on construit facilement un groupe multicast afin de limiter le nombre de messages à diffuser sur le réseau.
Tous les *RI* sont similaires, ils implémentent les mêmes modules. De ce fait, le routeur fonctionne parallèlement sur tous les nœuds et peut gérer un nombre assez élevé de transactions simultanément. Aussi chaque RI doit être capable d'interpréter les différents messages que lui adressent les autres RI.
Pour s'assurer que deux transactions reliées par une contrainte de précédence seront traitées dans le même ordre sur toutes les répliques, on va estampiller les transactions.
En effet, l'estampille attribuée à une transaction est son horodate de lancement couplée avec le numéro du nœud sur lequel elle est lancée, ceci afin d'éviter l'égalité des estampilles pour deux transactions lancées au même instant sur deus sites distincts.
Alors avant d'exécuter une transaction *Ti*, lancée par le site de numéro i, le *RI* envoie une procédure *request (ts(Ti))* avec ts(Ti) l'horodate de la transaction pour vérifier s'il n'y a pas une autre transaction lancée au même instant . La fonction request est envoyée à toutes les autres répliques maîtres.
Si *request (ts(Ti))* = Ø alors la probabilité d'avoir un conflit de précédence est nulle et on peut exécuter Ti. Dans le cas contraire c'est-à-dire si *request (ts(Ti))* = $\{(T_k) \mid k$ entier naturel$\}$, alors

s'il existe un $T_k$ qui doit précéder $Ti$, la transaction $Ti$ est annulée, sinon elle est exécutée. L'algorithme est le suivant :

```
Si Detection_conflit = vrai alors annuler Ti
Sinon exécuter Ti

//Procédure de détection de conflit
Detection_conflit (ts(Ti))  {
Si request (ts(Ti)) = Ø alors  retourner faux   // il n'y a
pas de conflit
Sinon
Pour chaque (Tk = request (ts(Ti))  ) faire
Si Tk < Ti alors retourner vraie     // existence de conflit
Fin pour
Retourner faux
}
```

#### II.2.3.5.Routage des transactions de synchronisation

Comme le temps de synchronisation peut augmenter fortement le temps de réponse total de la transaction, le module de synchronisation propage les transactions par anticipation afin de préparer certains noeuds en vue des exécutions à venir. Pour limiter la surcharge, la synchronisation s'effectue seulement sur les noeuds libres (sans transaction en cours).

Ainsi, chaque RI en collaboration avec le module de synchronisation, essaie de propager les mises à jour situées sur le nœud qu'il manage. Pour cela, il scrute la table de routage afin de recenser les nœuds libres, et par la méthode ***push*** il diffuse les mises à jour.

Le RI d'un nœud peut utiliser la méthode ***pull*** pour réceptionner les mises à jour, si l'un des critères cités dans la section « Module de synchronisation » se présente, mais seul les nœuds libre peuvent répondre à son appel.

#### II.2.3.6. Equilibrage de la charge

Un routeur ne connaît pas la charge de tous les nœuds. Par contre un nœud peut, par exemple, indiquer un niveau de disponibilité pour traiter les requêtes et les transactions. De là on voit que l'essentiel de l'équilibrage des charges est effectué durant le choix du nœud, sur lequel traiter la transaction, qui se fait en choisissant le nœud le moins chargé.

## II.3.Validation

Dans cette section, nous allons valider notre approche à travers la simulation. Nous décrivons tout d'abord l'outil utilisé à savoir le simulateur Peersim. Puis nous allons aborder l'implémentation des protocoles utilisés pour traiter les transactions en prenant le soin de présenter notre modèle d'exécution. Enfin à partir des expériences nous montrons que notre approche peut bel et bien tenir.

### II.3.1. Le simulateur Peersim

Peersim est un logiciel libre, écrit en java, destiné à simuler le mode de fonctionnement des réseaux P2P. Il permet alors de concevoir et de tester toutes sortes d'algorithmes des P2P dans un environnement dynamique. Le code source peut être télécharger à partir de http://peersim.sf.net .

Ce logiciel est caractérisé essentiellement par :

- La possibilité d'avoir une très grande échelle (plus de un million de nœuds)
- Une très grande dynamicité (arrivée/départ) des nœuds.
- Une architecture ouverte et composée de modules.

Cependant, Peersim ignore les détails de l'implémentation de la pile de protocole TCP/IP et ne modélise pas les messages échangés entre protocole, ceci dans l'optique de diminuer la charge CPU et la grande quantité de mémoire que la communication entre protocoles requiert. Néanmoins le développeur peut penser à les implémenter.

Essentiellement, Peersim supporte deux types de simulation à savoir :

- Simulation par cycle : La simulation s'effectue dans un ordre séquentiel et dans chaque cycle, chaque protocole peut exécuter son comportement.
- Simulation par évènement : elle supporte la concurrence ; un ensemble d'évènements (message) est planifié et les protocoles d'un nœud sont exécutés en fonction des évènements survenus.

Nous allons utiliser par la suite la simulation par cycle, plus facile à mettre en œuvre.

Les entités de bases utilisées dans une simulation par Peersim sont :

- **Les nœuds (nodes)**

Ce sont les éléments de bases qui représentent les différents pairs qui forment le réseau. C'est une interface implémentant un ensemble de méthodes permettant d'accéder à ces voisins, de référencer les protocoles, son index dans le réseau, …

- **Le réseau (Network)**

Il est constitué par l'ensemble des pairs (nœuds) et représente tous les participants.

- **Planificateur (Scheduler)**

Il détermine l'ordre dans lequel les différents composants du système seront exécutés. Par exemple avant d'exécuter un protocole, on effectue d'abord l'initialisation du système, qui est assurée par les dynamics. Cependant, le développeur peut définir la séquence dans laquelle les composants seront exécutés et la durée de chaque cycle.

L'architecture de Peersim est constituée de plusieurs composants. On peut ajouter de nouveaux composants dans Peersim. Les principaux composants sont les *protocols*, *Dynamiques* (et les initialiseurs) et les *observers*.

- Protocols

Il constitue le cœur de l'algorithme, conçu pour effectuer une tache précise. Plusieurs protocoles peuvent s'exécuter sur chaque nœud et chaque protocole possède un identificateur unique ID. Chaque protocole utilise seulement les informations locales au nœud sur lequel il tourne.

- Dynamics

Ils sont utilisés pour assurer la dynamicité des nœuds (arrivée/départ) et ils ont une vision globale du système. Ils peuvent être également utilisés comme des initialiseurs (i.e. pour donner l'état du système au début de la simulation). Leur nombre au cours d'une simulation n'est pas limité.

- Observers

Ils enregistrent les informations issues de la simulation et à l'image des *dynamics* ont une vision complète du système. Les résultats gardés par les observers peuvent être affichés directement à l'écran ou bien redirigé vers un fichier pour en faire d'autres usages.

Notons au passage l'existence d'une interface importante *Linkable* dont tout noeud l'implémentant peut accéder aux informations stockées par ses voisins.

Tous ces composants qu'on vienne de décrire peuvent être considérés comme des fonctionnalités de bas niveau. La principale composante fonctionnant au haut niveau est le fichier de configuration.

- Fichier de configuration

C'est un fichier texte qui permette d'effectuer les tâches suivantes :

- Que chaque composant de Peersim soit paramétrable en fonction des besoins.
- Définir la taille du réseau et la durée de la simulation (en termes cycles)
- Le nombre d'expérimentations voulu
- La manière dont le réseau évolue.

### II.3.2. Modèle d'exécution

Notre objectif est d'assurer le traitement des transactions tout en garantissant l'équilibrage de la charge des mises à jours. Pour cela, on va utiliser la réplication optimiste dans laquelle les transactions sont validées localement, ainsi la forte synchronisation entre copies durant l'exécution de la transaction (cas de ROWA), n'est plus nécessaire. La situation peut se résumer comme suit : étant donné l'état du système (charge des nœuds, transactions courantes, bande passante disponible, …), la répartition des données et les exigences d'une transaction T, choisir le nœud sur lequel l'exécution de T est optimale. Les étapes du traitement sont :

1. Des applications clientes envoient des transactions (et des requêtes) vers les nœuds. Une application envoie soit une seule transaction soit une seule requête. L'application attend de recevoir la réponse de la transaction (ou de la requête) avant d'envoyer la suivante.
2. Chaque nœud peut recevoir un nombre quelconque de requêtes (ou de transactions).
3. Une transaction (ou requête) est simple (i.e. courte) si elle n'accède qu'à une seule donnée, elle est complexe (i.e. longue) dans le cas contraire.
4. Toute transaction (ou requête) émise par une application, est interceptée pour être redirigée vers le routeur RI.
5. Le routeur choisit, en fonction des exigences requises par la transaction, le nœud sur lequel effectuer le traitement. Pour être plus rigoureux, nous allons supposer que les transactions requièrent une fraîcheur maximale, ce qui ne sera pas le cas avec les requêtes.
6. Pour simplifier notre modèle, on suppose, qu'au cours d'un cycle, un nœud peut exécuter toutes les transactions qui sont dans sa file d'attente en utilisant la stratégie FIFO.
7. La terminaison de la transaction est enfin envoyée à l'application. Cette terminaison peut être positive (commit) ou négative (abort).
8. Les mises à jour sont diffusées asynchroniquement aux autres copies suivant une stratégie réduisant la surcharge du système et ne compromettent pas la cohérence des données.

### II.3.3 Implémentation des protocoles

Pour simuler le fonctionnement de notre algorithme, on a installé Peersim sur Linux puis on utilise les étapes suivantes ; l'étape d'initialisation du réseau, celle de la mise en œuvre de la charge transactionnelle et enfin celle du routage proprement dit. Pour ce faire on a mis en place trois classes à savoir **LbpInitialize**r, **LbRprotocol** et **LbRObserver.**

1. **La classe LbpInitializer**

La classe **LbpInitializer** va assurer les taches suivantes en utilisant quelques méthodes définit dans **LbRprotocol**:

- Mettre en place un tableau contenant les noeuds
- Désigner parmi les noeuds ceux qui sont maîtres et ceux qui sont en lecture seule.
- Pour chaque noeud maître implémenter une table contenant la localisation de tous les autres noeuds maîtres. Chaque noeud maître peut avoir dans sa table quelques noeuds en lecture seule.

Idem pour tous les noeuds en lecture seule. **define_transaction()** qui est implémentée dans la classe **LbRprotocol** dont on donnera les détails par la suite. Une fonction dénommée **Initialize_file()** sera utilisée pour définir une file d'attente des transactions au niveau de chaque nœud. A l'état initial cette file est vide.

Pour le moment on a supposé qu'un nœud ne peut traiter qu'une transaction à un instant donné.

2. **La classe LbRprotocol**

Cette classe renferme le noyau de notre simulation et comporte les attributs et méthodes suivants :

- Les attributs

*Boolean State* : cet attribut permet de définir si le nœud est maître (state = true) ou pas.

*Boolean transact_type* : détermine si on a une transaction ou requête

*Table_maitre* : tableau dynamique contenant les nœuds maîtres

*Table_slave* : tableau dynamique contenant les nœuds esclaves

*Descripteur* [][] : Ce tableau contient toutes les informations qui permettent de décrire une transaction (le coût, la fraîcheur, les données manipulées, ….)

*stamp_element* [][] : tableau contenant la date de modification d'un élément.

*Data_base* [][] : Tableau représentant la base de données.

- Méthodes

Pour simplifier la présentation des méthodes, on ne listera pas les méthodes qui font parti du package de Peersim, on considère que ces fonctions sont standards même si on y a apporté quelques modifications.

public void **setState()** définit l'état d'un nœud i.e. transforme un nœud esclave en maître.

public void **getState()** permet de savoir l'état d'un nœud

public void **setBase()** définit le schéma de la base et y insère des données aléatoires. La base sera répliquée sur tous les autres nœuds.

public void **Update_simple()** assure l'exécution d'une transaction

public void **Requete_simple()** assure l'exécution d'une requête

public Object **define_transaction()** : cette fonction permet de définir une transaction en définissant sa fraîcheur, son coût et sa précédence. Elle peut générer de manière aléatoire soit une transaction courte soit une transaction longue.

En fait, pour nous, une transaction a un descripteur représenté sous forme d'un tableau à deux dimensions et qui garde les informations suivantes.

Si Descripteur est le nom du tableau on a,

Descripteur[i][0] : qui permet de définir si la transaction est une requête (ou transaction).
L'élément i détermine la position de la transaction dans une file d'attente.

Descripteur[i][1] : contient le coût en nombre de cycles

Descripteur[i][2] : représente la fraîcheur

Descripteur[i][3] : le nombre de mises à jour qui doivent précéder la transaction

Descripteur[i][4] : la ligne de l'élément mis à jour

Descripteur[i][5] : la colonne de l'élément accédé

Voici encore quelques fonctions d'une forte utilité.

public void **Initialize_file()** définit une file d'attente sur un nœud

public void **send_transaction()** Si un nœud esclave reçoit une transaction, il l'envoie via cette fonction à un autre nœud.

public void **choisir_noeud().** Cette méthode permet de choisir un nœud sur lequel traiter une transaction. Elle utilise les trois méthodes suivantes.

public void **calcul_charge() :** évalue la charge ciblée d'un noeud

public void **appliquer_precede()** une fois un nœud choisi pour exécuter la transaction, cette fonction permet d'appliquer les transactions qui doivent la précéder.

public void **ajuster_fraicheur()** : ajuste la fraîcheur d'un élément en évaluant d'abord le nombre de modifications effectué sur cet élément dans les autres répliques.

public void **synchroniser() :** cette module permet d'effectuer les mises à jour de synchronisation.

3. **La classe LbRObserver**

Cette classe va permettre d'afficher les résultats de la simulation comme le temps d'exécution moyen. Les données récupérées à partir de cette classe seront utilisées dans Excel pour tracer les graphes.

### II.3.4 Expérimentation

Dans l'expérience suivante, nous montrons que notre algorithme passe bien à l'échelle. On mesure le temps nécessaire pour traiter toutes les transactions, quelle que soit la taille du réseau (20 noeuds, 100, 1000,…). Chaque nœud (maître ou esclave) a une file de capacité égale à 20, en d'autres termes, un nœud peut traiter entre 1 et 20 transactions (ou requêtes) au cours d'un cycle.

On mélange des transactions (ou requêtes) courtes et longues pour s'approcher plus de la réalité, et on considère que l'arrivée des transactions courtes et longues est aléatoire. Chaque transaction (ou requête) a une consommation différente de ressources. Nous observons que quel que soit le nombre de nœuds, après 4 cycles d'exécution le nombre de transactions restantes est inférieur au cinquième du nombre de transactions initial.

Et comme le montrent les figures suivantes, après le cycle 10 le nombre de transactions (ou requêtes) restantes est quasiment nul.

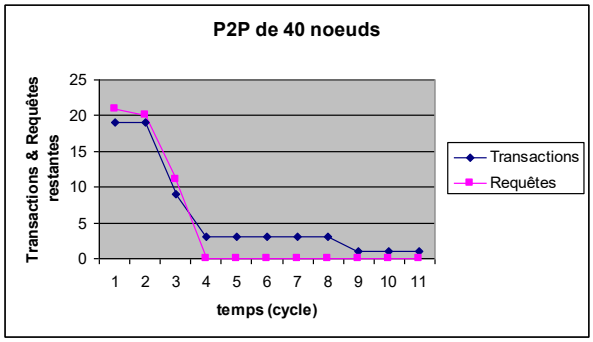


Figure 21a : Temps de traitement dans un P2P de 40 nœuds

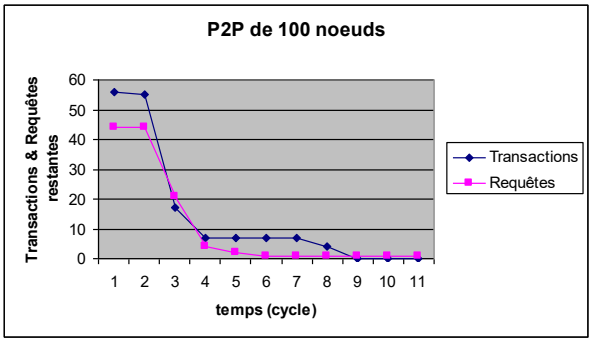


Figure 21b : Temps de traitement dans un P2P de 100 noeuds

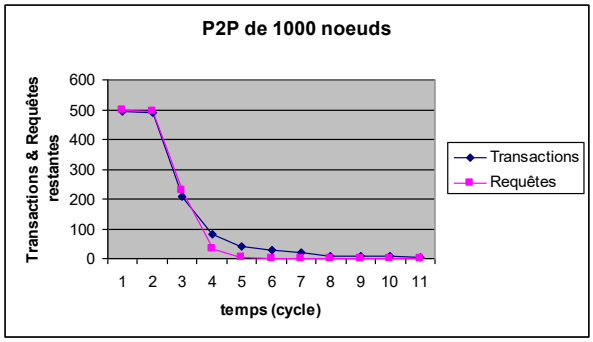


Figure 21c : Temps de traitement dans un P2P de 1000 nœuds

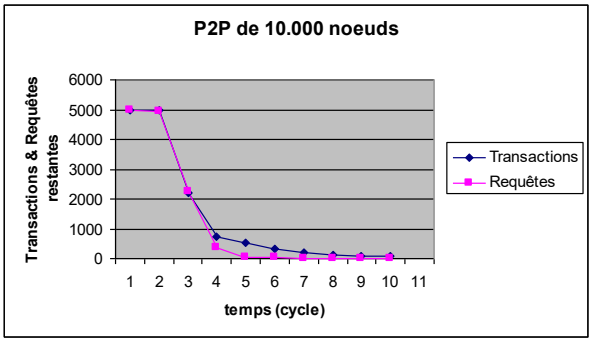


Figure 21d : Temps de traitement dans un P2P de 10.000 noeuds

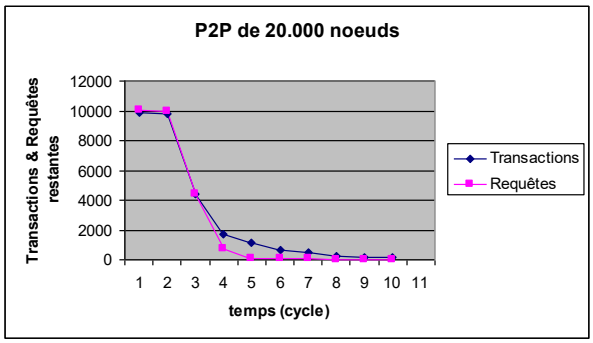


Figure 21e : Temps de traitement dans un P2P de 20.000 nœuds

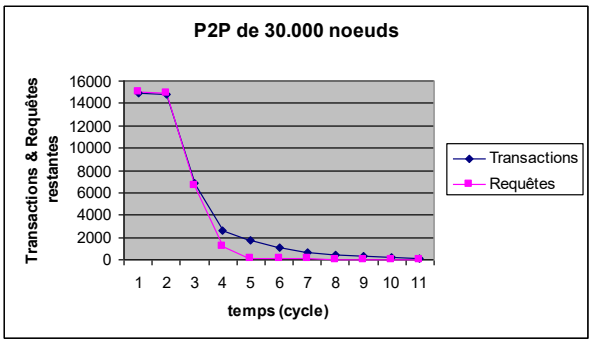


Figure 21f : Temps de traitement dans un P2P de 30.000 noeuds

On remarque aussi que l'agrandissement du système n'influe nullement sur le temps global d'exécution. Ceci s'explique par le fait que le traitement est complètement partagé entre les différents nœuds du réseau. Ainsi, si le nombre de transactions s'accroît en fonction du nombre élevé de nœuds constituant le réseau, il n'en demeure pas moins la probabilité de trouver un nœud maître capable de traiter une transaction (ou une requête). Parallèlement, ce résultat indique aussi une très grande absence de *skew* [13], ce qui engendre implicitement un bon équilibrage des charges.

Nous allons à présent mesurer la performance de notre algorithme (LB) en le comparant avec la stratégie round robin (RR). Cette stratégie utilise la réplication symétrique asynchrone mais les nœuds, à tour de rôle, vont traiter les transactions qui sont en attente. Nous utilisons toujours la même charge transactionnelle composée de requêtes (ou transactions) courtes et longues. Nous obtenons la figure suivante, qui montre que le temps de réponse donné par notre algorithme est infiniment petit par rapport à celui donné par l'algorithme RR.

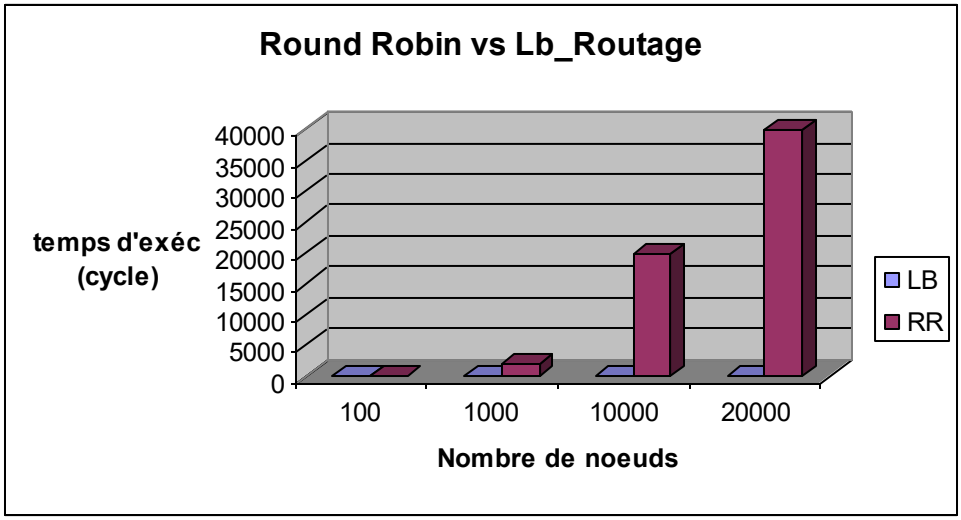


Figure 22 : Temps d'exécution avec LB et RR

On voit également que le désavantage de RR s'accroît quand le nombre de nœuds augmente au moment où notre LB maintient ses avantages.

Enfin nous allons vérifier l'apport positif qu'apporte le routage des transactions de synchronisation dans le temps d'exécution global. Nous comparons deux stratégies de synchronisation (immédiate et différée) avec un cas particulier de notre algorithme où le routage de synchronisation est désactivé. Pour la synchronisation immédiate, le routeur propage immédiatement la transaction sur tous les nœuds maîtres, ce qui garantit une fraîcheur maximale de tous les nœuds maîtres. Pour la synchronisation différée, le routeur ne synchronise les nœuds que lorsqu'ils sont à l'état *idle* (état dans lequel, ils n'ont pas de transactions à traiter) ou si leur TTL tend vers zéro. Pour une charge transactionnelle composée de requêtes (ou transactions) courtes et longues, la figure suivante montre que la synchronisation différée est plus bénéfique que celle immédiate, dans la mesure où elle permet une réduction importante du temps de réponse globale. Et même, quand le nombre de nœuds devient grand (≈10.000 nœuds) on voit que la synchronisation immédiate allonge le temps de réponse.

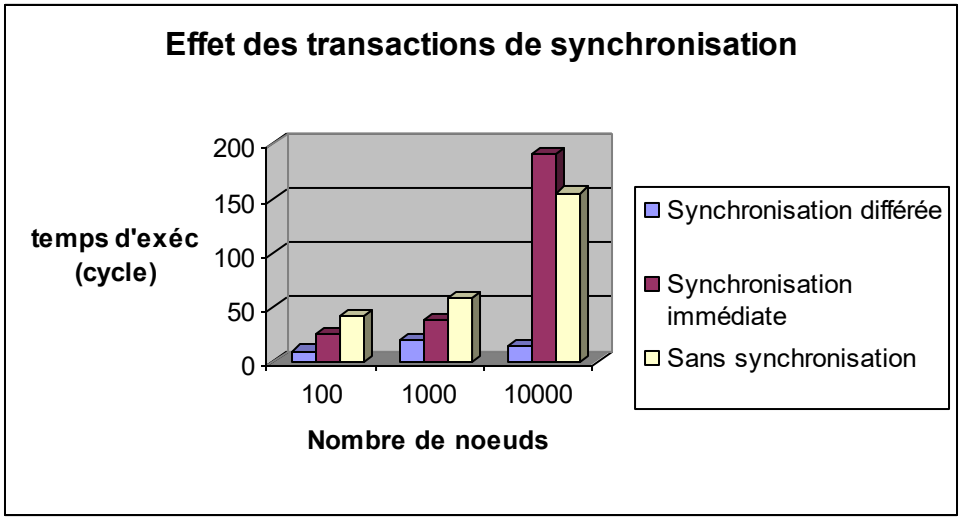


Figure 23 : Temps de réponse sans ou avec la synchronisation immédiate et différée

Néanmoins, quand le nombre de nœuds est de l'ordre de 1000, la synchronisation immédiate donne quelques avantages par rapport au routage sans synchronisation. Du fait que les transactions de synchronisation augmentent la charge transactionnelle, il est indispensable de trouver un compromis entre l'exécution des synchronisations et l'exécution des transactions. Ainsi dans notre expérience nous prenons quatre cycles pour traiter les transactions et un cycle pour effectuer la synchronisation. Durant la synchronisation, toutes les transactions sont bloquées.

Ces résultats prouvent que notre algorithme peut bel et bien être adapté dans un contexte P2P, car permettant de passer à l'échelle avec un bon équilibrage des charges tout en prenant en compte l'autonomie des pairs et l'aspect dynamique d'un tel réseau.