Institut National des Sciences Appliquées de Lyon

Ecole Normale Supérieure de Lyon

Laboratoire d'InfoRmatique en Image et Systèmes d'information

Réseaux haut débit, protocoles et services

Rapport de DEA sur :

Intégration de la fonction de proxy cache aux routeurs actifs

Réalisé Par :
GUEBLI Sid Ali

Stage effectué du 03-03-2003 au 09-07-2003 sous la responsabilité de :

Jean- Marc Pierson

Maître de conférences
à l'INSA de Lyon (INSA-Lyon),
au Laboratoire LIRIS.

Laurent Lefèvre

Chargé de recherches INRIA.
Membre de l'équipe RESO du LIP.
ENS - Lyon

Laboratoire :

Laboratoire d'Informatique en Image et Systèmes d'information
LIRIS

Remerciement

Ayant achevé le stage de DEA qui s'est effectué au LIRIS à l'INSA de Lyon. Je remercie Monsieur. **FLORY André** de m'avoir accepter dans ce DEA**.**

Je remercie **Jean Marc Pièrson** et **Laurent Lefèvre**, mes maîtres de stage, pour m'avoir encadré durant mon stage de DEA et la confiance qui m'ont accordé, ainsi que m'avoir fait bénéficier de leurs conseils et de leur expérience.

Un merci à **Jean Patrick Gelas** pour son aide qui m'a permis de réaliser ce travail

Un merci à tous les membres de l'équipe du LIRIS pour le soutien et l'ambiance familiale qu'ils n'ont pas manqué d'apporter tout le long du stage.

Je remercie également tous mes amis pour leur aide et leur soutien moral qu'ils m'ont apporté pour la réalisation de mon stage.

Et à tous ceux qui ont fait preuve de patience à mon égard…

Dédicaces

Je voudrais donc dédicacer ce mémoire,
fruit du travail de quatre mois,
à mon père qui m’a encouragé à venir poursuivre mes études,
à ma mère, à toute ma famille qui m’a soutenu moralement
depuis mon arrivée en France,
à ma meilleure amie Faiza,
à tous mes amis,
et à tous ceux qui m’ont aidé à réaliser ce projet.

Table des matières :

Chapitre 4 Mise en œuvre et tests

Table des figures :

Introduction

Le World Wide Web fournit un accès simple pour une grande variété d'informations et de services. Il est devenu l'application la plus utilisée sur Internet. Vu le nombre important des utilisateurs connectés à Internet, plusieurs d'entre eux peuvent demander les mêmes documents et en même temps à des serveurs distants. De tels scénarios génère inutilement du trafic réseau et une charge supplémentaire des serveurs. Afin de remédier à ces problèmes, et dans le but de rendre le Web plus attractif, des systèmes de cache ont vu le jour, permettant de sauvegarder des copies multiples du même document dans des caches dispersés géographiquement. Cette solution permet de réduire considérablement le trafic circulant dans le réseau.

Actuellement, les caches sont principalement déployés à deux niveaux : au niveau des navigateurs Web et au niveau des proxies caches. L'utilisation des proxies caches est la technique la plus utilisée. Dans un tel système, un ou plusieurs ordinateurs agissent comme un cache de documents pour les clients WWW. Ces clients sont configurés pour envoyer leurs requêtes Web vers le proxy. A la réception d'une requête de l'un de ces clients, le proxy répond à partir du cache si c'est possible (ie : le document demandé est déjà caché auparavant). Sinon le proxy renvoie la requête vers le serveur distant qui détient la copie originale du document requêté.

Le proxy cache tente d'améliorer les performances, en réduisant le trafic réseau, de réduire la charge des serveurs occupés ainsi que le temps de latence des clients pour obtenir un document. Dès qu'un document est sauvegardé dans le proxy cache, plusieurs requêtes Web peuvent être satisfaites directement du cache, sans générer du trafic supplémentaire et inutile depuis et vers les serveurs.

Pour obtenir de meilleures performances avec moins de ressources, on fait appel à la coopération entre les différents proxies cache. Dans une telle architecture, plusieurs proxies caches distribués coopèrent pour partager leurs ressources avec les autres, en essayant de combiner leurs caches individuels dans le but de maximiser l'usage du cache, et en agissant en transparence comme un seul proxy. Les proxies qui sont incapables de résoudre les requêtes entrantes, ont le choix entre renvoyer la requête directement au serveur distant, ou demander de l'aide aux proxies voisins pour l'objet demandé, en utilisant un protocole de communication inter cache (coopération).

Les différents systèmes de caches utilisent des méthodes de remplacement de cache afin de gérer l'espace disponible dans le cache. Malheureusement, peu d'entre eux favorisent la coopération des caches. Un point important de la coopération est de garder en cache les documents qui ont un intérêt commun entres les utilisateurs des caches coopératifs. Ceci se traduit par la notion de communauté d'utilisateurs, qui partage les mêmes centres d'intérêt. Pour ce fait, on s'intéressera à une méthode de remplacement de cache qui introduit de l'intelligence dans la gestion du cache, en introduisant la sémantique dans la gestion des documents. Cette technique a été développée par une équipe du laboratoire LIRIS à l'INSA de LYON.

La mise en œuvre d'un nouveau protocole de communication inter cache, qui peut être simple, flexible et extensible, reste un challenge aux vues des limites des réseaux classiques à déployer de nouvelles fonctionnalités et de nouveaux protocoles, sans faire appel à une norme de standardisation. Nous avons trouvé dans les réseaux actifs une solution pour contourner ces limites. Un réseau actif est une technologie récente qui permet de déployer de nouveaux services et protocoles, et de les injecter dans le réseau. Ces réseaux sont composés d'équipements programmables (des nœuds actifs), qui effectuent des traitements sur les paquets les traversant. Tout nouveau protocole ou service déployé dynamiquement sur un noeud actif doit prendre en compte les contraintes de ce dernier, à savoir, une capacité de traitement et de stockage limitée.

Ce stage de DEA abordera d'une manière originale le domaine fortement étudié des caches coopératifs. Se fondant sur l'approche dynamique des réseaux actifs, il met en parallèle les services intelligents proposés par les routeurs programmables et les caches coopératifs. L'idée est d'introduire une certaine intelligence dans le traitement des paquets traversant le réseau, en respectant les contraintes imposées par les routeurs actifs, afin de réduire le nombre de paquets qui circulent dans le réseau. Nous proposons ainsi d'ajouter les fonctionnalités de proxy cache aux routeurs actifs et étendre la capacité de ces derniers pour qu'ils puissent former un système de cache coopératif. Ce système de coopération a comme but principal de produire les fonctions de base de la coopération, à savoir la *découverte* : comment localiser les objets cachés, et la *délivrance* : comment délivrer ces objets aux caches voisins.

Un modèle d'architecture de routeur actif proxy cache (RAPC) est proposé, en respectant les contraintes de ces derniers, et visant à réduire le nombre de paquets qui circulent dans le réseau pour la récupération des documents, et permettre une meilleure coopération des caches. L'intégration et la mise en œuvre de ce système dans un environnement actif ont été réalisées. Nous avons utilisé pour cela la plate forme *TAMANOIR*, qui a été développée par l'équipe RESO de l'INRIA. La validation expérimentale du protocole et les tests ont montré l'efficacité de l'approche.

Ce rapport est divisé en cinq parties. Nous faisons en chapitre 1 un survol des études et recherches effectuées dans le domaine des proxies cache coopératifs, puis nous rentrons dans le cœur des réseaux actifs en chapitre 2. Dans le chapitre 3 nous détaillons notre modèle d'intégration de la fonction de proxy cache aux routeurs actifs. Une partie est consacrée à la mise en œuvre et l'implémentation de l'approche sur un réseau actif, suivie de quelques tests. Nous ferons une discussion sur cette approche, et nous finirons par une conclusion sur le travail réalisé et les perspectives envisagées.

Chapitre 1

Système de caches coopératifs

1.1 Introduction :

Le World Wide Web est devenu une ressource fondamentale pour la distribution des informations. Cela a conduit à un accroissement immense de la quantité d'informations disponible ainsi que le nombre de données requêtées. Les proxies caches ont été longtemps utilisés dans le but d'améliorer les performances pour une communauté d'utilisateurs [1,23,24,28].

Cependant, et du à la nature distribuée du Web et les garantis offertes aux utilisateurs, plusieurs systèmes ont été proposé [2,4,6,27,23,24,28], afin d'une part de rendre le Web plus attractif en réduisant le temps de latence, et d'un part de réduire la bande passante utilisée. Au lieu de laisser chaque cache opérer séparément du reste du réseau, il est souvent bénéfique de permettre aux proxies caches d'utiliser les ressources des autres caches. Cette technique permet de réduire le temps de latence, la charge du serveur et le trafic réseau. On appelle un tel système, *système de caches coopératifs*.

La gestion du cache dans un réseau se fait à différent niveaux, afin d'obtenir le maximum de résultats. Il existe trois types de caches Web : ***les caches client***, ***les caches serveur*** et ***les caches proxy***. En général, les proxies peuvent être localisés prés du serveur (dans ce cas ils agissent comme des serveur miroirs), ou prés des clients dans le but de réduire le temps de latence [1,5].

Le cache client est disponible au niveau de la machine du client sur son browser Web. Actuellement les browsers Web offrent différentes options, comme définir la taille du cache de la mémoire vive et du disque où sont stockés les documents reçus. En utilisant ce cache, le client accède à une page sans faire de requête à un serveur si les fichiers correspondants sont stockés dans sa mémoire ou son disque.

Le cache serveur offre les mêmes fonctionnalités au niveau du serveur : en disposant en mémoire vive les documents le plus fréquemment accédés, il accélère l'accès à ses propre ressources.

Le cache proxy quant à lui, est un dispositif commun à plusieurs clients permettant à ceux-ci d'optimiser leur accès au Web. La solution du proxy permet d'interposer une capacité de stockage entre le client et le serveur.

Le principe est simple, lorsqu'un client cherche à accéder aux ressources d'un serveur, il envoie une requête au proxy. Celui-ci cherche ce document dans son cache et le retransmet alors au client. Si le document demandé par le client n'existe dans le cache, le proxy envoie la requête au serveur lui-même et retransmet le document reçu au client en stockant au passage le document dans son cache s'il le juge intéressant.

1.2 Les systèmes de caches coopératifs :

L'augmentation de la quantité d'informations requêtées dans le Web a engendré un besoin de développer des mécanismes qui aident à la gestion de cette utilisation croissante du Web. Une des solutions proposée est l'utilisation des caches coopératifs. Dans cette approche, plusieurs proxies cache distribués peuvent coopérer pour partager leurs ressources [2,12,9,31].

Les caches coopératifs forment un groupe de serveurs proxies HTTP qui partagent leurs objets cachés. Bien que plusieurs recherches [2,12,9,31] aient montré l'intérêt des proxies cache en terme de performances, en réduisant le temps de latence, la charge du serveur, et l'ensemble du trafic réseau. Le débat n'est pas clos sur les politiques de coopération entre proxies cache. Nous trouvons dans la littérature trois grands systèmes de caches coopératifs. Ces systèmes sont « Geographical Push » [23], « Hierarchical » [24,28], et « Adaptive » [2,4,6,27].

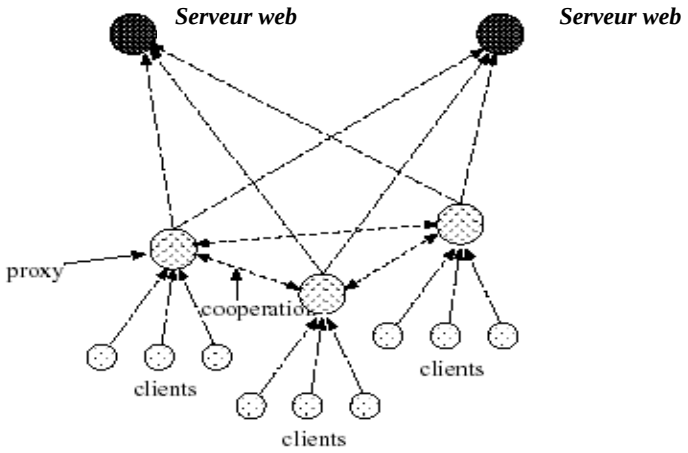

Figure 1.1. Schéma de proxies caches coopératifs.

1.2.1 Geographical Push caching :

Ce système a été proposé par Gwertzman et al [23]. L'idée de base et de garder les données proches des clients qui les demandent. Dans un telle architecture, le serveur primaire est responsable de ce qui doit être caché, où et quand. Il garde une trace de la fréquence et provenance de toutes les requêtes. Quand le nombre de requêtes atteint un certain seuil, le serveur primaire envoie (Push) le document demandé vers le cache distant. Le choix du cache approprié se fait en se basant sur l'espace disponible, la charge et l'origine des requêtes.

Pour récupérer un document, le client s'adresse directement au serveur. Ce dernier, au lieu de servir directement la requête, indique au client la position du cache le plus proche où il peut trouver le document. Le serveur peut toutefois traiter la requête lui-même, s'il est le plus proche.

1.2.2 Hierarchical caching :

Dans leur étude de transfert de fichier sur NSFNET, Danzig et al [25], ont montré qu'en fournissant un cache organisé de manière hiérarchique, la bande passante consommée pour le transfert de fichier peut être réduite énormément.

En suivant cette axe, les chercheurs de l'université de Colorado et l'université de Southern California, ont développé une stratégie de cache hiérarchique comme partie du projet Harvest [28].

Dans une telle architecture, les caches sont localisés à différents niveaux du réseau. Dans la plupart des cas, il est supposé que le niveau inférieur possède la meilleure qualité de service [21]. Donc, aux niveaux inférieurs de la hiérarchie, on trouve les caches des clients (L1) qui sont directement connectés aux clients (ie : caches incorporés dans les navigateurs). Au niveau suivant, on trouve les caches institutionnels (L2), qui peuvent être localisés dans des campus. Au niveau supérieur de la hiérarchie, on trouve des caches régionaux (L3) qui peuvent être connectés à un backbone national, qui relie plusieurs universités ou institutions à l'intérieur et l'extérieur du pays.

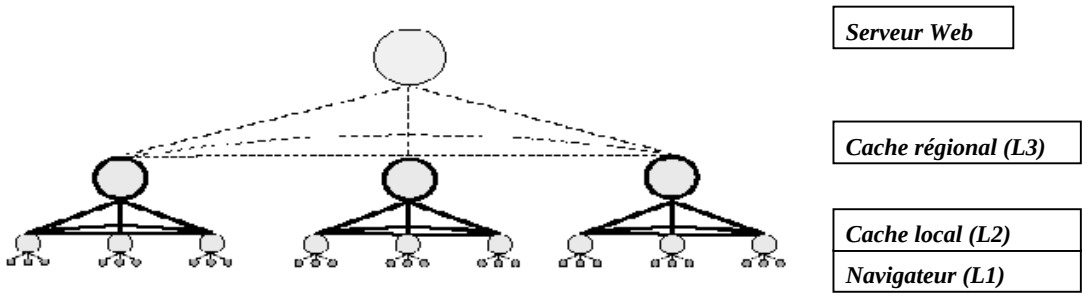

Figure 1.2. Structure hiérarchique de proxy caches coopératives.
(Lignes en gras indique la communication des caches L3).

Dans un système hiérarchique, quand une requête provenant d'un utilisateur, ne peut être satisfaite par le cache du client (L1), elle est redirigée vers les caches institutionnels (L2), ce dernier envoie les requêtes qu'il ne peut pas satisfaire aux caches régionaux (L3). A ce niveau le cache contacte directement le serveur d'origine. Quand un document est trouvé, il traverse la hiérarchie, en laissant une copie dans chaque cache intermédiaire. Placer des copies redondantes des documents dans les niveaux intermédiaires réduit le temps de connexion [30]. Ainsi, les prochaines requêtes pour le même document montent dans la hiérarchie jusqu'à ce que la requête trouve le document. L'environnement logiciel le plus utilisé pour les caches hiérarchiques est Squid [29], qui est un descendant du projet Harvest [28].

La structure de caches hiérarchiques est efficace pour une consommation modérée de la bande passante, particulièrement quand les caches coopératives n'ont pas une grande vitesse de connectivité. Cependant, malgré leurs bénéfices, il existe plusieurs problèmes liés aux caches hiérarchiques [16,30] :

1. Pour mettre en place une hiérarchie, les serveurs cache ont souvent besoin d'être placé à des points d'accès stratégiques du réseau. Cela demande une grande coordination entre les serveurs caches participants.

2. Chaque niveau de la hiérarchie introduit un délai additionnel.
3. Le niveau le plus haut de la hiérarchie peut représenter un goulot d'étranglement, et par conséquent il introduit des délais dans les réponses.
4. Des copies multiples du même document sont sauvegardées à différents niveaux du cache.

1.2.3 Adaptive caching :

Adaptive caching [1,2,4,6,27] explore le problème du cache comme moyen d'optimiser la diffusion des données. La première idée de Adaptive caching a été proposée par L. Zhang et al [2]. Dans cette approche, des groupes de caches forment des mailles (Mesh), qui se chevauchent. Les groupes de caches sont auto-configurables, et s'adaptent aux changements réseaux. Dans [27], S. Michel et al ont défini les mécanismes de cette architecture et fournissent la stratégie de déploiement.

Adptive caching [1] emploie le Cache Group Managment Protocol (CGMP). CGMP spécifie comment les mailles sont formées, et comment les caches rejoignent et quittent ces mailles. En général, les caches sont organisées en groupes multicast chevauchés, et utilisent un vote et la technique de feedback pour estimer le gain avant d'admettre ou d'exclure un membre du groupe.

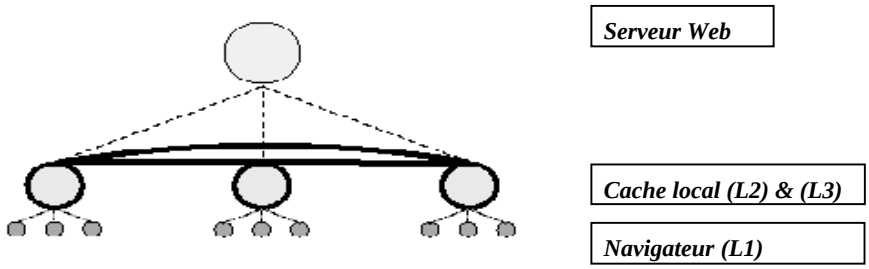

Figure 1.3. Structure en maille de proxy caches coopératives
(Lignes en gras indique la communication des caches L3).

Chacun des systèmes de caches coopératifs cités auparavant est basé sur un protocole qui permet de communiquer avec les autres caches de la communauté de coopération. Ce protocole définit aussi la manière de rechercher les objets dans les caches voisins. La section suivante s'intéressera aux protocoles de communication inter cache. Nous décrivant le principe de fonctionnement de chacun d'eux.

1.3 Protocoles de Communication inter-caches :

Les systèmes de caches tendent à être composé de multiples caches distribués pour améliorer l'extensibilité et la disponibilité du système. Afin de permettre une telle fonctionnalité un protocole de communication entre les caches coopératifs est nécessaire. On trouve dans la littérature plusieurs systèmes de communication inter cache. Nous détaillerons juste après les plus important :

1.3.1 Internet Cache Protocol (ICP) :

Les chercheurs du projet Harvest [28], ont développé un protocole de communication entre les caches appelé ICP (Internet Cache Protocol) [13,14]. Les groupes de caches peuvent bénéficier du partage des données entre eux, de la même

manière qu'un groupe d'utilisateur partage un cache. ICP fournit des méthodes rapides et efficaces pour la communication inter-caches, et il offre des mécanismes pour établir des caches hiérarchiques complexes [14].

ICP est principalement utilisé dans une maille de cache, pour trouver des objets Web spécifiques dans le voisinage d'un cache. Il donne des indications sur l'emplacement des objets Web [26]. A la demande d'un objet qui n'est pas caché localement, le serveur proxy diffuse (multicast) des requêtes ICP (ICP_QUERY) sur tous ces voisins (avec lesquels il est configuré à fonctionner). Les voisins renvoient des réponses ICP indiquant un « Hit » ICP_HIT[1] (le voisin possède l'objet demandé) ou un « Miss » ICP_MISS[2] (le voisin ne possède pas l'objet demandé).

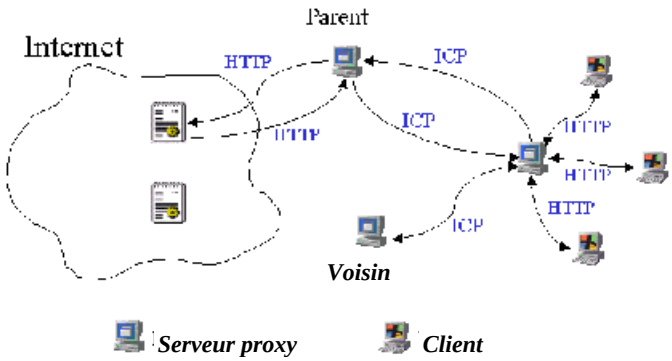

Figure 1.4. Le protocole de communication inter caches ICP.

Un message ICP est constitué d'un entête de taille fixe (20 octets) et de l'URL [14]. Dans la pratique ICP est implémenté au dessus de UDP, pour accélérer les messages requête/réponse échangés. Il ne contient aucune information des entêtes HTTP associés à un objet. Les entêtes HTTP doivent contenir des contrôles d'accès et des directives de cache.

En dépit de son succès et son utilisation répandue, ICP n'est pas vraiment un protocole très extensible. Dés que le nombre de proxies coopératifs augmente, le nombre de message émit pour retrouver un objet devient rapidement prohibitif. Une étude effectuée par Cao et al [9] a montré que les messages ICP augmentent le trafic UDP par un facteur de 73 à 90%.

1.3.2 Cache digests : (Résumés des caches)

Cache digest est une technique d'optimisation pour les caches coopératifs. Elle permet aux proxies de diffuser des informations sur leurs contenus disponibles aux voisins dans une forme compacte. Ce protocole est proposé par A. Rousskov et al [31], comme une alternative de ICP.

Cache digest contient, en format compressé, une information indiquant si une URL particulière est dans le cache ou non. La forme compacte de toutes les clés du cache est obtenue en utilisant la technique de *bloom filter* [9,31] *(voir Chapitre 3)*. Avec cette approche, les serveurs de cache s'échangent leurs résumés périodiquement.

[1] ICP_HIT : Pour indiquer que le cache possède l'objet demandé.
[2] ICP_MISS : Pour indiquer que le cache ne possède pas l'objet demandé.

Quand un cache reçoit une requête pour un objet (URL) de son client, il peut utiliser les résumés de ces voisins, pour trouver celui qui détient cet objet.

Pour réduire la taille du cache digest, les auteurs [31] ont proposé d'utiliser une partie du codage MD5 [8] (8 bit seulement à la place 128 bits) construit à partir de l'URL, et quatre fonctions de hachage pour le positionnement des bits à 1. Un autre point important est à signaler : chaque cache détermine la taille de son résumés indépendamment de ces voisins.

Cache digest a été implémentée au-dessus de Squid 1.2. Rousskov et al [31] ont fait des comparaisons entre Cache digest et la version Squid (à base de ICP). Ils ont trouvé que le temps de service moyen est amélioré de 100 millisecondes et plus, quand ils utilisent Cache digest à la place de ICP. De plus, avec des paramètres adéquat (période de mise à jours), cette technique consomme moins de bande passante. Cependant, tandis que ICP génère un flux stable de petits paquets, le volume de transfère de Cache digest est un peu élevé. En conclusion, cette approche n'est pas vraiment convenable sous des réseaux à faible débit, ou congestionné.

1.3.3 Summary cache :

Dans la même période ou Rousskov et al [31] étaient entrain de développer Cache digest, P. Cao et al [9] de l'université de Wisconsin, Madison étaient en train de développer le Summary cache. Cette approche est très similaire à Cache digest.

Dans Summary cache, chaque proxy garde un résumé compact des répertoires de cache, de tous les autres caches coopératifs. Quand un *MISS*[3] survient, le proxy enquête tous les résumés de ces voisins (localement), pour voir si la requête peut donner un *HIT*[4] dans d'autres proxies voisins. Si un tel cas se présente, le proxy envoie des messages de requête seulement aux proxies qui paraissent prometteurs d'après leurs résumés. Comme Cache digest, Summary cache n'est pas exact à tout moment (faux HIT vs faux MISS). Un *faux HIT*[5] se produit quand le résumé indique un HIT, alors que l'objet n'est plus dans le cache. Dans le cas d'un faux MISS, le document n'est pas caché dans certains caches, mais leur résumé indique qu'il l'est. Ces erreurs affectent le *Hit ratio*[6] total et le trafic inter-proxies, mais pas la validité du système de caching [9]. Les résumés des répertoires sont stockés en utilisant la technique du bloom filter [9,31] *(voir Chapitre 3)*.

Les simulations effectuées par P. Cao et al [9], ont montré que le protocole *Summary cache* réduit les messages inter-proxies par un facteur de 25 à 60% ; réduit la consommation de la bande passante de plus de 50%, et élimine entre 30% à 95% de la charge CPU, par rapport à Squid. Summary cache a été implémentée pour accroître ICP dans Squid 1.1.13. Cette implémentation est publiquement disponible sur [32].

A première vue, on ne remarque pas une grande différence entre Cache digest et Summary cache. En réalité la différence majeure entre eux est que la mise à jour des résumés (summaries) s'effectue ici en utilisant la technologie Push (ie : le serveur doit

[3] Miss : le document demandé n'existe pas dans le cache.
[4] Hit : le document demandé existe dans le cache.
[5] En réalité c'est un Miss, alors que la table utilisée indique le contraire. Donc elle donne une fausse information.
[6] Hit ratio : le nombre de bites servis du cache sur le nombre total d'octets échangés.

maintenir l'état de tous ces fils). C'est d'ailleurs l'inconvénient de cette approche, alors que pour le Cache digest, il y a un échange périodique des résumés.

1.4 Méthodes de remplacement de cache :

La stratégie de remplacement des documents joue un rôle important dans la gestion des objets du cache. Elle permet de contrôler la taille de ce dernier, en excluant les objets qui sont jugés non intéressant.

Il existe plusieurs algorithmes de remplacement [11,7]. La différence entres ces algorithmes se situe dans la manière d'expulser les documents. Les algorithmes les plus connus sont :

1. ***LRU*** (least recently used) : expulser le document qui a été demandé le moins récemment ;
2. ***LFU*** (least frequently used) : expulser le document qui n'est pas accédé fréquemment;
3. ***SIZE*** : expulser le plus grand document ;
4. ***LRU à seuil*** : la même chose que LRU, sauf que les documents dépassants un certain seuil ne sont pas cachés ;
5. ***Log (Size) + LRU* :** expulser le document qui a la plus petite valeur de Log (size), et appliquer la stratégie LRU pour les documents ayant la même valeur de Log (size) ;
6. ***HYPER-G*** : un affinement de LFU avec prise en compte de la taille et de la dernière date d'accès ;
7. ***PitKow/Recker***: supprimer les documents qui sont LRUs, sauf si tous les documents sont accédés le jour même (aujourd'hui), dans ce cas supprimer le plus grand ;
8. ***Lowest Latency First*** : supprimer les documents avec le plus petit temps de téléchargement;
9. ***HYBRID****:* il vise à réduire la latence totale. Une fonction est utilisée pour calculer le poids des documents, celui qui à la plus petite valeur sera expulsé ;
10. ***Lowest Relative Value***: inclut la taille pour calculer la valeur qui estime si un document est gardé en cache (expulser celui qui a la plus petite valeur). Le calcul est basé sur des analyses de la trace des données.
11. ***GreedyDual-Size*** : combine la localité, la taille et le temps de latence pour atteindre une meilleure performance.

En général, l'algorithme de remplacement décide quel document sera caché et lequel sera remplacé. Il y a plusieurs paramètres qui rentre en jeu : les liens chers, document traversant des réseaux encombrés, ... L'algorithme le plus utilisé est le LRU (least recently used), il est très simple, mais il ne prend pas en compte la taille du fichier et le temps d'attente.

12. cache sémantique :

La majorité des systèmes de cache proposés [11,7,45] utilisent des méthodes de remplacement de cache qui sont basées sur des informations opérationnelles, et elles

ne prennent pas en compte la sémantique des informations liées aux contenus des documents.

Nous pensons qu'utiliser les informations sémantiques et contextuelles des documents peut être une meilleure alternative qui fournit aux utilisateurs un outil pour la recherche des documents. En effet, l'existence de thèmes chauds « Hot spot », correspond à des thèmes beaucoup plus demandés que d'autres à un instant donné (souvent pour des raisons liées à l'actualité), joue un rôle très important sur le distribution des requêtes qui suit en général une loi Zipf [7], ce qui signifie que les requêtes sont concentrées sur un petit nombre de documents très populaires.

Cette stratégie est a été développée par l'équipe LIRIS de INSA de Lyon. Pour plus de détail se référer à l'article présenté par L. Brunie et al [7].

C'est la technique de remplacement des éléments dans le cache que nous utiliserons, sans toutefois avoir apporter de nouveautés à celle-ci. Nous la détaillons ici succinctement pour une meilleure compréhension globale du système.

12.1*. *Indexation des documents : L'information attachée à un document est généralement liée à un ou plusieurs thèmes : science, sport, politique,…La tâche d'associer un document à un de ces thèmes s'appelle *indexation*. Dans une telle approche, les documents sont classés dans un certain nombre de thèmes. En subdivisant ces thèmes de manière hiérarchique, on obtient une hiérarchie d'indexation qui permet de caractériser chaque document.

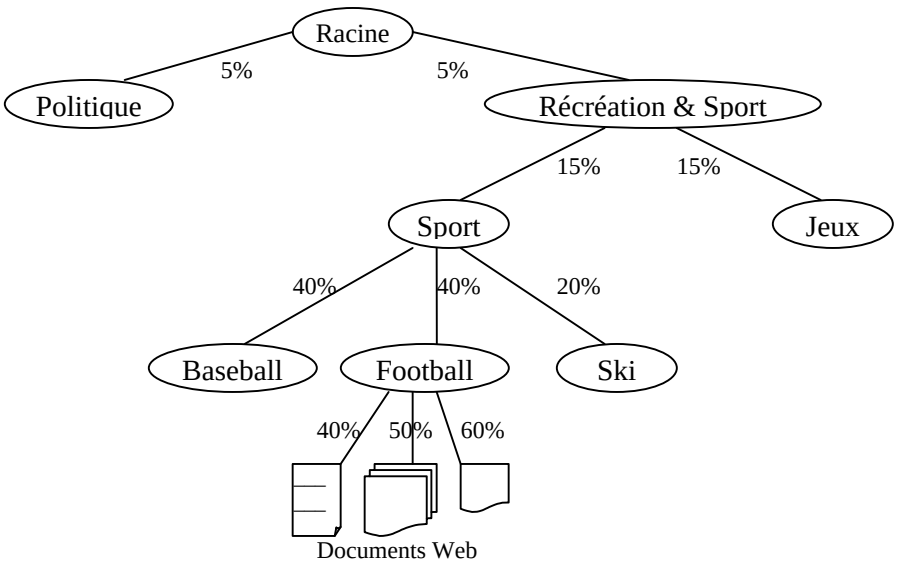

Figure 1.5 Représentation graphique de la structure des thèmes.

12.2 Calcul de la température : Le principe de la température permet d'optimiser les politiques de remplacement de cache implémentées par les proxies. En effet, les documents qui sont liés à des thèmes et des sujets chauds auront plus de chance de rester dans le cache que les autres documents.

Le calcul de la température se produit après un nombre de requêtes. Le nombre d'accès à un document entre deux calculs consécutifs sert à calculer la température de ce dernier. Les documents ayant fait l'objet d'au moins une requête depuis le dernier

calcul de température verront leur température augmentée, alors que les autres qui ne sont pas accédés seront refroidis.

Pour plus de détails sur l'algorithme d'indexation, se référer à [7].

Un autre point très important à signaler est la cohérence des données cachées avec la version courante qui se trouve au serveur. Sans prévoir des mécanismes de vérification, on peut tomber sur des documents non frais. La section suivante va aborder ce sujet avec plus de détail.

1.5 Mécanismes de validation/réplication de documents :

Les caches proxies coopératifs sont des groupes de caches qui partagent des objets cachés et coopèrent entre eux, pour faire le même travail comme un seul cache web. Cependant, le cache a le désavantage de retourner des documents non frais. Cela signifie que le document n'est pas consistant avec la version qui est sur le serveur. Ce problème peut être détourné en utilisant les mécanismes de validation/réplication.

La validation/réplication, est un mécanisme qui permet de distribuer et de maintenir la consistance des documents, c'est un point très important pour l'implémentation d'une architecture de distribution de documents. On trouve deux grandes stratégies pour des documents dans un système de cache [21], le mécanisme *pulling* et le mécanisme *pushing*.

Dans le mécanisme *pulling*, chaque cache interroge périodiquement le serveur origine (ou bien le cache du niveau supérieur selon l'architecture définie). Dans le mécanisme de *pushing,* les caches n'ont pas besoin d'interroger le serveur périodiquement pour la fraîcheur du document, ce dernier distribue les documents modifiés aux caches intéressés ou abonnés. A l'intérieur de ces stratégies, on peut définir plusieurs mécanismes de validation/réplication, qu'on regroupe en trois blocs : *Time Live (TTL*), *Validate Verification* et le *Callback* (invalidation).

1)-Time To Live (TTL) : les serveurs ajoutent une marque de temps (time stamp) pour chaque document requêté par le cache. Cette marque définie la période de temps durant laquelle le document est encore valide. Par exemple le champ *Expire* du protocole HTTP/1.0 est un bon exemple de cette approche.

2)-Validate Verification : ce mécanisme est lié à TTL. Le cache reçoit un document avec une marque de temps qui indique la date de la dernière modification de ce dernier. Quand un client demande ce document, le cache envoie un message de vérification au serveur, en incluant la marque de temps enregistrée dans le document. Le serveur valide la marque de temps de ce document avec la date de dernière modification du document, et envoie un message de « No-Modification » (le document n'a pas été modifié), ou bien le nouveau document avec la nouvelle marque de temps. Cette approche est implémentée dans HTTP/1.0 en utilisant la balise If-Modifie-Since (IMS).

3)-Callback (invalidation) : chaque serveur garde une trace de tous les caches qui ont demandé une page donnée, et quand cette page change, il avertit ces caches. Cette approche présente une faible extension, vu qu'elle nécessite la sauvegarde de la liste

de tous les demandeurs du document, ainsi qu'une augmentation de la charge du système et du réseau, induite par l'obligation de contacter tous les demandeurs d'un document à chaque modification. Cependant, ce problème d'extensibilité peut être surmonté en utilisant la diffusion (multicast) pour la transmission de l'invalidation (ICM), en attribuant un groupe multicast pour chaque page. La diffusion résout le problème d'extensibilité au niveau du serveur, mais il en crée d'autre au niveau des routeurs [21].

1.6 Performances des systèmes de caches coopératifs :

Les systèmes de caches coopératifs ont prouvé leurs performances dans plusieurs domaines : système de fichier, mémoire virtuelle, et dans des réseaux locaux.

D'après des études effectués par Wolman et al [12], les auteurs se sont posés la question suivante : *est ce que n'importe quelle forme de coopération de cache fournit un bénéfice significatif au delà de deux niveaux* ?

Ils ont montré que le bénéfice de la coopération de cache est dû simplement à l'augmentation du nombre d'utilisateurs partageant un ensemble de documents Web. De plus, la performance s'améliore seulement au delà d'une certaine taille de population. Les auteurs ont monté aussi qu'une grande communauté de proxies gagne peux en coopérant entre eux, et qu'il est plus intéressant d'avoir des petites communautés qui coopèrent à petite échelle. En conséquent, ils ont trouvé qu'il n'y pas de besoin à une large coopération de proxies. Cette étude s'est adressée à la question fondamentale dans le domaine des caches coopératifs : quelle est l'étendue et la nature du partage des documents sur le Web?

1.7 Conclusion :

Comme les services Web sont devenus de plus en plus populaires, les utilisateurs souffrent des problèmes de surcharge des serveurs et de congestion du réseau. Plusieurs recherches ont été faites pour améliorer la performance du Web. Le cache est reconnu comme étant une technique efficace pour diminuer la congestion des serveurs, et de réduire le trafic réseau, et de cette façon, minimiser le temps de latence des utilisateurs.

Exploiter la coopération entre les caches permet d'accroître les performances du système de cache, plutôt que d'avoir plusieurs proxies qui accèdent aux mêmes ressources de manière indépendante. La clé d'efficacité de n'importe quel schéma de coopération réside dans le protocole de communication de cache.

Le prochain chapitre fera une introduction aux réseaux actifs, leur apparition, leur évolution, leur principe et leurs avantages.

Chapitre 2

Réseaux actifs

2.1 Introduction :

Le concept de réseau actif est né dans le milieu des années 90 aux Etats-Unis. Il s'est rapidement étendu à l'Europe et constitue aujourd'hui un sujet de recherche universitaire et industriel très actif.

Bien que de nombreux progrès aient été réalisé ces dernières années pour rendre la configuration des équipements de réseaux plus souple, ceux-ci restent relativement figées et spécifiques à un type d'application (comme la téléphonie par exemple).

L'extensibilité de ces réseaux reste limitée. Ils ne fournissent que les services pour lesquels ils ont été conçus. Or les opérateurs et fournisseurs de services ont un besoin crucial de dynamicité pour répondre dans un délai très court aux besoins des usagers. Malheureusement, les services de base fournis par les protocoles existant du réseau sont mal adaptés à ces applications d'un genre nouveau.

Pour y remédier des solutions à ce problème sont apparues véritablement à partir de 1995 avec la création de nombreux atelier (*workshop*) consacrés aux ***réseaux programmables***.

Définition :

Un réseau programmable est un réseau de transmission de données ouvert et extensible disposant d'une infrastructure dédiée à l'intégration et à la mise en oeuvre rapide de nouveaux services sur l'ensemble de ces composants.

Un réseau traditionnel est un réseau de transport de données qui possède un nombre restreint et fixé de services implantés dans les équipements et qui n'offre aucun moyen d'en ajouter. Par conséquent il est impossible de modifier dynamiquement le comportement global du réseau. Au contraire, un réseau programmable est un réseau de transport de données étendu par un environnement de programmation à l'échelle du réseau comportant un modèle de programmation des services, des mécanismes de déploiements et un Environnement d'Exécution (EE).

En 1996, sur une proposition de Tennennhouse et al. [10] apparu le concept de ***réseaux actifs***.

Définition :

Un réseau actif est un réseau dans lequel tout ou une partie de ses composants dans les différents plans (signalisation, supervision, données) sont programmables dynamiquement par des entités tierces (opérateurs, fournisseurs de services, applications, usagers).

Cette approche est plutôt issue des travaux sur l'insertion de services applicatifs dans le réseau Internet. Elle étend le concept de programmation en ouvrant les

équipements de l'ensemble du réseau et en partant du principe que tout usager peut concevoir et déployer au travers d'interface de programmation de nouveaux services. Ces nouveaux services peuvent alors être utilisés de manière dynamique dans le réseau.

2.2 Approche des réseaux actifs :

Dans la famille des réseaux actifs il existe deux classes [3] : l'une se base sur une approche ***paquet actif***, l'autre sur une approche ***noeud actif***.

2.2.1 L'approche paquet actif :

Elle permet de transporter le code des services dans le même flux que les données traitées par ce service. On parle aussi d'***approche intégrée***. Classiquement le code du service est émis avant l'arrivée du flux de paquets de données. Mais l'émission du code est effectuée dans le flux usager. Et dans un cas extrême, chaque paquet de données contient le code du service à y appliquer. Généralement la vie du service est liée à celle du trafic affecté.

2.2.2 L'approche noeud actif :

Les services sont également déployés dynamiquement sur les noeuds du réseau mais hors du flux de données utilisateurs. Cela se rapproche fortement du concept de réseau programmable. La différence avec ce dernier repose sur le fait que dans l'approche noeud actif, les composants de service sont déployés physiquement sur les noeuds du réseau, le plus souvent grâce à des techniques de code mobile. Dans les réseaux programmables, la logique de service et son implantation ne nécessitent la plupart du temps pas de déploiement sur les noeuds, celle-ci étant exécutée au sein de l'environnement de programmation du réseau. Cette programmation externe est aussi appelée ***approche discrète***.

L’une des plates formes qui offre la possibilité de mettre en place des réseaux actifs, composés d’un ensemble de nœuds actifs est la plate forme TAMANOIR [20]. C’est l’une des rares plates formes qui est adaptée aux contraintes de haute performances lors du transport et du traitement à la volée des flux [18]. Complètement écrite en JAVA elle propose un environnement d’exécution portable. Nous donnerons dans la section suivante plus de détails sur cette plate forme.

2.3 Plate forme TAMANOIR :

L’équipe RESO de l’INRIA a développé la plate forme *TAMANOIR*, qui permet de déployer de nouveaux services sur le réseau. Nous allons utiliser cette plate forme pour mettre en ouvre notre système de coopération de cache, et intégrer la fonctionnalité de cache aux routeurs actifs, afin d’ajouter de l’intelligence dans le traitement des requêtes.

2.3.1 Description d'un nœud actif :

Un nœud actif est un routeur, une passerelle ou un proxy qui reçoit des paquets du réseau, effectue des opérations sur ces derniers et les transmet au prochain nœud actif. Les données sont transportées sous forme de paquets au format ANEP [17].

Un nœud TAMANOIR est composé essentiellement de deux composants principaux, *TAMANOIRd* et *ANM* (Active Node Managment). Le premier composant *TAMANOIRd*, est un démon qui tourne sur un nœud actif *TAN* (Tamanoir Actif Node) et qui agit comme un routeur programmable actif. Une fois installé sur un nœud, il fait la liaison avec les nœuds actifs voisins dans le réseau. Le nœud TAN reçoit et envoie les paquets en effectuant des traitements sur les paquets marqués comme étant actifs en faisant appel aux services utilisateurs. En effet le démon *TAMANOIRd* redirige les paquets actifs reçus vers le service adéquat qui est lancé sous forme d'un processus léger (thread). Le paquet résultant de ce traitement est soit envoyé au prochain nœud *TAN*, soit au destinataire final.

Le second composant est *ANM* (Active Node Manager), il est chargé principalement de mettre à jour la table de routage locale, ainsi que télécharger de nouveaux services, qui ne sont pas présent sur le nœud *TAN*.

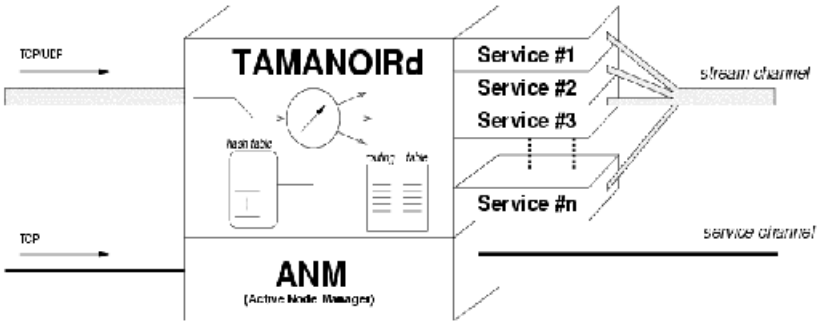

Figure 2.1. Structure d'un nœud TAMANOIR

2.3.2 Déploiement de services :

Ce mécanisme permet d'injecter de nouveaux services dans le réseau actif. A la réception d'un paquet faisant appel à un service qui n'existe pas localement sur le nœud *TAN*, le *ANM* s'occupe de la récupération de ce nouveau service. On trouve deux stratégies de déploiement : la première consiste à déployer le service de voisin en voisin, et la deuxième solution consiste à utiliser un serveur de services Tamanoir (Broker). Nous pouvons voir ces deux mécanismes de déploiement à travers la figure 3.2.

1- *Service Broker* : c'est une solution centralisée, dans laquelle un serveur Web dédié à la distribution des services Tamanoir. Les services seront localisés par des URL, de cette façon les applications déploient des services que les nœuds actifs peuvent télécharger sur ce serveur dédié.

2- *Déploiement de voisin en voisin* : cette solution contrairement à la première est complètement décentralisée. Le service est déployé à la volée au fur et à mesure de la traversée des routeurs actifs. Si un nœud TAN ne possède pas un service donné, le

ANM (Active Node Manager) demande le chargement du service demandé du dernier routeur actif qui a traité le paquet, ce dernier renvoie le code du service invoqué.
Après le déploiement du service, il est lancé pour traiter le paquet.

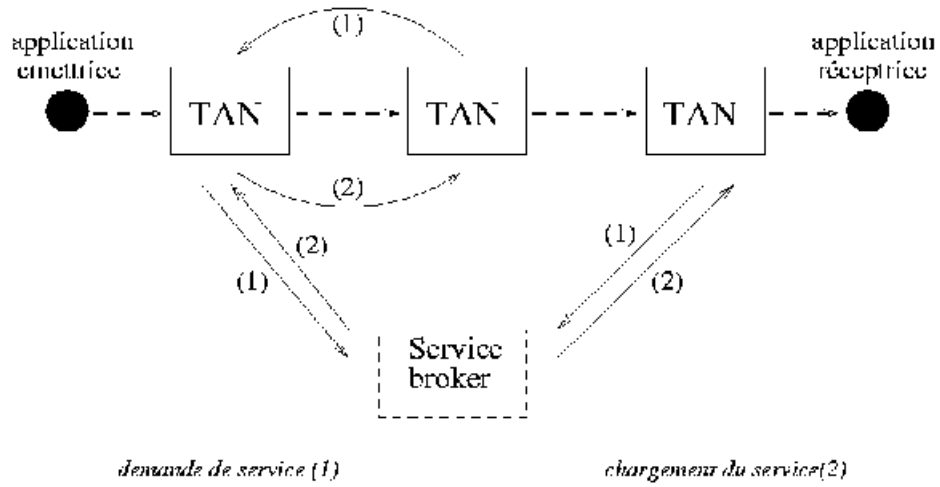

Figure 2.2. Stratégie de déploiement de services.

2.4 Réseaux actifs et système de cache :

Les réseaux actifs sont considérés comme une piste prometteuse pour accroître la capacité du réseau et de nouveaux services sur le réseau, en permettant aux utilisateurs d'injecter des programmes personnalisés dans les nœuds actifs du réseau. Le but visé par les différentes études effectuées dans ce domaine [3,18] est de mettre en place des mécanismes qui accélèrent le déploiement de nouveaux protocoles et services, ainsi qu'augmenter la flexibilité et l'innovation dans les réseaux, sans changer les structures actuelles.

Très peu d'études se sont adressées au problème de cache sous un environnement actif. Ce domaine et un nouvel axe qui s'inscrit dans l'axe de recherche du présent stage de DEA.

2.5 Conclusion :

Les réseaux actifs sont considérés comme une piste prometteuse pour accroître la capacité du réseau, en permettant aux utilisateurs d'injecter des programmes personnalisés dans les nœuds actifs du réseau. Ils offrent des mécanismes qui accélèrent le déploiement de nouveaux protocoles et services, ainsi qu'augmenter la flexibilité et l'innovation dans les réseaux, sans changer les structures actuelles, ni faire attendre la normalisation d'un nouveau protocole.

Ceci nous mène à penser à développer un protocole de coopération pour des systèmes de caches, en intégrant la fonctionnalité de proxy cache aux routeurs actifs, et en profitant de la souplesse des réseaux actifs.

Chapitre 3

Vers la conception d'un routeur actif proxy cache

3.1 Introduction :

L'objectif qu'on a fixé à travers ce travail est d'ajouter un peu d'intelligence dans le routage des requêtes au niveau des routeurs actifs, en utilisant des caches coopératifs. Ajouter une telle fonctionnalité à un routeur actif nécessite la définition d'un protocole de coopération, qui a moins de complexité que les autres systèmes de coopération, et qui n'introduit pas un échange massif et fréquent des informations de coopération entre les caches pour mettre en place un tel système. L'intérêt d'un protocole est, d'un côté de réduire la charge du réseau et le nombre de paquet qui circulent, d'un autre d'alléger la tâche du routeur actif.

Le système proposé effectue deux fonctions de base principales pour permettre l'instauration de la coopération de cache : ***Découverte*** et ***Délivrance***.

La ***Découverte*** est la manière de localiser les objets cachés dans une communauté de caches coopératifs. Le système devra fournir à un groupe de cache un mécanisme de localisation rapide et souple[7] des objets dans la maille de coopération. La deuxième fonction quant à elle, est la ***Délivrance*** : comment délivrer les objets du cache vers les caches voisins ? Elle définit un protocole d'échange d'objets entre les caches coopératifs.

Les noeuds actifs ont la capacité de faire des traitements sur les paquets qui les traversent, en plus de la fonction habituelle (envoie des paquets entrants vers la meilleure destination). Ces traitements doivent être des petits services qui n'alourdissent pas le routeur, et ne perturbent pas son bon fonctionnement. De plus, comme n'importe quel routeur classique, l'espace de stockage est limité. Face à toutes ces contraintes nous somme obligés a traité les routeurs actifs avec prudence, et de faire face à ces limites.

L'originalité de notre système et l'intégration de la fonction de cache à des routeurs actifs. Nous détaillerons ci-dessous le modèle de notre système.

3.2 Le système de cache coopératif :

Les caches sont organisés en hiérarchie de 2 niveaux, comme le montre la figure Figure 3.1. Au premier niveau se trouvent les caches fils, qui jouent le rôle de proxies cache, et qui permettent de faciliter et accélérer l'accès des clients. Au niveau

[7] Comme c'est un routeur actif qui effectue cette tache, cette dernière ne devra pas trop le surcharger, sachant que son rôle principal est le routage des paquets.

supérieur se trouvent les caches parents qui jouent le rôle de coordinateurs ou d'administrateurs.

Au début de ce travail, nous avons pensé à faire une coopération entre des caches proxies utilisateur, cette solution permettant d'avoir une grande communauté de coopération. Les caches proxies utilisateur sont installés sur les machines des utilisateurs et jouent à la fois le rôle de proxy et de cache en même temps. Deux problèmes surviennent :

1- Un utilisateur n'est pas disponible à tout moment (la machine peut être éteinte) ;

2- Les restrictions imposées par les Firewalls : ainsi deux proxies utilisateurs n'ont pas toujours la possibilité de communiquer directement. De ce fait, nous avons mis les proxies caches fils à un niveau qui leur permettent de communiquer directement.

La structure hiérarchique de cette architecture est utilisée pour distribuer les informations concernant la localisation des documents et non pour sauvegarder des copies de ces documents, comme c'est le cas pour les caches hiérarchiques. Nous voulons à travers cette structuration mettre en place un système de cache distribué.

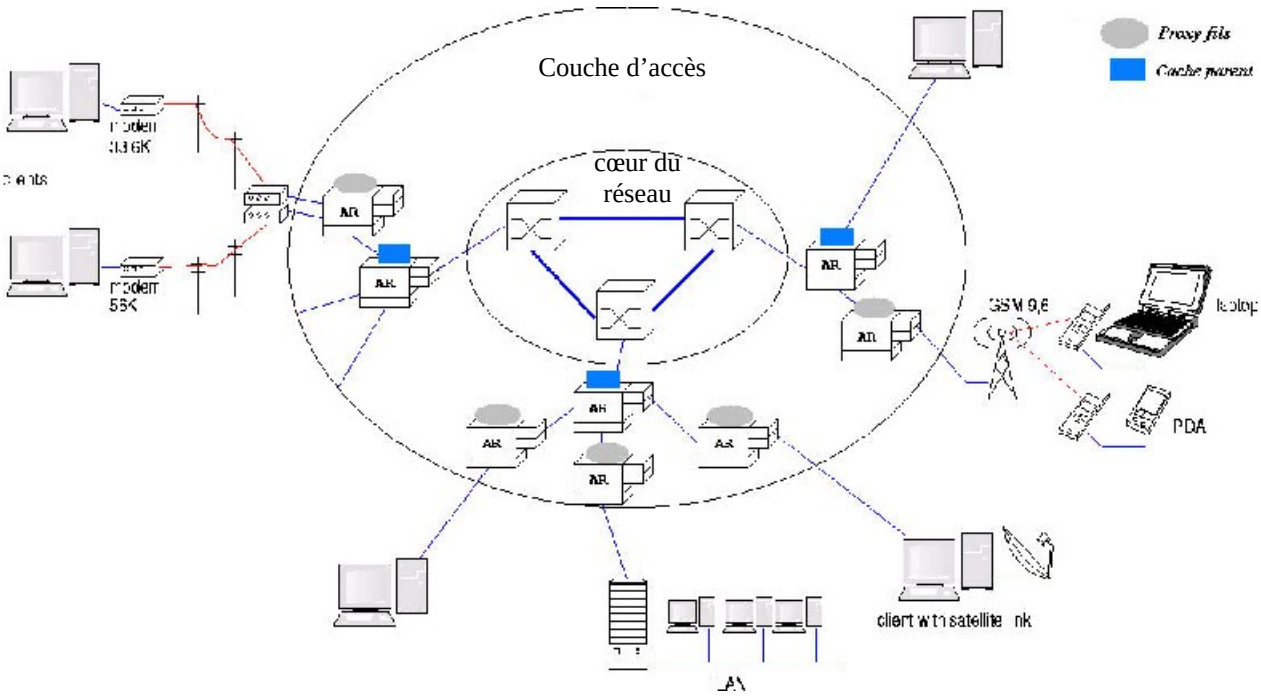

Figure 3.1. Schéma de du réseau de coopération.

Nous allons présenter par la suite l'architecture du système de cache, ses différentes fonctionnalités, ainsi que son principe de coopération.

3.3 Les tables miroirs :

Avant de détailler le principe de fonctionnement du système, nous commençons à donner quelques terminologies et définir les structures d'échange d'informations utilisées par les caches coopératifs. Celle-ci est basée sur une codification compressée des informations contenues dans les caches. Nous rappelons que nous travaillons au coeur d'un routeur actif, et l'espace de stockage est un facteur important. Nous utiliserons la technique du *bloom filter* [9] afin de gagner dans l'espace de représentation.

3.3.1 Bloom filter :

C'est une méthode qui a été proposée par Burton Bloom en 1970, pour représenter un ensemble de *n* éléments E = { a_1, a_2, …. ,a_n } (aussi appelé clé) afin de supporter les requêtes sur des ensembles (i.e. requête "est ce que x $\in$ *Y* ?").

Le principe de cette technique est d'allouer un tableau de bits *v* de taille *m* (initialement tous à 0), qui sera utilisé comme filtre. On choisit après *k* fonctions de hachage indépendantes h_1, h_2, …,h_k, les valeurs de ces fonctions varient entre [0 .. *m*]. Pour ajouter un élément *a* de E au filtre *v*, on calcule un ensemble de *k* valeurs {$h_1(a)$, $h_2(a)$, . . , $h_k(a)$}. Ces valeurs spécifient les positions des bits dans *v* qui seront mis à 1 (un bit particulier pourra être mit à 1 plusieurs fois). La figure suivante illustre ce principe clairement :

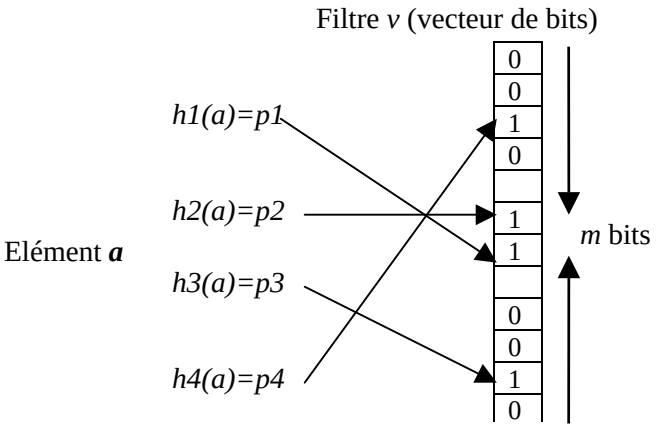

Figure 3.2 Principe de bloom filter à quatre fonctions de hachage.

Pour examiné si une entrée *b* est présente dans le filtre *v*, nous calculons les *k* positions {*h(b), h(b), . . , h(b)*} par les mêmes fonctions de hachage (h_1, h_2, …,h_k). On examine les bits correspondants dans le filtre *v*. Si l'un de ces bits est à zéro, alors certainement b ne fait pas partie de l'ensemble E, sinon il y a une probabilité que *b* soit présent. On peut se tromper avec une certaine probabilité. Ceci est appelé faux HIT.

Dans le bloom filter on utilise les termes *HIT* et *MISS* pour indiquer si les bits du filtre prédisent qu'un objet donné est présent dans le cache ou non. Alors que les termes *vrais* et *faux* décrivent la validité de la prédiction. Donc on trouve :

Vrai Hit : le filtre prédit correctement l'existence d'un objet dans le cache.
Faux Hit : le filtre prédit incorrectement l'existence d'un objet dans le cache.
Vrai MISS : le filtre prédit correctement l'absence d'un objet dans le cache.
Faux MISS : le filtre prédit incorrectement l'absence d'un objet dans le cache.

Le bloom filter a toujours un *Faux Hit* non égal à zéro. Ceci est le pris payé pour sa représentation compacte. La taille du filtre *m*, et le nombre de fonction de hachage *k* jouent un rôle très important pour déterminer si la correspondance : tous les bits sont à 1= Hit, est correcte ou non. Ils doivent être choisi de telle manière que le faux Hit soit acceptable. On observe que en insérant *n* éléments dans une table de *m* éléments. L'étude mathématique de cette méthode [9] prouve que la probabilité qu'un bit reste à 0 est : $p_0 = \left(1 - \frac{1}{m}\right)^{kn}$

Donc la probabilité de faux HIT est : $p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$

Le coté droit est minimisé pour $k = \frac{m}{n} \ln 2$. Ce résultat sera pris en compte dans la partie de tests que nous allons effectué.

3.3.2 Structure de les tables miroirs

Nous allons parler maintenant de la structure ou l'unité d'échange d'information de base entre les caches fils et le cache coordinateur. Celle ci permet la localisation des documents dans une communauté de caches coopératifs.

Nous nous sommes inspiré des travaux réalisés par P. Cao et al [9] et Rousskov et al [31], qui ont proposé une forme très compacte pour coder et échanger les URLs des objets présents dans les caches, la technique de *bloom filter* (cette technique a prouvé son efficacité dans le domaine de base de donnée). Nous avons défini la *table miroir*. Cette table est une structure qui contient plusieurs informations : l'adresse du cache reflété par cette table, un tableau de bits (pour le bloom filter), la taille du tableau de bits, le nombre de fonctions de hachage, ainsi que des statistiques sur le nombre de *faux HITs*.

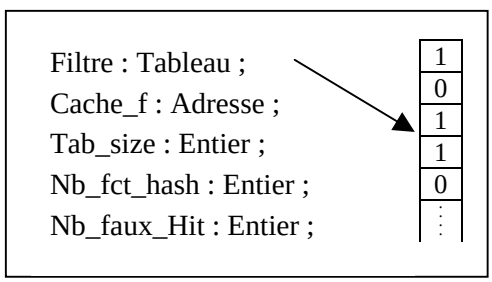

Figure 3.2. Structure de la table miroir.

Caque proxy cache parent possède une image (d'où l'appellation de table miroir) de tous ces caches fils. A la réception d'une requête, le cache extrait l'URL de l'objet demandé, puis il vérifie l'existence de cet objet dans les tables miroirs qu'il possède. Apres ce contrôle, le proxy contacte le cache propriétaire de la première table miroir qui répond à vrais à la requête (indique un *Hit*).

3.4 Construction de la table miroir :

Si l'ajout est possible dans le bloom filter standard, la suppression par contre est impossible. Chaque bit qui est mit à 1 dans le filtre, peut être positionné par une ou plusieurs entrées (URLs). En conséquent, forcer un bit à zéro après la suppression d'une entrée, peut falsifier la cohérence du filtre, mais le laisser à 1 peut également entraîner une incohérence si c'était la seule entrée qui l'avait positionné. Afin de contourner ce problème, la solution qui permettra la suppression d'une entée dans la table, serait d'utiliser un compteur qui sauvegarde le nombre de fois un bit a été positionné à 1, au lieu d'avoir un seul bit. Cette solution va bien sûr augmenter la taille de la table miroir, mais en contre partie, elle diminue le nombre de *faux HITs*.

Les caches parents (ceux du niveau supérieur de la hiérarchie) possèdent des copies des tables miroirs de leurs caches dépendants. Quand la table miroir devient non consistante (il y a beaucoup de *faux HIT*), le cache père peut demander à son fils une nouvelle copie de la table miroir (mise à jour). Ce mode de coopération engendre un trafic réseau supplémentaire, ce qui peut dégrader les performances du système. Pour remédier à ce problème, les caches fils gardent une structure de données un peu spéciale (table locale) des objets présents dans leurs caches. Ceci est fait en maintenant pour chaque position *p* dans le bloom filter (à la place du bit utilité dans cas standard) un compteur *C(p)* qui sauvegarde le nombre de fois que cette position a été mise à 1. Initialement tous les compteurs sont mis à 0. Quand une nouvelle clé (URL) est ajoutée ou supprimée du filtre local, les contenus des compteurs pointés par les positions concernées seront incrémentés ou décrémentés respectivement.

A partir de la table locale, un cache fils construit sa table miroir pour l'envoyer à son cache parent. Le processus de construction de la table miroir est très simple, il est donné par l'algorithme suivant :

Pour chaque position *p* du tableau de compteur local ***faire***
  Si le compteur *C(p) > 0* ***alors***
    Le bit de la position *p* dans la table miroir est mis à 1 ;
  Sinon
    Mettre le bit à 0 ;

Avec ce mécanisme, les caches parents, implantés sur des routeurs de petite mémoire, ils auront à sauvegarder de petites quantités de données de leurs fils. Le trafic engendré par les mises à jour des tables sera raisonnable (quelques octets), et les caches fils auront des mécanismes pour garder leurs tables miroirs pertinentes.

3.5 Communication inter cache :

Le principe de notre approche est illustré par le schéma de la Figure 3.3. Il est basé sur le protocole de communication inter cache, qui utilise la localisation des objets à l'aide des tables miroirs. Dès l'arrivée d'une requête d'un client, son proxy cache local C1, regarde dans sa table d'index s'il peut satisfaire la requête localement (ie : le cache possède l'objet demandé). Si c'est le cas il retourne cet objet au client demandeur, sinon le cache envoie une requête à son cache parent. Une fois qu'un cache parent reçoit une demande de l'un de ces fils, il consulte ces tables miroirs, si une d'entres elles indique que l'objet requêté est peut être présent[8] dans l'un des caches fils (les autres frères du cache demandeur). Il envoie alors la requête au cache qui possède l'objet, pour lui demander de répondre à la demande de son frère. La réponse se fait en envoyant directement l'objet requêté au cache demandeur (transfert pair à pair).

Laisser le cache qui possède l'objet s'occuper du transfert de ce dernier au cache demandeur, permet d'un côté d'éviter de surcharger les caches des niveaux supérieurs [25], et d'un autre côté, de réduire le nombre de faux HIT (un cache qui répond avec un HIT à un objet, peut ne plus posséder cet objet après un petit laps de temps, car cet objet pourrait être élu pour quitter le cache par l'algorithme de remplacement). Ce principe est illustré par l'exemple suivant :

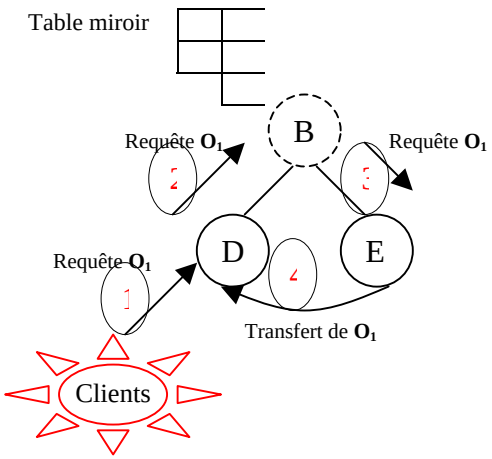

Figure 3.3. Mécanisme de communication inter cache.

Exemple :

A la demande d'un objet $\mathbf{O_1}$ par un client du cache **D**, ce dernier envoie une requête à son parent **B** pour la traiter. Le cache **B** consulte ses tables miroirs, et cherche lequel des caches possède l'objet. Dans notre exemple, **B** trouve (d'après ses tables miroirs) qu'il y a de fortes chances que l'objet $\mathbf{O_1}$ soit présent dans le cache **E**. Dans ce cas, il

[8] La table miroir n'indique pas avec exactitude qu'un objet donné, est présent dans le cache, mais elle donne une probabilité qu'il soit présent ou absent. Cela est du à la compression du format de la table, en utilisant la technique du *bloom filter*, qui peut engendrer des faux HIT/MISS, comme vue en 3.3.1.

envoie la requête à ce dernier, en lui fournissant plus d'informations sur le demandeur initial de l'objet (son adresse ici **D**). Si **E** possède effectivement l'objet $\mathbf{O_1}$, il pourra le transmettre directement à **D**, sans repasser par **B**. Dans le cas contraire (un *faux HIT*), le cache **E** transmet un message à **B**, pour lui indiquer qu'il ne possède pas/plus cet objet.

Ce scénario se présente dans toutes les communautés de caches coopératifs, sous l'autorité d'un cache parent.

3.6 Consistance de la table miroir :

La taille des caches n'étant pas infinie, l'utilisation des méthodes de remplacement d'objet est une obligation pour contrôler la taille du cache, et permettre une bonne gestion, afin d'améliorer les performances de ce dernier [11]. Ceci est d'autant plus important que la qualité de mémoire (disque ou mémoire vive) est limitée sur les routeurs actifs. Une méthode de remplacement se charge d'exclure des objets, pour aménager de la place aux nouveaux éléments requêtés par le cache. Ces transactions vont générer un nombre important de *faux HIT* au niveau des caches parents (ie : la table miroir indique que l'objet est présent chez le fils, alors que ce dernier là peut être exclu de son cache). Cela va influencer la performance globale du système de cache. En effet, les *Faux Hits* engendrent des communications supplémentaires inutiles entre les caches parents et les caches fils, et par conséquent la coopération des caches perd son intérêt.

Afin de remédier à ce problème, nous avons introduit des traitements au niveau des caches. Chaque cache père maintient à jours des informations sur les tables miroirs (plus exactement le nombre de *faux Hit*). Dès qu'un *faux HIT* survient, le cache incrémente la valeur du nombre de faux Hit du cache qui a produit cette situation. Une fois que ce nombre dépasse un seuil α, le cache parent demande au cache concerné de lui transmettre une nouvelle version (copie mise à jours) de sa table miroir considérant alors que sa table miroir n'est plus assez cohérente. Cette solution convient bien dans le cas général, où le cache parent interroge souvent ces fils.

Tant qu'il y a une communication assez importante entre les caches parents et les caches fils, la première solution pourra bien diminuer les effets des *faux Hits*, mais dès qu'un cache fils est peu sollicité, le contenu de son cache local pourra être mis à jours plusieurs fois sans que le cache parent ne soit informé (ie : la méthode de remplacement de cache sur les fils supprime des objets au profit d'autres). Dans cette situation, on remarque bien que la table miroir qui est présente au niveau du cache père ne reflète pas exactement le contenu du cache fils, et on pourra anticiper à envoyer sa table miroir à son cache parent. Une solution à ce problème, consiste cette fois ci à attribuer des traitements aux nœuds fils, qui ressemblent à ceux effectués aux caches parents.

En effet, chaque cache fils contrôle le nombre d'objets qu'il a récupéré et caché dans son cache, ainsi que le nombre d'objets supprimés. Si ce nombre dépasse un certain seuil β, sans que le cache parent ne soit informé, le cache fils anticipe la mise

à jour de sa table, et fait une notification à son parent, et lui envoyant une nouvelle version de sa table miroir, et remettre à zéro le compteur d'opération. Le cache parent, accepte les mises à jour et remet à zéro ces compteurs pour ce fils.

Il faut aussi signaler que le problème de faux hit et un problème incontournable, car il est lié à la structure compressée du bloom filter. Il est paramétré par la taille du filtre, le nombre de fonction de hachage et le nombre d'entrée dans la table.

Contrairement à Cache digest [31] et Summary cache [9], le maintien de la table miroir s'effectue au moment où les caches (fils/parents) jugent que leurs tables miroirs ont une faible pertinence, il n'y a pas des mécanismes de pull/push. Par conséquent, ils choisissent le moment opportun pour lancer la mise à jour.

Si on reprend l'exemple de la communication inter-cache (paragraphe précédant), la mise à jours de la table miroir du côté cache parent et du côté cache fil est comme suit :

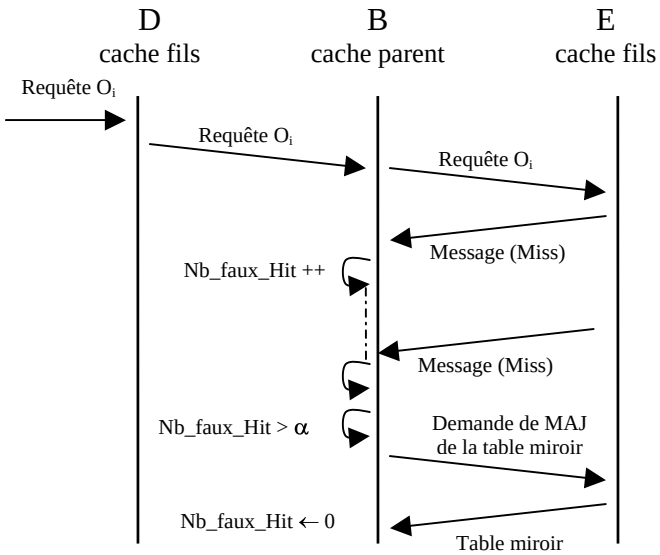

Figure 3.4. Mise à jours de la table miroir côté cache parent.

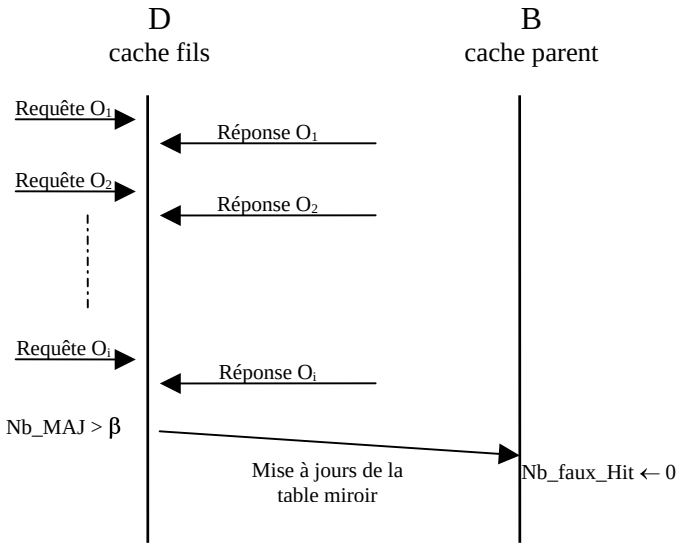

Figure 3.5. Mise à jours de la table miroir côté cache parent.

3.7 Construction de la communauté de coopération :

Cette étape permet la construction du système de coopération, en regroupant les noeuds actifs en communauté et de mettre en place une certaine hiérarchie pour la distribution des responsabilités. Elle se fait de manière statique ou dynamique.

3.7.1 Méthode statique :

Au lancement des nœuds, des initialisations sont nécessaires pour permettre l'instauration du système de caches coopératifs. En effet il faut définir des règles de hiérarchie entre les différents nœuds de la communauté de coopération. Le cache parent se lance en mode *cache parent*, effectue son rôle de routeur actif le plus normalement. Dés qu'un nouveau cache fils qui se lance en mode *cache fils* et veut rejoindre une communauté de caches, il envoie un message d'inscription (service *JointparentS*) au cache parent concerné. Du coté du cache parent, à la réception d'un message d'inscription, le service d'inscription extrait les informations du nouveau nœud et l'ajoute à la communauté des caches coopératifs.

3.7.2 Méthode dynamique :

Contrairement à la méthode statique, une inscription dynamique permet d'ajouter de nouveau cache et de retirer d'autre du système suivant des critères de performances. Cette tache sera effectuée par le cache coordinateur : comme c'est un routeur en même temps, il peut analyser les flux entrants venant d'autres nœuds actifs, s'il juge que l'ajout d'un nœud comme cache au système de coopération, il lui envoie une demande pour rejoindre le système de coopération de cache. Cette demande est sous forme d'un paquet actif qui fait appel au service *QuerryJointparentS*. Le nœud

sollicité a la possibilité de refuser cette demande, ou l'accepter en s'inscrivant à ce cache parent.

3.8 Equilibrage de charge :

Il est souhaitable que le système de cache distribue la charge équitablement à travers les proxies du réseau. Les charges des différents proxies doivent être maintenues à un niveau acceptable, de tel sort qu'aucun proxy ne soit sollicité plusieurs fois pour résoudre les requêtes, alors que d'autres proxies qui peuvent faire la même tache sont en repos. Afin d'éviter ce problème et d'équilibrer la charge des différents proxies, le cache parent distribue le flux des requêtes sur ces fils selon la technique LRU [11] (least recently used). Dans notre architecture, les documents sont cachés dans les proxies feuilles (ie : du niveau 1). Un cache parent peut gérer plusieurs fils qui possèdent le même document, et selon la procédure de recherche de document, si le cache fils ne possède pas le document il envoie la requête à son cache parent. A ce niveau le cache parent consulte ces tables miroirs pour rechercher lequel des proxies fils possède le document requêté. Si on ne tient pas compte de l'ordre de recherche dans les tables miroirs, un proxy donné peut être sollicité à chaque fois que son cache parent à besoin d'un document qu'il possède, alors que même document existe dans un autre cache. On remarque bien que la performance de ce proxy sera dégradée.

Comme déjà cité, nous proposons que le cache parent consulte ces tables miroirs dans un ordre LRU de ces caches fils. Avec ce principe le cache parent essaie de réduire la charge du proxy le plus utilisé, et de faire travailler le proxy le moins sollicité (à condition qu'il possède bien sûr le document demandé).

3.9 Conclusion :

A travers ce chapitre, nous avons essayé de proposer un système de coopération de cache intégré aux routeurs actifs, afin de mettre de l'intelligence dans le traitement des requêtes, et en respectant les contraintes imposées par les routeurs actifs. Ce système est défini par un protocole de communication entre les caches coopératifs. Il va falloir maintenant mettre en œuvre ce protocole et faire des tests pour valider le système et voir sa pertinence opérationnelle.

Chapitre 4
Mise en œuvre et tests

4.1 Introduction :

Après avoir expliqué l'architecture qui permet d'intégrer la fonctionnalité de cache à des équipements réseaux (routeurs actifs), nous allons maintenant aborder la phase de mise en oeuvre du protocole proposé.

Nous avons utilisé la plate forme *TAMANOIR* [20] pour mettre en place notre système de coopération de cache. Chaque noeud actif *TAN* est lancé soit en mode parent, soit en mode fils. Nous avons choisi les nœuds intérieurs qui sont plus proches des utilisateurs comme des caches fils qui jouent le rôle proxies pour l'accès au Web, alors que les caches parents quand à eux sont à un ou plusieurs niveaux plus haut, pour pouvoir regrouper plusieurs nœuds fils.

4.2 Le fonctionnement du système de coopération :

Les navigateurs des utilisateurs Web sont configurés de manière à envoyer leurs requêtes vers un des proxies cache fils de notre système coopératif. Le proxy lance pour chaque requête entrante le service *HttpHandlerS* qui s'occupe de servir cette dernière. Ce service contacte le gestionnaire de cache local pour savoir s'il peut satisfaire la requête localement, si c'est le cas il envoie la réponse directement au client demandeur, autrement il met la requête en attente, et fait appel à la coopération de ces caches frères en envoyant la requête au cache parent (cette requête serra traitée par le service *CacheCooperationS*).

An niveau du cache coordinateur (parent), le service de coopération vérifie dans les tables miroirs qu'il possède l'existence de l'objet demandé chez les caches fils. Le service contacte le cache qui possède cet objet s'il existe, et lui demande de transmettre l'objet au demandeur initial. Dans le cas d'un *faux Hit* chez le fils sollicité, celui-ci déclenche une notification du cache parent pour incrémenter son nombre de *faux Hit* dans la table miroir, ainsi qu'avertir le cache demandeur pour qu'il contacte lui-même le serveur distant pour cet objet.

Dans le cas ou le cache fils possède l'objet demandé, il répond directement à son voisin qui a initié la requête, en lui envoyant l'objet demandé (transfert pair à pair). A la réception de l'objet de l'un des voisins, le service *HttpWaitHandlerS* est déclenché pour servir toutes les requêtes des utilisateurs qui sont en attente. La figure 4.1 résume brièvement le principe de fonctionnement d'un cache fils.

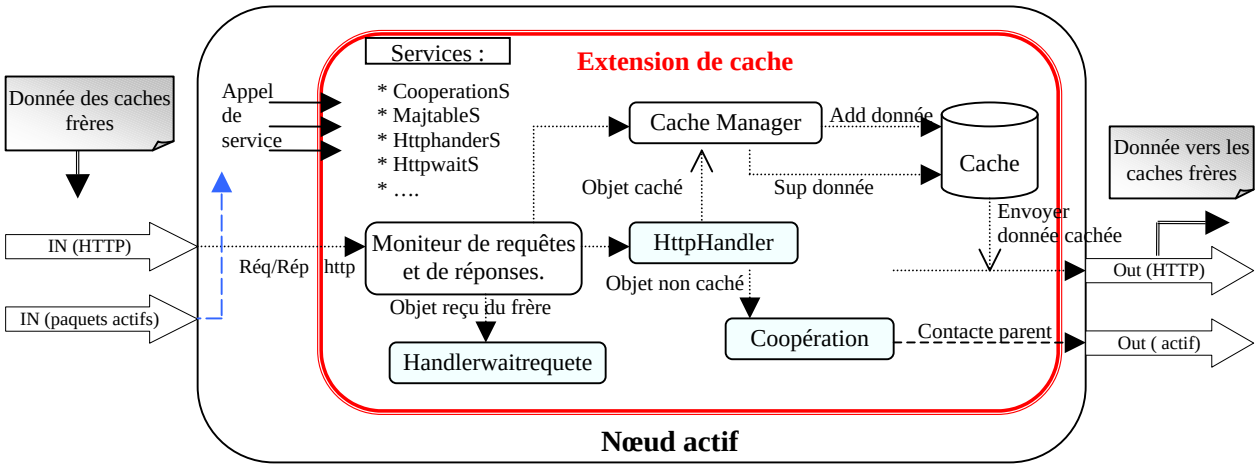

Figure 4.1. Architecture fonctionnelle d'un cache fils

4.3 Tests et résultats :

4.3.1 Architecture de simulation :

Nous avons proposé une architecture de test, qui permet de simuler le modèle proposé, en local (monoposte) ou en multiposte. L'architecture proposée permet aussi de faire des simulations Off line (ie : on à pas besoin d'avoir une connexion à Internet, tout est simulé localement).

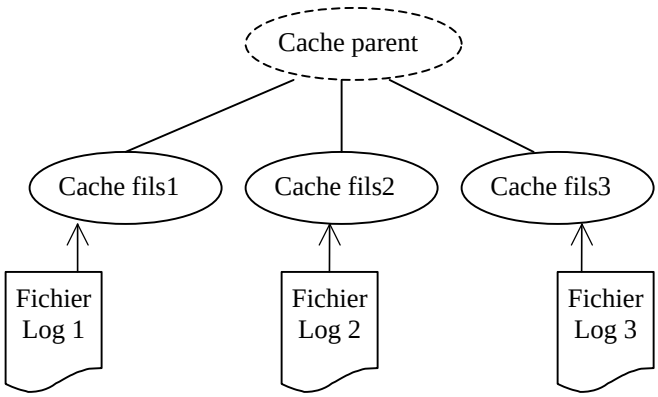

Figure 4.2 Architecture de simulation.

La simulation est basée principalement sur des fichiers Log (récupéré à partir d'un serveur proxy[9]), qui contiennent différents champs. Les champs qui nous intéressent sont :

[9] Ces traces sont disponibles sur le site : http://www.ircache.net

L'utilisateur qui a émis la requête, la méthode http utilisée (GET, HEAD, …), l'URL demandée, et la taille de la réponse envoyée à l'utilisateur. Le format d'une ligne de ce fichier est la suivante :

Time elapsed remotehost code/status bytes method Ur rfc931 peerstatus/peerhost type

Exp:

1035849606.566 394 252.183.145.92 TCP_CLIENT_REFRESH_MISS/200 1160 GET http://br.yimg.com/i/br/cat.gif - DIRECT/200.185.15.91 image/gif

L'architecture de simulation est représentée par la figure 4.2.

Nous avons déployé quatre nœuds TAMANOIR sur la même machine, un d'entre eux joue le rôle de cache coordinateur et les trois autres jouent le rôle des proxies caches fils. On simule les requêtes des utilisateurs d'un proxy cache fils par un fichier Log qui contient les requêtes de ces derniers. On se limite à un ensemble réduit de requêtes (1 000 requêtes par proxy) pour matérialiser le fonctionnement du système. L'ensemble des requêtes d'adressent à 1766 objets différents.

4.3.2 Expérimentations :

Nous essayerons à travers ces expérimentations de valider l'architecture proposée. Vu que ce système prend en compte plusieurs paramètres tels que : la taille du cache, la taille des tables miroirs, les seuils de revalidations des tables miroirs, la détermination des paramètres adéquats reste ouvert à plusieurs tests et expérimentation.

Pour ces tests nous avons choisi la même taille des tables miroirs pour tous les caches. Comme cité dans le paragraphe 3.3.1, la taille a été choisie pour avoir une probabilité de faux hit ≈ 0,024 [9] (4 fonctions de hachage, taille du filtre m=8000 bits ≈ 0,97 kilo octets). Nous avons pris les mêmes valeurs pour les seuils de revalidation α et β (α et β dans toutes les expérimentations)

Définition :
Nous définissons le terme de ***quasi Hit*** *pour désigner le gain de la coopération (ie : le nombre de documents récupérés des caches voisins).*

La formule générale qui relie le Hit, Miss et quasi Hit est la suivante :

Nombre de requêtes = Hit + Miss + quasi Hit

Relation entre taille du cache, seuils de revalidation et la mise à jour des tables :

1 -Caches infinis :

Pour étudier l'effet des seuils de mises à jours des tables miroirs nous avons fait des tests sous condition qu'on possède des caches infinis, cela pour ne mettre en valeur que l'effet des seuils de revalidation (on prend les mêmes seuils β pour les fils et le parent α), sans les méthodes de remplacement de caches.

La figure 4.3 montre la relation entre le seuil de revalidation des tables et le nombre de mise à jour de ces dernières. On remarque que ces deux éléments varient dans un sens inverse. Il est primordial d'avoir moins de mises à jour (moins de trafic pour les données de coopération), mais en contre partie en perd l'intérêt de la coopération.

La figure 4.4 illustre ce phénomène, en montrant la variation du *quasi Hit* en fonction du seuil de revalidation. On constate que dès que ce seuil augmente, le nombre de *quasi Hit* diminue, et la coopération perd de son efficacité.
Sur les 3000 requêtes, sont concentrées sur 1766 documents, le Hit local de chaque cache est constant pour chaque cache : *cache1* = 439 ; *cache2* = 221 ; *cache3* = 339.

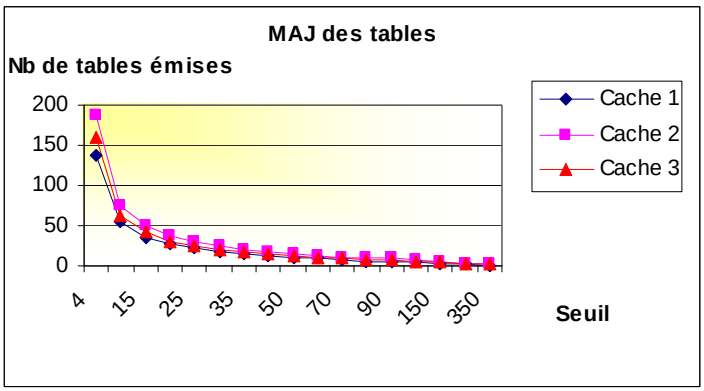

Figure 4.3. Variation des MAJ des tables (cache infini)

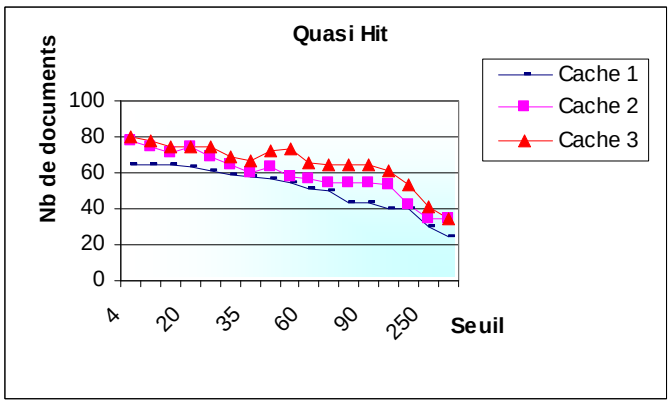

Figure 4.4. Variation du quasi Hit (cache infini)

1 –Caches finis :

Nous avons effectué d'autres tests mais cette fois si, en utilisant des caches de taille finie (10% de la taille du cache infini) afin de voir l'effet des méthodes de remplacement. Pour cela nous avons implémenté une méthode de remplacement simple LRU (paragraphe 1.4). Les résultats sont représentés par les figures 4.5, et 4.6.

Les mêmes constatations que celles d'avant, le nombre de mises à jour et le nombre de quasi hit diminue avec l'augmentation des seuils de revalidation (α, β). Nous avons aussi remarqué que le Hit de cache a diminué à cause de la méthode de remplacement (*cache1* = 355 ; *cache2* = 149 ; *cache3* = 291).

Une autre remarque importante c'est que le *quasi hit* a augmenté par rapport à l'expérience de cache infini. Ceci est du à la mise à jours des tables miroirs après la suppression des éléments du cache. Ces mises à jour sont déclanchées du côté de caches fils après un seuil *β.*

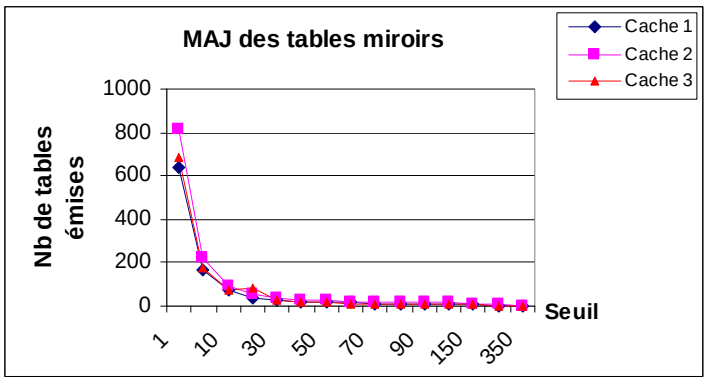

Figure 4.5. Variation des MAJ des tables (cache fini)

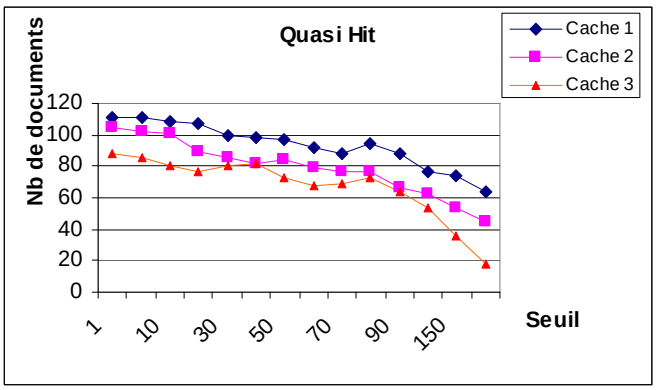

Figure 4.6. Variation du quasi Hit (cache fini)

5. Discussion :

Le but principal de ce système de cache est de permettre une localisation rapide d'un document dans une communauté de caches coopératifs, en évitant que les caches parent soient obligés de garder une copie de ces documents, comme c'est le cas dans le cas des caches hiérarchiques.

Dans cette approche, et contrairement à l'approche hiérarchique, le cache n'a pas besoin d'interroger ses voisins pour déterminer si un objet existe chez eux ou non. On évite les problèmes de gestion liés à la maintenance des informations des caches voisins pour chaque cache fils. Cette tache est attribuée aux caches parents, chaque cache à une vision globale du contenu de ces fils en gardant la trace de chacun d'eux (ie : la table miroir). Cela permet à n'importe quel cache fils de connaître le contenu de ces caches voisins en envoyant une seule requête à son cache parent. Ce système supprime l'obligation d'avoir une grande capacité de stockage au niveau des caches parents, et d'utiliser la hiérarchie pour avoir une localisation rapide des objets.

Le temps de latence dû à un Miss est aussi réduit, par rapport aux caches hiérarchiques. En effet, dans les systèmes hiérarchiques, un cache est obligé d'interroger tous ces caches frères, puis contacter son cache parent lors d'un *Miss*, et finalement contacter le serveur distant si le cache parent n'a pas résolu la requête. Dans notre proposition, le fils contacte seulement son père par le biais d'une requête de coopération, pour avoir la localisation d'un objet. Est-ce qu'il existe chez ses caches frères ou non ? Si ce n'est pas le cas, il s'occupe lui-même de la récupération de ce document du serveur distant. Les caches fils s'occupent de cette tache, pour ne pas trop surcharger leur cache parent.

Un autre avantage de cette solution est : le nombre de messages échangés entre les caches coopératifs limité : seuls quatre messages au maximum sont nécessaires pour récupérer un objet donné.

De plus, l'échange des tables miroirs ne se fait qu'à des moments opportuns, où les caches parents et les caches fils jugent utile le transfert de ces tables. Ce moment est conditionné par des seuils de revalidation, qui sont des paramètres du système. Ces seuils sont des valeurs empiriques, qu'on peut trouver expérimentalement, et que le cache parent ainsi que les caches fils peuvent modifier pour contrôler la fréquence des mises à jour des tables.

Dès que le nombre de requêtes augmente dans le réseau, les caches proxies deviennent responsables de servir plusieurs requêtes en provenance des autres caches voisins, en plus des requêtes de leurs clients. Si ce nombre devient très important, le proxy pourrait être un goulot d'étranglement.

Mais, on peut bien évidemment remédier à cet inconvénient en :

- ✓ Autorisant les proxies caches à travailler dans un contexte « Best effort », ça veut dire que le cache peut prendre la décision de refuser les requêtes en provenance des autres caches voisins, quand il se trouve dans une situation de surcharge, afin de ne pas perturber son bon fonctionnement.
- ✓ Une autre possibilité, c'est de mettre en place des petites caches au niveau des caches coordinateurs (parent), ainsi, ils peuvent retourner les objets qui existent chez eux sans les demander aux caches fils.

La détermination des paramètres adéquats de ce système, reste aussi ouvert à de nombreux testes. En effet, plusieurs paramètres rentrent en jeu dans la mise en place de cette approche, à savoir, la taille des tables miroirs, les seuils de revalidation des tables miroirs, et le nombre de fonctions de hachage utilisées dans le bloom filter.

6. Conclusion et perspectives :

Le travail réalisé pendant ce stage a permis d'intégrer la fonction de proxy caches coopératifs sur des nœuds actifs. Ce système permet de partager et distribuer un ensemble de documents dans une communauté de caches coopératifs, en respectant les contraintes de limitation de capacité de stockage et de traitement des noeuds actifs.

Nous avons franchi un pas supplémentaire vers l'utilisation des réseaux actifs, en développant de nouvelles gammes d'applications pour ces derniers. Celui-ci se matérialise par l'intégration de la fonction de cache aux routeurs actifs. Nous avons présenté une approche originale qui permet d'introduire de l'intelligence dans le traitement des requêtes, et de faire une collaboration efficace de cache. Ce système a comme avantage la *localisation* rapide des objets dans la communauté de caches coopératifs, et la *délivrance* efficace de ces données aux demandeurs de ces objets. Les nœuds actifs jouent à la fois le rôle de routeur actif et le rôle d'un proxy cache qui s'occupe de servir les requêtes des utilisateurs. Nous avons utilisé les *tables miroirs* qui donnent aux caches parents une description (d'où la notion d'image) du contenu de leurs caches fils, pour une localisation rapide des objets, afin de réduire le trafic généré par l'échange d'informations.

La mise en œuvre sous TAMANOIR et les tests effectués ont monté la pertinence fonctionnelle et opérationnelle de l'approche.

Ce travail ouvre la porte sur des nouveaux thèmes de recherches, et des problématiques sous jacentes. En effet, plusieurs sujets restent ouverts, et n'ont pas été abordées dans ce stage. Le système proposé prend en compte plusieurs paramètres (taille du cache, taille du filtre, seuils de mises à jour α et β). Une étude expérimentale plus poussée affinera les paramètres et leurs interdépendances.

De plus en plus de pages Web sont générées dynamiquement notamment par l'utilisation des Web services : il devient plus délicat de cacher ces documents. Ce problème est connu sous le nom de « cache actif » Il serait très intéressant d'ajouter la fonctionnalité de cache des documents actifs sur des routeurs actifs. Une piste de recherche serait de cacher les Web services sur les routeurs actifs et les faire exécuter en locale.

Un autre sujet est comment rendre le déploiement des caches sur les routeurs actifs dynamiques. Cela veut dire que les routeurs peuvent acquérir l'extension de cache (cacher des objets) quand c'est bénéfique (il faut définir des critères de performance), comme ils peuvent abandonner cette extension. Bien évidemment, ceci ce fait par la définition d'un protocole de communication inter routeurs, qui permet l'ajout, la suppression de l'extension cache, ainsi que le choix du moment de déclenchement de ces événements.

Références :

[1] G. Barish and K. Obraczka, "*World Wide Web Caching: Trends and Techniques*," IEEE Communications, May 2000

[2] L. Zhang, S. Floyd, and V. Jacobson. *Adaptive Web Caching*. In Pvceedings of the NLANR Web Cache Workshop, June 1997. 14

[3] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahab, H.Pindi, and A. Nagarajan. *Implementation of a Prototype Active Network*. In Proceeding of the 1st IEEE Conference on Open Architectures and Network Programming (OPENARCH '98), April 1998.

[4] Thijs Lambrecht, Peter Backx, Bart Duysburgh, Liesbeth Peters, Bart Dhoedt, Piet Demeester, "*Adaptive Distributed Caching on an Active Network*", IWAN 2001 short paper, Philadelphia, PA, US, disponible sur : http://allserv.rug.ac.be/~pbackx/

[5] L. Brunie, J. M. Pierson, D. Coquil , "*Semantic collaboratif web cache*", The Third International Conference on Web Information Systems Engineering (WISE'02) December 12 - 14, 2002

[6] M Kaiser, KC Tsui and J Liu, *Adaptive Distributed Caching*, in Proceedings of the IEEE Congress on Evolutionary Computation, pp. 1810-1815, May, 2002.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "*Web caching and zipf-like distributions: Evidence and implications*," in Proceedings of the IEEE Infocorn 1999

[8] A. J. Menezes, P. C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography*: CRC Press (5th printing) August 2001

[9] L. Fan, P. Cao, J. Almeida, A. Z. Broder, *Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol*, IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 8, NO. 3, JUNE 2000

[10] Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall and G.J. Minden, "*A Survey of active network research*," IEEE communications Magazine, pp.80--86, January 1997.

[11] P. Cao and S. Irani, *Cost-Aware WWW Proxy Caching Algorithms*, Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, pp. 193-206, Dec 1997.

[12] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, H. M.Levy, *On the scale and performance of cooperative web proxy caching*. In Symposium on Operating Systems Principles, pages 16–31, 1999.

[13] Duane Wessels, *Squid and ICP: Past, Present, and Future* August 6, 1997

[14] Duane Wessels, k. claffy *ICP and the Squid Web Cache* August 13, 1997

[15] U. Legedza, J. Guttag, *Using Network-level Support to Improve Cache Routing*, Appears in the Proceedings of the 3rd International WWW Caching Workshop, Manchester, England, June 1998

[16] J. Wang, *A survey of Web Caching Schemes for the Internet*, ACM Computer Communication Review, 29(5):36--46, October 1999

[17] JP. Gelas, L. Lefèvre, *Performance et dynamicité dans les réseaux : l'approche Tamanoir*, JDIR 2002, Toulouse, France, 4-6 Mars 2002

[18] JP. Gelas, L. Lefèvre, *TAMANOIR: A High Performance Active Network Framework, Active Middleware Services (AMS)*, Kluwer Academic Publishers, ISBN 0-7923-7973-X, August 2000.

[19] JP. Gelas, L. Lefevre, *Mixing High Performance and Portability for the design of Active Network Framework with Java*, 3rd International Workshop on Java for Parallel and Distributed Computing, International Parallel and Distributed Processing Symposium (IPDPS 2001), San Fransisco, USA, April 2001.

[20] *TAMANOIR project home page*
disponible sur *:* http://www.ens-lyon.fr/~jpgelas/TAMANOIR/index.html

[21] Víctor J. Sosa, Leandro Navarro, *Influence of the Document Validation/Replication Methods on Cooperative Web Proxy Caching Architectures (2002)*

[22] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahhab, H. Pindi and A. Nagarajan, *Implementation of a Prototype Active Network*, disponible sur http://www.ukans.magic.net/KU/MAGIC-II/docs/OpenArch98.ps.gz.

[23] J. S. Gwertzman and M. Seltzer. *The case for geographical push-caching*. In Proceedings of the Workshop on Hot Topics in Operating Systems, pages 51-57, May 1995.

[24] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, T. Jin, *Evaluating Content Management Techniques for Web Proxy Caches*, Internet Systems and Applications Laboratory HP Laboratories Palo Alto HPL-98-173 April, 1999

[25] D Povey, J. Harrison, "*A Distributed Internet Cache* ", 20th Australian Computer Science Conference, Sydney, Australia, February 5-7. 1997,

[26] Mohammad Salimullah Raunak, *A Survey of Cooperative Caching*, Décembre 15, 1999, disponible sur http://citeseer.nj.nec.com/raunak99survey.html

[27] S. Michel, K. Nguyen, A. Rosenstein and L. Zhang, *Adaptive Web Caching: Towards a New Caching Architecture*, In 3rd international Web Caching Conferenced, 1998

[28] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel, *A hierarchical Internet object cache*, Usenix'96, January 1996.

[29] Squid Web Proxy Cache, disponible en ligne sur : http://www.squid-cache.org/

[30] P. Rodriguez, C. Spanner, E.W. Biersack, "*Web Caching Architectures: Hierarchical and Distributed Caching*". 4th International Caching Workshop, 1999.

[31] A. Rousskov and D. Wessels, "*Cache Digest*" Proceedings of the WCW'98, Manchester, England, 1998.

[32] Li Fan, P. Cao, J. Almeida and A. Broder, *A prototype implementation of Summary-Cache Enhanced ICP Implementation*, disponible sur http://www.cs.wisc.edu/~cao/sc-icp.html.