Réplication adaptative sur les réseaux pair à pair

Michel Chilowicz

10 mars 2006

1 Introduction

Les réseaux pair à pair sont des systèmes décentralisés permettant l'échange de ressources — tels que des fichiers — entre nœuds égalitaires. Cependant la popularité des ressources mises à disposition sur un réseau pair à pair est peu homogène et non-stationnaire : certaines données peuvent être très demandées que lors d'un court laps de temps. Nous étudions ici l'approche LAR proposée par Gopalarishnan, Silaghi, Bhattacharjee et Keleher [5] afin d'assurer une réplication adaptative des données au sein d'un réseau pair à pair. Ce protocole est surtout adapté à la réplication d'informations de routage mais peut être également utilisé pour la réplication de ressources de petite taille.

2 Réseaux pair à pair et tables de hachage distribuées

Nous abordons ici quelques généralités sur la conception de réseau pair à pair : nous nous intéresserons tout particulièrement aux tables de hachage distribuées et à leur répercussion sur le routage des requêtes de ressources.

2.1 Modèles d'acheminement de requêtes sur un réseau pair à pair

Différentes approches ont été envisagées afin de rechercher des ressources sur un réseau pair à pair.

2.1.1 Recherche par des serveurs centralisés

La méthode historique de recherche sur un serveur pair à pair consiste à déléguer la recherche de ressources à un serveur central : seuls les transferts de ressources sont réalisés de pair à pair. Cette technique a été utilisée notamment par le réseau Napster et continue également à être employé par des systèmes tels que BitTorrent [2] (utilisation d'un *tracker*).

L'utilisation d'un serveur centralisé est un moyen simple afin de réaliser un équilibrage de charge entre nœuds disposant d'une ressource : un nœud peut signaler régulièrement au serveur central sa charge actuelle et la charge maximale souhaitée. Le serveur central peut ensuite distribuer équitablement les demandes de ressources au nœuds afin d'équilibrer la charge. Il maintient donc en permanence la liste exhaustive des nœuds disposant d'une ressource donnée avec leur adéquation à répondre aux demandes.

Toutefois le modèle par utilisation d'un serveur d'annuaire présente l'inconvénient de faire reposer la fiabilité du réseau entier sur un nombre limité de serveurs. Une panne d'un serveur d'annuaire compromettrait alors grandement la stabilité de tout le réseau. De plus une telle solution résiste difficilement à un passage à l'échelle et pose des problèmes légaux aux mainteneurs des serveurs d'annuaires. Les réseaux pair à pair organisés autour de serveurs d'annuaires sont donc abandonnés au profit de réseaux décentralisés.

2.1.2 Recherche par inondation de requêtes

La recherche de ressources par inondation de requêtes est utilisée sur les réseaux pair à pair non-organisés : ne pouvant alors connaître la localisation potentielle d'une ressource, un nœud procède à l'inondation du réseau par des requêtes. Ce modèle est notamment utilisé par la première version du protocole Gnutella. Certains dispositifs tels que l'utilisation d'un champ de vie de la requête décrémenté à chaque passage par un nœud (champ *Time To Live*) ou une mémorisation des dernières requêtes acheminées par chacun des nœuds permettent d'éviter une circulation éternelle des messages-requêtes. Certaines optimisations telles que l'utilisation de filtres de Bloom permettent de réaliser une présélection des nœuds susceptibles de pouvoir répondre positivement à la requête.

Inconvénients La non-organisation d'un réseau pair à pair et l'utilisation de requêtes inondantes présente l'inconvénient de surcharger le réseau (la fraction de messages de requêtes par rapport aux données utiles échangées est élevée) et d'entraîner une latence élevée pour la satisfaction des requêtes. En effet, si l'on ne dispose pas de nœud dans son voisinage immédiat susceptible de satisfaire la requête, il est possible que celle-ci doive réaliser de nombreux sauts avant d'être satisfaite. Cela explique que les réseaux utilisant des méthodes de recherche par inondation soient généralement organisés autour de super-nœuds disposant d'importantes ressources.

2.1.3 Recherche sur une table de hachage distribuée

Les réseaux les plus récents s'organisent autour de systèmes de table de hachage distribuées. Il s'agit alors de distribuer l'annuaire des ressources disponibles sur le réseau sur tous les nœuds de celui-ci. Concrètement, chaque ressource est identifiée par un identifiant unique (qui peut être, par exemple, le résultat de l'application d'une fonction de hachage cryptographique sur le contenu du fichier mis à disposition). L'association entre identifiant unique de ressources et adresses des nœuds disposant de cette ressource est stockée sur différents nœuds du réseau selon la valeur de l'identifiant. Le principe du routage d'une requête consiste alors à acheminer celle-ci vers un nœuds maintenant l'association recherchée (la clé étant l'identifiant unique de ressource) dans la table de hachage distribuée.

Il est nécessaire qu'un système de table de hachage distribué équilibre correctement la taille des portions d'index mémorisées par chaque nœud afin d'équilibrer la charge lors d'une recherche. En théorie, l'usage de fonctions de hachage cryptographique garantit une bonne répartition des identifiants uniques de ressources. Cependant, on pourrait noter que la popularité des ressources n'est pas homogène : on peut néanmoins supposer que celle-ci est totalement indépendante de leur identifiant.

Quelques tables de hachage distribuées

Différents systèmes de table de hachage distribuée sont actuellement utilisés. Nous en citons ici quelques uns à titre d'exemple :

- CAN (Content Adressable Network) : cette table de hachage utilise un espace vectoriel cartésien multidimensionnel : chaque nœud gère une portion de l'espace et dispose d'informations sur ses voisins immédiats. Chaque identifiant unique de ressource est converti en coordonnées dans l'espace CAN et le routage est réalisé de proche-en-proche : pour réaliser un saut, le nœud réceptionnant la requête la communique au voisin le plus proche des coordonnées recherchées.
- Chord [9] : les nœuds sont organisés en anneau. Chaque nœud dispose d'un identifiant et est chargé de gérer l'index pour les identifiants qui lui sont supérieurs ou égaux et inférieurs à l'identifiant du nœud suivant sur l'anneau. Un nœud d'identifiant n maintient une table

de routage vers les nœuds d'identifiants $n + 2^{i-1}$. La table est donc de taille logarithmique et les opérations de recherche de clé sont réalisées en temps logarithmique.

- Pastry [8] : ce système s'inspire de la topologie en anneau de Chord. Chaque nœud maintient une table de routage d'une taille $O(logN)$ (avec N le nombre de nœuds actifs dans le réseau) : le routage est alors assuré en choisissant pour le prochain saut le nœud de la table présentant le plus long préfixe binaire commun avec l'identifiant de la ressource (en pratique, les identifiants ont une taille de 128 bits).
- Tapestry [10].
- Skipnet [6].

Réplication de clés La plupart des systèmes de table de hachage distribuée gèrent la problématique de réplication d'index afin de pallier les problèmes liés à la faible durée de vie des nœuds sur le réseau. Lorsque ce n'est pas le cas, la redondance d'index peut être obtenue en utilisant simultanément plusieurs tables de hachage et en déduisant de multiples identifiants uniques pour chaque ressource.

Nous nous intéressons ici tout spécialement au système Chord qui est notamment utilisé par les auteurs de l'article que nous étudions [5] afin de réaliser des tests sur leur protocole de réplication.

2.2 La table de hachage distribuée Chord

Insertion d'un nœud sur l'anneau Pour la table de hachage distribuée Chord, tous les nœuds sont situés sur un anneau : la version originale stipule que chaque nœud adopte un identifiant propre de t bits[1] n et s'insère sur l'anneau entre le nœud d'identifiant immédiatement inférieur et celui immédiatement supérieur (son prédécesseur $p(n)$ et son successeur $s(n)$ sur l'anneau). Le nœud n gère alors les clés de valeur comprises entre $p(n) + 1$ et n (une ressource est donc gérée par son nœud successeur). Lors de son insertion, un nœud récupère alors la gestion d'une partie de l'espace des clés dévolue à son successeur.

Routage sur l'anneau

Table de routage de chaque nœud Chaque nœud maintient une table de routage (*fingerlist*) : en théorie, seule la connaissance du successeur est nécessaire, le routage peut alors être effectué de proche-en-proche en temps $O(N)$ pour N nœuds sur l'anneau. Un tel procédé de routage est néanmoins inefficace pour le nombre important de nœuds rencontrés sur un réseau pair à pair : ainsi on réalise une table de routage dont l'entrée i contient l'adresse du nœud $s\left(\left(n + 2^{i-1}\right) \mod 2^t\right)$. La table de routage utilise un espace en $O(\log N)$. La constitution d'une telle table de routage peut être réalisée par la connaissance de la table de routage de p_n. D'autre part, il est nécessaire de mettre à jour cette table lors de la suppression et l'insertion de nœuds sur l'anneau : ce point est notamment très délicat à réaliser pour la suppression inopinée d'un nœud.

Routage d'une requête Le routage d'une requête sur l'anneau est réalisé en utilisant la *fingerlist* de chaque nœud. Pour la recherche d'une ressource d'identifiant id, la problématique consiste à connaître le nœud d'identifiant immédiatement inférieur ou égal à id : on recherche alors dans la table de routage l'entrée i telle que $\left(id - (n + 2^{i-1})\right) \mod 2^t$ soit une valeur positive ou nulle minimale. Le nœud suivant sur le chemin de routage réitère l'opération, ... : la requête se rapproche alors de la destination (à chaque saut, sa distance de la destination est globalement divisée par deux) jusqu'à l'atteindre. Le routage est ainsi réalisé avec un nombre de sauts en $O(\log N)$ dans le pire des cas.

[1]En pratique $t = 160$, la fonction de hachage cryptographique SHA-1 étant utilisée.

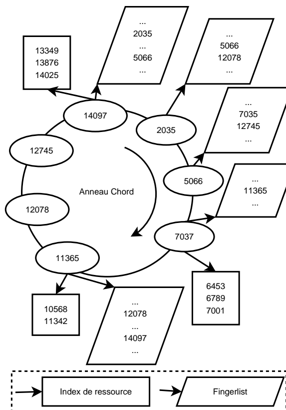

Figure 1: Exemple d'anneau Chord

Exemple de routage Nous présentons ici un exemple de routage avec l'anneau Chord présenté en figure 1 (seule un nombre réduit de nœud de l'anneau est présenté). Le nœud d'identifiant 14097 recherche la ressource d'identifiant 10568. Le nœud 14097 consulte sa *fingerlist* : le nœud le plus proche inférieur à 10568 qu'il connaisse est 5066 : ce nœud est choisi comme prochain saut. La requête arrive sur 5066 qui décide 7037 comme prochain saut. 7037 est le prédécesseur de 11365, le nœud maintenant la clé recherchée 10568 : 7037 lui transmet donc la requête.

2.3 Réplication classique de contenu

2.3.1 Réplication par le demandeur d'une ressource

Préalablement à la discussion du protocole de réplication LAR, nous pouvons noter que la plupart des réseaux pair à pair d'échange de fichiers introduisent une réplication de données assurée par le demandeur. En effet, lorsqu'un nœud a obtenu une ressource, celui-ci la met à disposition des autres nœuds : la popularité d'une ressource entraîne ainsi la création de nouveaux réplicats. Les nœuds disposant d'une ressource suite à son obtention sur un autre nœud signalent leur mise à disposition de la ressource, dans le cadre de réseaux pair à pair organisés, par insertion de sa clé dans le système de table de hachage distribué ou alors par message envoyé au serveur centralisé d'annuaire.

2.3.2 Incitation à la réplication

Ce mécanisme assure déjà un équilibrage de charge acceptable entre nœuds pour une ressource devenant populaire. On notera cependant que les pairs participant au réseau peuvent choisir de ne pas créer un réplicat pour chaque ressource qu'ils obtiennent d'un autre nœud (phénomène de *freeriding*) : la présence de nombreux nœuds adoptant ce comportement compromet l'accessibilité de données de forte popularité au sein du réseau. Certaines solutions peuvent être adoptées pour inciter un nœud à la réplication de ressources téléchargées. On peut noter par exemple la solution adoptée par BitTorrent (*tit for tat* [3]). On pourrait également envisager d'attribuer des scores de réputation aux nœuds (tels que Eigentrust [7])dépendant notamment de leur "générosité" à créer des réplicats et à les distribuer à d'autres nœuds.

2.3.3 Mécanismes de découpage de ressource

Pour les données de taille conséquente, il parait judicieux de procéder à un découpage préalable des fichiers en paquets de taille modeste. Un fichier peut ensuite être récupéré par un nœud depuis

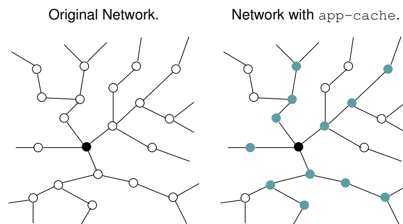

Figure 2: Mise en cache de la ressource demandée auprès d'un nœud sur le chemin d'acheminement de la requête [5].

plusieurs sources mettant à disposition des paquets de données. Les mécanismes de découpage en paquets permettent de créer des réplicats partiels (avant le téléchargement complet d'un fichier). Plusieurs approches peuvent être envisagées comme :

- Le découpage linéaire de données : le fichier de données de taille n est découpé en n/s paquets de taille s.
- L'utilisation de techniques de *network coding* : chaque paquet est la combinaison linéaire de plusieurs portions du fichier. Une étape de reconstruction du fichier par résolution d'un système linéaire d'équations est alors nécessaire pour reconstituer le fichier. Ce procédé est notamment utilisé par le système pair à pair *Avalanche* [4] pour résoudre le problème de non-homogénéité de disponibilité des différents paquets d'un fichier sur le réseau.

3 Le protocole app-cache

Pour réaliser leurs tests de leur protocole *LAR*, les auteurs de l'article auquel nous nous intéressons décident de le comparer à un protocole qu'ils nomment *app-cache* utilisé pour certains réseaux tels que CFS. Ce protocole consiste simplement à cacher les ressources demandées auprès des nœuds traversés pour l'acheminement d'une requête.

Prenons l'exemple d'un nœud n_0 demandeur d'une ressource r : en utilisant un protocole de routage spécifique, cette requête est acheminée par k nœuds $n_1, \ldots, n_k$. Le nœud n_k dispose de la ressource et la communique à n_0. Par le protocole *app-cache*, on créé des réplicats sur les nœuds $n_1, \ldots, n_k$ (voir figure 2). Ainsi si une requête pour la même ressource est traitée par le réseau, celle-ci sera probablement acheminée vers le nœud n_k : les sauts utilisés pour l'acheminement de la requête utiliseront avec une forte probabilité un ou plusieurs des nœuds utilisés sur le chemin de la précédente requête identique $n_1, \ldots, n_k$. Un tel nœud accueillant un réplicat pourra répondre directement au demandeur sans avoir à rediriger la requête : on décharge alors le nœud n_k disposant originellement de la ressource.

4 Le protocole LAR

4.1 Objectifs de LAR

4.1.1 Gestion de phénomènes de foules

Le principal objectif du protocole *LAR* introduit ici consiste à gérer la surcharge induite par des phénomènes de foule — se manifestant par la montée en popularité soudaine d'une ressource

pendant un court intervalle de temps — par la création de réplicats. Le protocole adopté devra donc présenter la particularité de créer rapidement de nouveaux réplicats lors de la montée en popularité d'une ressource, voire de les supprimer pour privilégier d'autres ressources lors d'une chute de la demande.

4.1.2 Équilibrage de charge

On cherche à distribuer harmonieusement la charge de demande de ressources entre les différents nœuds du réseau. La création de réplicats permet d'assurer une redondance des ressources populaires, mais il est également nécessaire que les nœuds du réseau aient connaissance des nouveaux réplicats créés et les privilégient pour leur demande de ressources aux serveurs déjà chargés.

Intervalles de charges On distingue trois intervalles de charge pour une machines exécutant une application pair à pair :

1. Un intervalle de charge faible ($0 \leq l < l_{faible}$) : un serveur de charge faible doit augmenter sa charge en accueillant des demandes de ressources pour de nouveaux réplicats et permet ainsi de transférer un surplus de charge d'un autre serveur.
2. Un intervalle de charge équilibrée ($l_{faible} \leq l < l_{forte}$).
3. Un intervalle de charge forte ($l_{forte} \leq l < l_{max}$) : des serveurs de charge trop forte doivent transférer leur surplus à des serveurs de charge faible.

4.1.3 Décisions locales

Toutes les décisions réalisées par l'algorithme doivent être réalisées uniquement sur la base d'informations localement disponibles afin de supporter un passage à l'échelle. Il est préférable d'éviter le transfert de données spécifiques pour la gestion des réplicats pour ne pas surcharger le réseau : ainsi par exemple, il est coûteux qu'un nœud hébergeant à l'origine une ressource soit informé exhaustivement des évènements de création ou suppression de réplicats (surtout dans la mesure où le nombre de réplicats pour certaines données peut être particulièrement élevé). On autorise donc que des réplicats soient créés ou supprimés sans qu'en soient informés les nœuds originaires.

4.2 Description du protocole LAR

Nous décrivons ici le fonctionnement du protocole de réplication adaptative *LAR*.

4.3 Création de réplicats par mesure de charge

LAR présente l'originalité de prendre en compte la charge des serveurs pour la prise de décision de création de nouveaux réplicats. Concrètement, l'équilibrage de charge est réalisé lorsqu'un nœud demandeur n_j réalise une demande de ressource à un nœud offrant n_i : une fois la ressource obtenue de n_i, n_j peut créer un réplicat accessible de la ressource. Le réplicat est effectivement créé si n_i est en surcharge ($l(n_i) > l^i_{forte}$) et si $l(n_i) - l(n_j) > K$ où K est une constante définissant la différence de charge nécessaire pour la création de nouveau réplicat (n_i peut préalablement connaître la charge de n_j car celui-ci la lui a communiquée). Si $l^i_{faible} \leq l(n_i) < l^i_{max}$, alors un nouveau réplicat est créé ssi $l_i - l_j \geq l^i_{faible}$. Si nécessaire, l_j peut également procéder à la création de réplicats autres que ceux réclamés auprès de l_i si ceux-ci permettent de diminuer sa charge et respectent les règles énoncées précédemment : les ressources générant le plus de charge sur n_i sont alors considérées en premier lieu pour l'équilibrage de charge.

4.4 Destruction de réplicats

Chaque nœud maintient sur sa mémoire de masse un ensemble de réplicats. L'espace disque consacré pour le stockage des réplicats peut être défini par l'utilisateur du nœud : lorsque l'espace alloué est dépassé, il est nécessaire de supprimer un ou plusieurs réplicats afin d'en accueillir un nouveau. On utilise alors une politique de suppression *LRU* (*Least-Recently-Used*) : le réplicat dont l'utilisation est la plus ancienne (heuristiquement la moins populaire actuellement sur le réseau) est évacué du cache.

Il est nécessaire de noter que la suppression de réplicats est, dans le protocole *LAR*, une décision purement locale qui ne fait l'objet d'aucune communication et coordination extérieure. Les nœuds ayant connaissance de l'existence de ce réplicat ne sont donc pas prévenus de la suppression de celui-ci.

4.4.1 Caches de pointeurs vers des réplicats

Annonce des réplicats Après avoir créé de nouveaux réplicats, il est nécessaire de les annoncer auprès des autres nœuds. L'utilisation de tables de hachage distribuées peut ainsi être utilisée pour ajouter de nouvelles valeurs correspondant à la clé identifiant la ressource. *LAR* propose une approche générique consistant à signaler à tous les nœuds sur le chemin de la requête utilisée l'existence du nouveau réplicat créé.

Annonce sur le chemin de la requête La solution adoptée par *LAR* consiste à notifier à tous les nœuds sur le chemin de la requête entre n_j (le nœud demandeur de ressource et accueillant le réplicat) et n_i (le nœud offrant la ressource et demandant la réplication) l'existence du réplicat sur n_j.

Réponse raccourcie aux requêtes La présence de pointeurs sur le chemin de la requête permet ensuite, de répondre plus rapidement aux requêtes suivantes, et potentiellement de réduire le nombre de sauts nécessaires pour le routage. En effet, si l'on atteint un nœud disposant d'un pointeur vers un réplicat plutôt que le nœud disposant originellement de la ressource, on suivra ce pointeur.

Cache de taille paramétrable avec politique LRU Le cache de pointeurs maintenu par chaque nœud possède une taille limitée paramétrable. De la même façon que pour les répliques, une politique *LRU* est utilisée : le pointeur le moins récemment utilisé est supprimé.

Politique de dissémination Pour chaque ressource, un nœud maintient un nombre limité de pointeurs vers des réplicats. D'autre part, lorsque l'on transmet un message contenant des entrées de nouveaux réplicats au nœuds sur le chemin de la requête, on inclut non-seulement le nouveau réplicat créé mais plutôt un nombre fixe de réplicats connus pour la ressource : on sélectionne les k réplicats créés le plus récemment par le nœud (qui ont le plus de probabilité d'être inconnus du nœud). Lorsqu'un nœud reçoit un message d'annonce de réplicats et que son cache est saturé pour la ressource concernée par les réplicats (ce qui signifie que la ressource est très populaire), il peut paraître intéressant de supprimer du cache les pointeurs les moins récemment utilisés pour les remplacer par ceux véhiculés par le message.

Obsolescence des pointeurs Un des problèmes soulevés par le protocole *LAR* réside dans la possibilité de subsistance de pointeurs obsolètes suite au départ inopiné d'un nœud contenant des réplicats ou tout simplement lorsqu'un réplicat est supprimé par politique *LRU*. Une solution envisageable à ce problème consiste pour le demandeur d'une ressource à continuer préalablement la recherche de la ressource par routage classique en parallèle du test de l'existence du réplicat pointé par des nœuds sur le parcours. Lorsque la ressource est trouvée (soit par recherche classique, soit par utilisation de pointeur sur réplicat), la recherche est arrêtée.

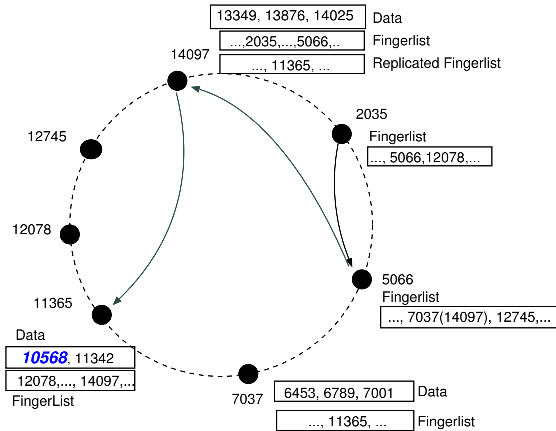

Figure 3: Routage Chord en utilisant la réplication LAR

5 Tests de performance du protocole LAR

Gopalakrishnan et al. réalisent des tests de performance de leur protocole *LAR* en utilisant comme base le simulateur [1] proposé dans le cadre du projet Chord. Ils comparent les performances obtenues par l'utilisation d'un protocole *App-cache* et *LAR* avec l'usage d'aucune technique de réplication.

5.1 Usage avec la table de hachage distribuée Chord

La table de hachage distribuée Chord maintient des tables de routage pour chacun de ces nœuds (appelées *fingerlist*). *LAR* peut alors être utilisé pour réaliser la réplication de *fingerlist*. La figure 3 présente un exemple de réplication de *fingerlist* et la réalisation de routage en utilisant de tels réplicats :

1. Le nœud 2035 demande la ressource d'identifiant 10568.
2. Le nœud immédiatement inférieur à 10568 connu par le nœud 2035 est 5066 qui est choisi pour le prochain saut.
3. La requête parvient au nœud 5066 qui choisit de l'acheminer vers me nœud 7037 : cependant disposant en cache de l'information de réplication de la fingerlist de 7037 en 14097, il décide de rediriger la requête vers 14097.
4. La requête arrive en 14097 : la table de routage répliquée contient une référence vers le nœud 11365, nœud successeur de 10568.
5. Finalement le nœud 11365 détient la ressource 10568 et la transmet à 2035.

5.2 Paramètres de simulation

Les paramètres de simulation utilisés pour les tests sont les suivants :

- 1000 nœuds sur le réseau.
- 32767 ressources hébergées.
- l_{max} est fixé à 10 requêtes par seconde.

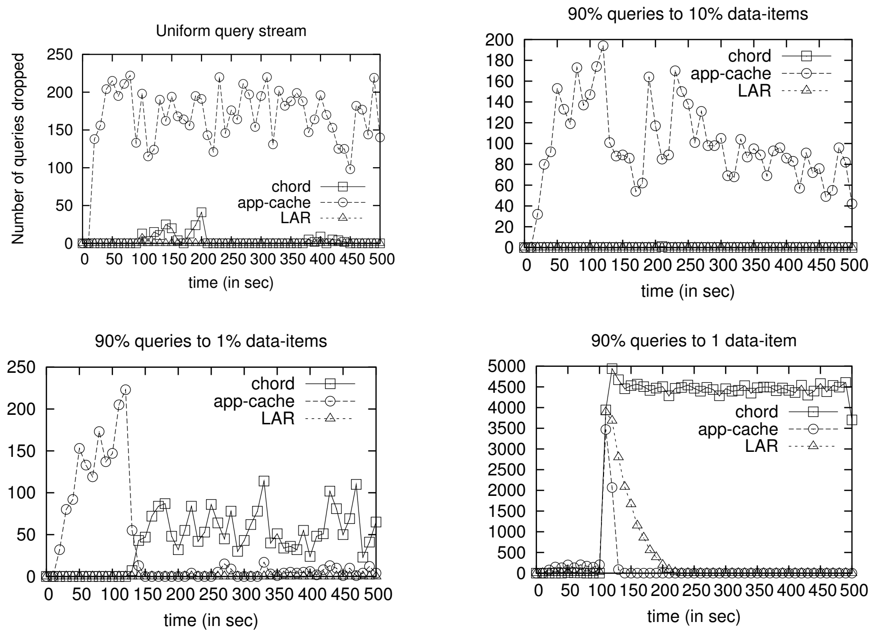

Figure 4: Nombre de requêtes rejetées pour différentes distribution de requêtes [5]

- Chaque nœud utilise une file de traitement de requête de capacité égale au nombre de ressources qu'il héberge (32).
- La charge est recalculée toutes les deux secondes.
- Chaque saut sur le réseau est réalisé en 25 millisecondes.
- 500 requêtes sont générés par seconde sur le réseau.
- Le chemin moyen d'une requête contient moins de 5 sauts[2] (ce qui correspond à une charge moyenne inférieure à 25% pour chaque nœud).
- Sources de requêtes sélectionnées aléatoirement (distribution uniforme).
- Contribution égale de chaque transfert de message à la charge et à la congestion.
- Les messages arrivant sur un serveur surchargé ($l > l_{max}$) avec une file pleine sont rejetés.

5.3 Effet de la distribution des requêtes

Les diagrammes de la figure 4 reflètent les effets de la variation de la distribution des requêtes. Les tests sont réalisés sur 500 secondes.

5.3.1 Distribution uniforme

Lorsque l'on considère une distribution de requêtes uniforme (chaque ressource possède une probabilité égale de faire l'objet d'une requête), on constate la faible performance réalisé par le protocole *app-cache* : jusqu'à 20 requêtes peuvent être rejetées chaque seconde (soit 4% des requêtes) : cet effet s'explique par le surcoût entraîné par la réplication systématique (et ici inutile) opérée par *app-cache*. En effet, sur la période de test de 500 secondes, plus de 1 million de réplicats sont

[2]Nous pouvons noter que dans le pire des cas, le routage est réalisé en $\log_2 N$ sauts dans un réseau Chord de N nœuds, soit ici environ 10 sauts.

créés puis rapidement supprimés. Au contraire, moins de 1% des requêtes sont rejetées avec LAR ou aucun mécanisme de réplication sur Chord : LAR ne créé que 5000 réplicats.

5.3.2 90% des requêtes sur 10% des ressources

app-cache présente dans cette situation un rejet pouvant atteindre 40% des requêtes : près de 1 million de réplicats sont créés. LAR et *Chord* présente un taux de rejet de requêtes négligeable.

5.3.3 90% des requêtes sur 1% des ressources

Pendant les 150 premières secondes de la simulation, *app-cache* créé un nombre important de réplicats : la surcharge ainsi générée entraîne un fort taux de rejet de requêtes (jusqu'à 22 par seconde). Au-delà, le rejet des requêtes est faible. LAR, contrairement à *app-cache*, par son mécanisme de création de réplicats dépendant de la charge, permet un taux de rejet de requêtes négligeable.

5.3.4 90% des requêtes sur une ressource

Cette situation simule un phénomène de foule extrême (démarrant à la seconde 100 de la simulation). Pour un tel scénario, *app-cache* permet de satisfaire près de 244000 requêtes tandis que LAR n'en satisfait que 233000 environ. L'usage d'aucune technique de réplication conduit à la satisfaction de 72000 requêtes environ (soit 29% des requêtes). L'usage d'un procédé de réplication prend donc tout son intérêt. Le dernier diagramme de la figure 4 montre la dynamique de rejet de requêtes lors de ce phénomène de foule : lorsque l'on passe subitement d'une répartition uniforme des requêtes à 90% des requêtes sur une ressource, Chord sans réplication rejette la quasi-totalité des requêtes tandis que *app-cache* et LAR, par la création de réplicats, équilibrent la charge et permettent une diminution progressive du taux de requêtes rejetées. Finalement en moins de 100 secondes, le taux de rejet devient négligeable : la politique de création agressive de réplicats de *app-cache* permet néanmoins une chute plus rapide du taux de rejet.

Réplication dans Chord On remarquera que la demande d'accès à une ressource d'identifiant *id* sollicite le nœud d'identifiant immédiatement supérieur à *id* sur l'anneau. Toutefois, avant l'arrivée sur ce nœud, le protocole de routage de Chord conditionne le passage par le prédécesseur de ce nœud. Cette observation démontre l'intérêt à la réplication des tables de routage des prédécesseurs. On peut toutefois noter qu'il s'agit d'une spécificité de Chord non rencontrée pour tous les systèmes de tables de hachage distribuées.

5.4 Influence des coûts de transfert

Pour les conditions standard de simulation, le coût de transfert de tout message, qu'il s'agisse d'un document, d'une requête ou d'un message de contrôle est considéré identique. On pourra noter qu'il s'agit d'une situation très théorique peu rencontrée dans la pratique. Si l'on considère cependant que le coût de transfert d'un document (ressource) est dix fois supérieur aux coûts de transferts des autres messages, dans le cadre d'une distribution 90/10, le taux global de rejets de requêtes passe de 2% à 42% pour *app-cache*. Pour LAR, le taux de rejet passe de 0% à 10%. LAR supporte donc mieux la réplication de données de taille importante. On notera toutefois que l'hypothèse d'un coût 10x pour les messages de données est encore très largement sous-estimé par rapport aux situations réelles.

5.5 Charge moyenne

On remarque que *app-cache* est plus sensible à l'augmentation de charge sur une machine que LAR (les tests ont été réalisées avec une distribution des requêtes 90/10) : LAR peut répondre à plus de 97% de requêtes avec une charge de 50% contre 83% pour *app-cache*.

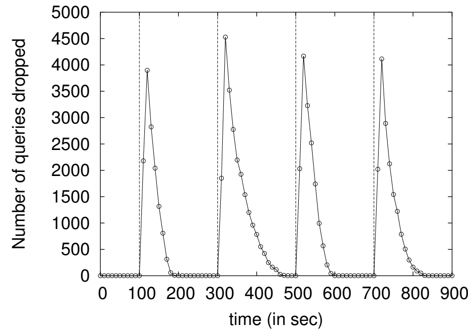

Figure 5: Nombre de requêtes rejetées sur 900 secondes pour une distribution de 90% de requêtes sur une ressource, avec changements brusques aux secondes 100, 300, 500 et 700 [5]

5.6 Évolution de la popularité de données

Le diagramme de la figure 5.6 présente le nombre de requêtes rejetées à travers le temps (cumul sur les dix dernières secondes) lors de changements soudains de popularité. On utilise une répartition de 90% de requêtes pour une seule ressource, aux secondes 100, 300, 500 et 700, la ressource populaire change. Ces changements provoquent une augmentation rapide du nombre de requêtes rejetées. La charge du serveur proposant la ressource populaire devenant importante, il demande la création de réplicats auprès des émetteurs de requêtes. Cette phase de création de réplicats permet d'absorber progressivement le surplus de requête : *LAR* est ainsi capable de s'adapter relativement rapidement (en moins de 2 minutes) à une nouvelle ressource populaire.

5.7 Passage à l'échelle

Comment *LAR* supporte-t-il le passage à l'échelle ? Les expériences précédentes étaient réalisés sur 1000 nœuds : la figure 6 montre le comportement de *LAR* pour un nombre variable de nœuds (le flux de requêtes est adapté pour garantir une charge moyenne de 25%, la distribution des requêtes est de type 90/10). On constate que l'augmentation du nombre de nœuds entraîne nécessairement une augmentation du taux de rejet des requêtes lors de la phase d'adaptabilité (création de réplicats) : en effet, le nombre de requêtes transitant sur le réseau est plus important. Cependant la durée de la phase d'adaptabilité reste quasi-constante (moins de deux minutes).

6 Conclusion

Nous avons présenté le protocole *LAR* avec quelques tests réalisés sur un réseau pair à pair utilisant un système de table de hachage distribuée Chord. *LAR* présente des avantages certains sur *app-cache* : il introduit un concept d'adaptabilité selon la charge du nœud, ce qui permet de ne pas pénaliser le fonctionnement du réseau en l'absence de phénomènes de ressources fortement populaires. Le protocole proposé est effectivement indépendant du système d'organisation utilisé : cependant les tests réalisés ne sont effectués que sur Chord. D'autre part, il est possible de noter que le mécanisme de réplication proposé par constitution de réplicat sur un demandeur de la ressource n'est en rien une nouveauté sur les réseaux pair à pair.

La seule nouveauté introduite consiste en un équilibrage de charge par analyse de la charge locale et demande de constitution de réplicats sur d'autres ressources que celles demandées. Un tel procédé ne peut être efficace que sur des données de faible taille : un tel mécanisme peut

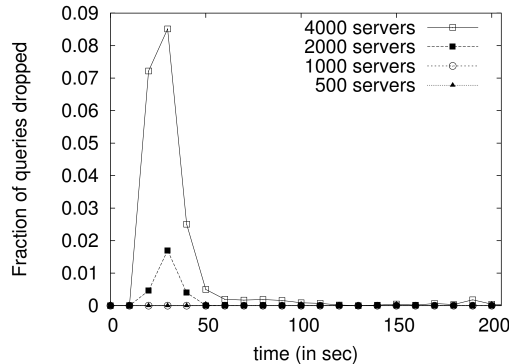

Figure 6: Fraction des requêtes rejetées pour différents nombre de nœuds sur le réseau (charge moyenne de 25% sur chaque nœud, distribution des requêtes 90/10) [5]

s'avérer efficace pour réaliser la réplication de pages Web. Toutefois il est nécessaire de remarquer que le protocole *LAR* ne peut traiter que des données immutables : aucun mécanisme n'est prévu pour la mise à jour des réplicats, un tel mécanisme étant indispensable pour la mise au point de proxy Web pair à pair où une page Web peut faire l'objet de mises à jours.

Pour des tailles de documents importantes, le système peut sans doute être généralisé par découpage en petits paquets soit par une technique de fractionnement linéaire, soit par utilisation d'une méthode de *network coding* [4]. Il pourrait être intéressant à cet effet de réaliser des simulations dans le cadre de distribution de fichiers de taille conséquentes découpées en petits paquets stockés et répliqués indépendamment sur chaque nœud du réseau.

Enfin, on remarquera que tous les tests ont été réalisés sur un réseau pair à pair statique. Dans la pratique, les réseaux pair à pair sont dynamiques, dans le sens où la durée de vie d'un nœud sur le réseau est faible ; à chaque instant un nombre important de nœuds entrent et quittent le réseau. Le protocole *LAR* ne dispose d'aucun mécanisme pour la gestion de départ des nœuds : des pointeurs vers des réplicats peuvent rapidement devenir obsolètes. Une approche pro-active de suppression des pointeurs obsolètes n'aurait-elle pas été préférable ?

References

[1] Chord simulator.

[2] Bram Cohen. Système pair à pair BitTorrent.

[3] Bram Cohen. Incentives build robustness in BitTorrent, 2003.

[4] Christos Gkantsidis and Pablo Rodriguez. Network coding for large scale content distribution. Technical report, Microsoft Research, 2004.

[5] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems, 2003.

[6] Nicholas Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[7] S.D. Kamvar, M.T. Shloesser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. 2003.

[8] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[9] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[10] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.