**MASTER EN SCIENCES INFORMATIQUES**

# Optimisation d'un logiciel de transmission de MMS

Marquet, Renaud

*Award date:*
2004

q : identification des messages id.

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR
Institut d'Informatique
Année académique 2003-2004

# Optimisation d'un logiciel de transmission de MMS

Renaud Marquet

Mémoire présenté en vue de l'obtention
du grade de Maître en Informatique.

*A Barbara, qui illumine mes jours...*

**Résumé**

Le *Multimedia Messaging Service* (MMS) peut être considéré comme le futur remplaçant du SMS. Le contenu riche en éléments multimédias d'un MMS implique que le centre nerveux de la technologie du MMS, le *MMS Relay*, doit être soigneusement optimisé afin d'éviter de devoir utiliser des ressources matérielles trop importantes pour assurer la transmission des MMS. Nous allons présenter ici un moyen d'optimiser le *Nextenso MMS Proxy Relay* développé par la société *Nextenso SA* en utilisant un système de cache pour éviter les opérations redondantes sur les messages envoyés à plusieurs destinataires. Pour ce faire, nous allons tout d'abord, après avoir parcouru de nombreuses recherches en la matière, développer une politique de remplacement originale adaptée au contexte du MMS. Ensuite, nous présenterons les spécifications des optimisations qui seront introduites en tenant compte de l'architecture particulière du *Nextenso MMS Proxy Relay*. Nous terminerons alors en discutant quelques détails de l'implémentation.

**Mots-clés :** MMS, *MMS Relay*, cache, *proxylet*

**Abstract**

The *Multimedia Messaging Service* (MMS) can be regarded as a multimedia extension of the well known SMS. Because of the rich data content of the messages, the transmission system, called *MMS Relay*, must be carefully optimized in order to avoid the use of a lot of hardware resources. This paper presents a means of optimizing the *Nextenso MMS Proxy Relay* developed by *Nextenso, SA* using a cache system to avoid redundant operations on messages sent to numerous recipients. First, we will present an in depth review of existing cache technologies. Next, we will introduce an original policy of replacement adapted to the context of the MMS. The specifications of the solution holding account of the particular architecture of the *Nextenso MMS Proxy Relay* are then presented. Finally the details of the implementation are exposed.

**Keywords :** MMS, *MMS Relay*, cache, *proxylet*

# Avant-propos

Avant tout, j'aimerais remercier certaines personnes sans qui la réalisation de ce document ainsi que le bon achèvement du stage qui en est la source n'auraient pas été possible.

Je remercie mon promoteur, Monsieur le professeur Jean Ramaekers pour son expérience et ses précieux conseils quant à la rédaction d'un mémoire. Il a toujours su m'orienter discrètement de manière à maintenir un aspect académique à ce mémoire qui se base pourtant sur des réalisations pratiques en entreprise.

Je tiens ensuite à remercier Monsieur Thierry Godfroid pour l'attention qu'il m'a accordé durant mon stage au sein de l'entreprise *Nextenso SA* ainsi que pour ses relectures très critiques et approfondies de ce document qui m'ont permis d'en améliorer grandement le contenu.

Je remercie également Monsieur Henri Colette pour le temps qu'il a passé à corriger l'orthographe d'une grande partie de ce document.

Merci aussi à tout le département *R&D* de *Nextenso*, et tout particulièrement aux membres de l'équipe MMS, pour l'ambiance chaleureuse, détendue et constructive qui a régné pendant toute la durée de mon stage.

Merci à ma famille pour son soutien discret mais sans lequel la rédaction ce document n'aurait pas été possible.

Je remercie tout spécialement Barbara et lui dédie cet ouvrage.

Il ne me reste plus qu'à souhaiter au lecteur une agréable lecture. J'espère qu'il prendra autant de plaisir à découvrir son contenu que j'en ai eu à le rédiger.

# Table des matières

# Table des figures

# Glossaire

FIFO : *First In First Out.* Type de politique appliqué à une file d'attente qui spécifie que les premiers éléments entrés dans la file sont toujours les premiers à en sortir,

HLR : *Home Location Register.* Base de donnée contenant des informations sur les clients d'un opérateur ainsi que la position géographique de leurs terminaux,

HTTP : *HyperText Transfer Protocol.* Protocole de communication réseau utilisé pour le transport des fichiers hyper-textes à travers l'Internet,

Instance : Dans le cadre de la *Nextenso Proxy Platform*, il s'agit d'une occurrence d'une application exécutée sur une machine,

MIME : *Multipurpose Internet Mail Extension.* Protocole permettant l'envoi de contenu binaire à travers l'Internet (images, sons, vidéos, ...). A l'origine, il a été développé pour augmenter les capacités des e-mails,

MMS : *Multimedia Messaging Service.* Service très similaire à celui des SMS, il permet toutefois d'ajouter du contenu multimédia (images, sons, vidéos, ...) au texte dont la taille n'est par ailleurs plus limitée,

MMS Relay : Organe central des technologies du MMS. C'est lui qui assure la transmission des MMS entre les utilisateurs. Le plus souvent, chaque opérateur possède son propre *MMS Relay*,

Proxy : Type particulier de serveur qui agit comme un intermédiaire entre une application cliente (comme par exemple un navigateur Internet) et un vrai serveur,

SMIL : *Synchronized Media Integration Language.* Langage permettant de spécifier la présentation de plusieurs fichiers média ensemble. Par exemple, synchroniser du son avec une vidéo,

SMS : *Short Message Service.* Fonctionnalité disponible sur certains réseaux de téléphone sans fil qui permet aux utilisateurs d'envoyer et recevoir des messages écrits de 160 caractères,

SOAP : *Simple Object Access Protocol.* Protocole de communication basé sur l'XML utilisé principalement pour l'échange de messages avec les *web services*,

Terminal : Dans le cadre de ce document, tout appareil capable de recevoir et envoyer des MMS. Il s'agit le plus souvent d'un téléphone portable,

UAProf : Document de type XML contenant toutes les caractéristiques d'un téléphone mobile. Il permet aux opérateurs de connaître précisément les capacités techniques de tous les téléphones,

URL : *Uniform Resource Locator.* Standard permettant de spécifier l'emplacement d'une ressource sur l'Internet,

VAS : *Value-Added Service.* Services extérieurs à l'opérateur proposés aux utilisateurs des MMS. Il s'agit de n'importe quel type de service dont le MMS peut servir de support,

WAP : *Wireless Application Protocol.* Protocole de communication réseau utilisé principalement pour accéder à l'Internet avec des appareils sans fil comme, par exemple, les téléphones mobiles,

XML : *eXtensible Markup Language*. Standard d'échange de données permettant, de manière très flexible, de définir le format exact des données à échanger,

# Introduction

Le *Multimedia Messaging Service* (MMS) constitue la nouvelle génération de messagerie mobile instantanée. Il permet à différents utilisateurs de téléphones portables de s'échanger des messages contenants aussi bien du texte que du son, des images ou de la vidéo. Cette technologie spécifie également de nombreux ponts vers d'autres systèmes de communication tels que les e-mails ou les téléphones fixes. On suppose que le succès du MMS sera aussi important que celui du SMS, qu'il est d'ailleurs, à terme, appelé à remplacer.

Au centre de cette technologie se situe un logiciel particulier : le *MMS Relay*. C'est lui qui assure la transmission des messages entre les utilisateurs. Le riche contenu des MMS qui peuvent peser plusieurs dizaines de kilo-octets impose une charge de travail très importante à cet organe central. Pour pouvoir assurer le service, il est nécessaire d'utiliser des solutions matérielles performantes, le plus souvent de manière distribuée.

Afin de minimiser les efforts à fournir, en termes de matériel, est apparue la nécessité d'optimiser le *MMS Relay*.

Dans le cadre d'un stage effectué auprès de la société *Nextenso SA*, nous avons été amené à proposer des optimisations pour le *Nextenso MMS Proxy Relay* qui est un *MMS Relay* développé au sein de cette société.

Après nous être familiarisé avec toutes les technologies associées au développement d'un tel produit, nous avons effectué différentes mesures des performances du logiciel selon la taille des MMS employés pour la réalisation de ces tests. Nous avons ainsi pu mettre en évidence des zones critiques sur lesquelles il semblait le plus profitable d'appliquer des optimisations ainsi que des cas d'utilisation particuliers qui pouvaient nous orienter dans la recherche de ces optimisations. Au vu de ces considérations, nous avons décidé de développer une mémoire cache particulière pour permettre d'optimiser les traitements du *Nextenso MMS Proxy Relay*. Ceci fait, avec la présentation des différentes technologies associées au MMS, l'objet de la première partie de ce document.

Dans une deuxième partie, nous nous attachons à présenter en détail les différentes notions à prendre en compte pour le développement d'une mémoire cache. Nous avons parcouru la littérature scientifique dans ce domaine et tentons d'en résumer l'essentiel du contenu. Tous ces concepts sont introduits d'une manière qui se veut pédagogique afin d'aider le lecteur à bien comprendre les applications qui en sont faites.

La troisième partie décrit la solution finale qui a été implémentée pour réaliser les optimisations. Pour commencer, nous mettons en relief les différents aspects du design de la cache en suivant les méthodes présentées dans la deuxième partie. Cette approche théorique est confrontée aux contraintes spécifiques à l'architecture utilisée par le *Nextenso MMS Proxy Relay*.

Après avoir présenté les spécifications techniques du projet, nous discutons les points clés de l'implémentation ainsi que les solutions qui ont été trouvées pour rendre celle-ci aussi efficace que possible.

# Première partie

# La problématique

# Chapitre 1

# Le MMS

Le MMS, acronyme de *Multimedia Messaging Service*, est une extension conceptuelle[1] du SMS. A l'instar de ce dernier, c'est un système qui permet à deux utilisateurs de téléphone portable de s'échanger des messages écrits. Cependant, le MMS[2] n'est plus limité à 160 caractères et permet, en outre, l'inclusion d'un contenu multimédia (images, vidéos, sons, ...).

De plus, le MMS n'est plus uniquement réservé aux utilisateurs de téléphone portable. En effet, il fait interagir différents types de réseaux (voir figure 1.1) : avec ou sans fil, internet, ... et utilise différents protocoles de communication empruntés à ces différents réseaux.

Actuellement, le marché du MMS commence tout juste à se développer mais l'on peut parier qu'il remplacera, dans un futur plus ou moins proche, l'utilisation du traditionnel SMS.

Afin de ne pas engluer le lecteur dans un amas de spécifications techniques trop poussées qui l'éloignerait du but de ce chapitre, nous avons délibérément choisi d'aborder la présentation des technologies du MMS de manière conceptuelle et concise. Toutefois, il sera nécessaire, par moment et pour bien comprendre la problématique qui sera exposée au chapitre 3, d'expliciter certains passages de manière plus technique. Néanmoins, dans la mesure du possible, ils ont été rassemblés dans la dernière section du présent chapitre.

## 1.1 Format d'un MMS

Les messages MMS proprement dit sont en fait basés sur la technologie e-mail. Un message MMS est, le plus souvent, un message de type MIME possédant des champs supplémentaires qui permettent de stocker les propriétés propres à l'univers du MMS.

Le corps d'un MMS est composé de plusieurs parties[3]. Chacune de ces parties contient un élément multimédia du MMS (un texte, une image, un son, ...). Généralement, une des parties

[1] Bien que, *techniquement*, cela ne soit pas le cas.
[2] MMS désigne également le message échangé.
[3] Il s'agit en fait du format *MIME/Multipart*, le lecteur pourra se reporter à [10] pour plus d'informations techniques.

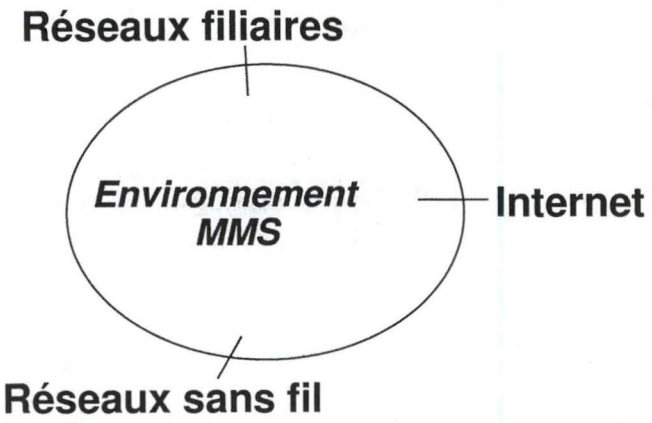



FIG. 1.1 – Environnement du MMS

(inspiré de [1, page 11])

contient un petit document SMIL décrivant la manière dont les différentes parties du message doivent être affichées à l'écran. C'est au terminal[4] qu'il revient d'utiliser ce fichier pour afficher un message aussi fidèlement que possible en fonction de ses capacités techniques.

Ce format peut éventuellement être encodé par le protocole de transport afin, par exemple, d'en limiter la taille. Cette procédure est, par exemple, utilisée sur les réseaux de téléphonie sans fil où la bande passante est très précieuse. L'encodage WSP spécifié dans le WAP-209, par exemple, est spécialement prévu pour le transport du format MMS de manière compressée sur les réseaux de téléphonie sans fil.

## 1.2 Cas d'utilisation principal

L'envoi et la réception de MMS se basent sur un système asynchrone[5] où le destinataire peut choisir, ou non, de récupérer un message qui lui est destiné. Cette mesure est nécessaire car le destinataire d'un MMS peut être facturé pour pour le téléchargement d'un message reçu (En effet, la taille de celui-ci peut être importante et la bande passante dans les réseaux de téléphonie sans fil est coûteuse pour l'opérateur).

Pour mieux comprendre, examinons un cas typique d'utilisation (représenté à la figure 1.2) : l'envoi d'un MMS entre deux utilisateurs de téléphones portables. Quand l'utilisateur $A$ envoie un MMS à destination de l'utilisateur $B$, celui-ci est tout d'abord stocké chez l'opérateur[6]. Une confirmation d'envoi est ensuite envoyée à $A$. $B$ va alors recevoir une notification[7] de réception

[4] Le terme *terminal* désigne tout appareil capable de recevoir et envoyer des MMS.
[5] La confirmation de l'envoi d'un message arrive avant que ce dernier ne parviennent à son destinataire.
[6] Après diverses vérifications comme, par exemple, la validité des destinataires, la facturation, ...
[7] Notons que le destinataire n'est pas facturé pour cette notification.

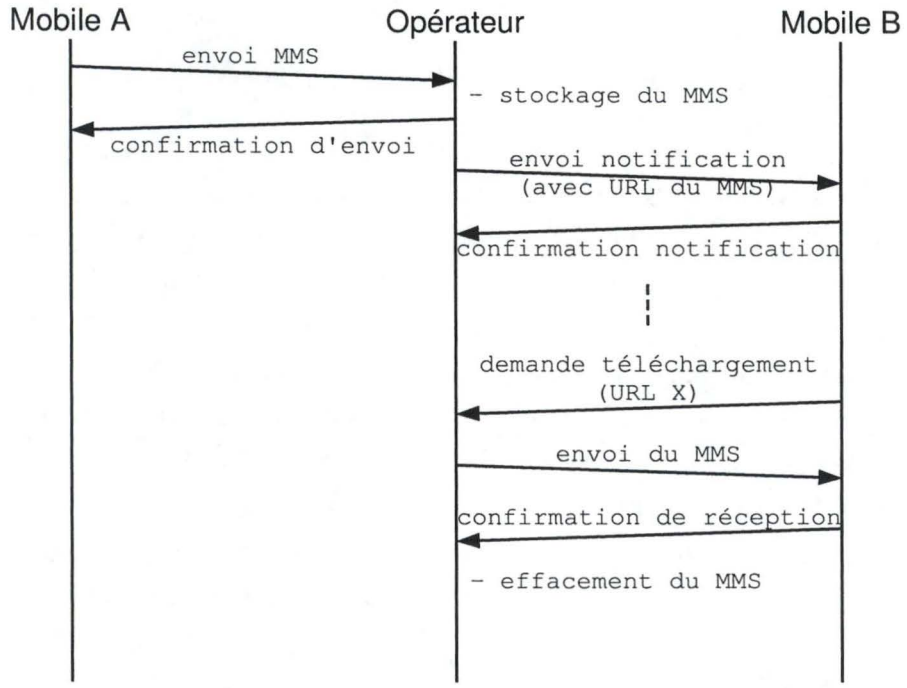



FIG. 1.2 – Cas d'utilisation typique du MMS

de ce message. Il peut alors décider de télécharger ce message sur son téléphone portable pour pouvoir le lire. S'il ne le fait pas, le message finira, au bout d'un temps, par être effacé du serveur de l'opérateur.

Notons qu'une connection internet[8] est nécessaire pour pouvoir télécharger le message. Remarquons d'ailleurs que le message stocké sur un serveur de l'opérateur est accessible au moyen d'une URL.

La notification, quant à elle, peut être reçue via cette connection (pour cela, il faut que le téléphone soit connecté au moment où la notification doit être envoyée) ou bien par SMS. Quelle que soit la méthode choisie, elle est totalement transparente pour l'utilisateur final (les détails sont réglés par le logiciel du téléphone).

Il est également possible de demander l'envoi d'un accusé de réception certifiant que le destinataire a bien téléchargé le message ainsi qu'un accusé de lecture précisant quand celui-ci a effectivement lu le message.

[8] Actuellement, la plus répandue est le WAP, mais le GPRS ou l'UMTS conviennent parfaitement.

### 1.2.1 Autres cas d'utilisation

De nombreux autres cas d'utilisations existent dont certains joueront un rôle clé dans l'exposition de la problématique (voir chapitre 3).

**Téléphone mobile vers un *Value Added Service* (VAS)** Un exemple courant de ce cas d'utilisation est celui d'un utilisateur désirant participer à un concours. Il envoie donc un MMS concernant les informations nécessaires au service adéquat (via un numéro court, par exemple : 1234).

**VAS vers un téléphone mobile** le VAS envoi un MMS (par exemple un bulletin météo) à un grand nombre de destinataires. Il peut utiliser la possibilité d'un MMS de posséder plusieurs destinataires[9]. Cette remarque est particulièrement intéressante, nous y reviendrons plus en détails dans le chapitre 3.

**Téléphone mobile vers e-mail** Ce cas d'utilisation est, tout comme son complémentaire, assez évident. On notera toutefois que, de par le format utilisé par un MMS, les conversions à effectuer seront minimes.

**Transfert (*forwarding*)** Un utilisateur ayant reçu un message peut décider de le transférer à un autre utilisateur ou bien sur un autre serveur de l'opérateur afin qu'il ne soit pas effacé après un certain laps de temps.

...

## 1.3 Adaptation de contenu

En examinant les cas d'utilisations du paragraphe précédent, il est une question qui saute immédiatement à l'esprit. Chaque utilisateur du MMS est issu de technologies non équivalentes : les téléphones portables 3G[10] et les e-mail ont, par exemple, de plus grandes possibilités qu'un téléphone fixe ou un portable de seconde génération. Comment, dès lors, peut-on assurer le transfert des MMS entre ces différents utilisateurs ?

A cette fin, l'opérateur est autorisé à effectuer certaines opérations sur le contenu des MMS. Cette procédure, effectuée au moment du téléchargement d'un MMS par un destinataire, consiste à altérer le format du message en modifiant le moins possible son contenu informatif. Il peut s'agir, par exemple, de redimensionner une image au format du terminal cible ; ou encore de changer la profondeur des couleurs, transformer une vidéo en diapositives, changer le format d'un son, ...

Les capacités d'affichage du terminal cible ne sont pas les seuls paramètres à prendre en compte. En effet, la taille du message est aussi très importante car la mémoire de certains appareils est beaucoup plus limitée que d'autres.

[9] Tout comme l'e-mail, le MMS possède les champs To, Cc et Bcc qui permettent de stipuler plusieurs destinataires.

[10] Les téléphones sont classés par génération : 1G pour les téléphones fixes, 2G pour les téléphones portables et 3G pour les téléphones portables dernière génération, type UMTS.

Pour permettre de connaître précisément les caractéristiques techniques d'un terminal, chaque constructeur met à disposition, au moyen d'une URL publique, un résumé de celles-ci dans un format standardisé basé sur l'XML et appelé *UAProf* (ou profil) du terminal. Chaque terminal enverra cette URL lors de *toute* communication avec le réseau d'un opérateur.

Cependant, même en disposant de toutes ces informations, il est difficile de décider quelles opérations effectuer et dans quel ordre afin d'obtenir le résultat désiré (point de vue résolution des images, formats de fichiers supportés, taille totale, ...). Pour faciliter l'implémentation, des procédures standards sont spécifiées dans [3]. Les terminaux y sont classés par groupe sur lesquels on peut appliquer des transformations pour le transfert de MMS d'un groupe à l'autre.

Il existe également un cas extrême : l'envoi de MMS vers un terminal qui n'accepte pas du tout les MMS. Dans ce cas, un SMS est envoyé au destinataire l'invitant à se connecter à une adresse Internet particulière afin de consulter son message.

## 1.4 Architecture

Sur la figure 1.3, nous pouvons voir une description plus précise de l'environnement du MMS[11].

Au centre de cet environnement se situe le *MMS Relay*. Il est l'organe vital de la transmission des MMS. C'est à lui de faire le lien entre les différents acteurs.

Le *MMS Relay* est en réalité composé de deux parties : le *Relay* et le *Server*. La partie *Relay* s'occupe du décodage des MMS[12], de toutes les vérifications d'usage (source, destination, crédit, ...), éventuellement en faisant appel à des éléments extérieurs, ainsi que du routage des messages. Le *Server*, quant à lui, s'occupe du stockage des MMS. C'est lui qui fournit les URL qui permettent d'accéder aux messages. Par abus de langage, on qualifiera l'ensemble de *MMS Relay*.

Généralement, chaque opérateur possède son propre *MMS Relay* qui lui permet de traiter les demandes de ses clients. Lorsqu'un client (source ou destinataire) se trouve chez un autre opérateur, les deux *MMS Relay* peuvent communiquer entre eux au moyen d'un protocole spécifique.

Pour permettre aux différents acteurs et composants de l'environnement d'interagir même s'ils sont issus de fournisseurs différents, huit interfaces, utilisant chacune leur propre protocole[13] ont été définies. Ceci permet d'avoir une totale indépendance entre ces différents éléments :

**MM1** Cette interface permet à un terminal d'envoyer ou de récupérer un MMS et au *MMS Relay* de délivrer les notifications. Notons également que c'est par ce biais que sont envoyés les accusés de réception et de lecture. Actuellement, l'implémentation la plus répandue de cette interface se fait via la technologie WAP.

**MM2** Celle-ci spécifie comment les partie *Relay* et *Server* interagissent entre elles pour stocker

[11] présenté à la figure 1.1 page 20.

[12] Rappelons que le MMS peut être encodé de différentes façons selon le protocole de transport utilisé.

[13] Ces protocoles ne seront pas présentés en détail ici, nous invitons le lecteur à consulter [1] et ses références pour plus d'information.

et récupérer des MMS.

**MM3** Par cette interface, on peut connecter d'autres types de messageries. La plus évidente est l'e-mail mais il en existe bien d'autres : fax, voice messaging, ...

**MM4** C'est par ce biais que deux *MMS Relay* différents (par exemple chez deux opérateurs différents) peuvent communiquer. Il s'agit en fait de l'utilisation du protocole SMTP. Le choix de ce protocole est assez intuitif au vu du format employé par les MMS.

**MM5** C'est le lien vers le *Home Location Register* (HLR) qui permet d'obtenir les informations d'identification d'un utilisateur.

**MM6** Cette interface fait le lien avec la base de donnée utilisateurs de l'opérateur chez lequel est installé le MMS Relay. Cette base lui permet de connaître la validité de la source d'un MMS (dans le cas d'un MMS entré par l'interface MM1) ou bien si le destinataire d'un message reçu via MM4 se trouve effectivement bien sous son autorité.

**MM7** C'est par cette interface que les MMS sont échangés entre les utilisateurs et les applications VAS. Ces applications fournissent, par exemple, un certain service ou contenu à ses abonnés. Un exemple typique de ce type de service dans le cadre du MMS est la gestion d'un album de photos personnalisé, consultable sur l'Internet mais qui est aussi provisionnable via MMS. Sans entrer dans le détail de ce protocole, il est nécessaire de connaître deux actions spécifiques à celui-ci : la possibilité d'annuler un message précédemment envoyé pour tous les utilisateurs qui n'auraient pas encore reçu de notification de réception (*MM7-Cancel*) ainsi que celle de modifier le contenu d'un message précédemment envoyé pour ~~peu qu'~~*aucun* destinataire n'ait déjà téléchargé ce message (*MM7-Replace*). Ces deux actions spécifiques vont, comme nous le verrons dans la partie 3, nous poser pas mal de problèmes.

**MM8** C'est l'interface du système de facturation. Tous les messages qui passent par le *MMS Relay* génèrent des informations de facturation qui sont ensuite traitées par le système de facturation en fonction des préférences établies par l'opérateur.

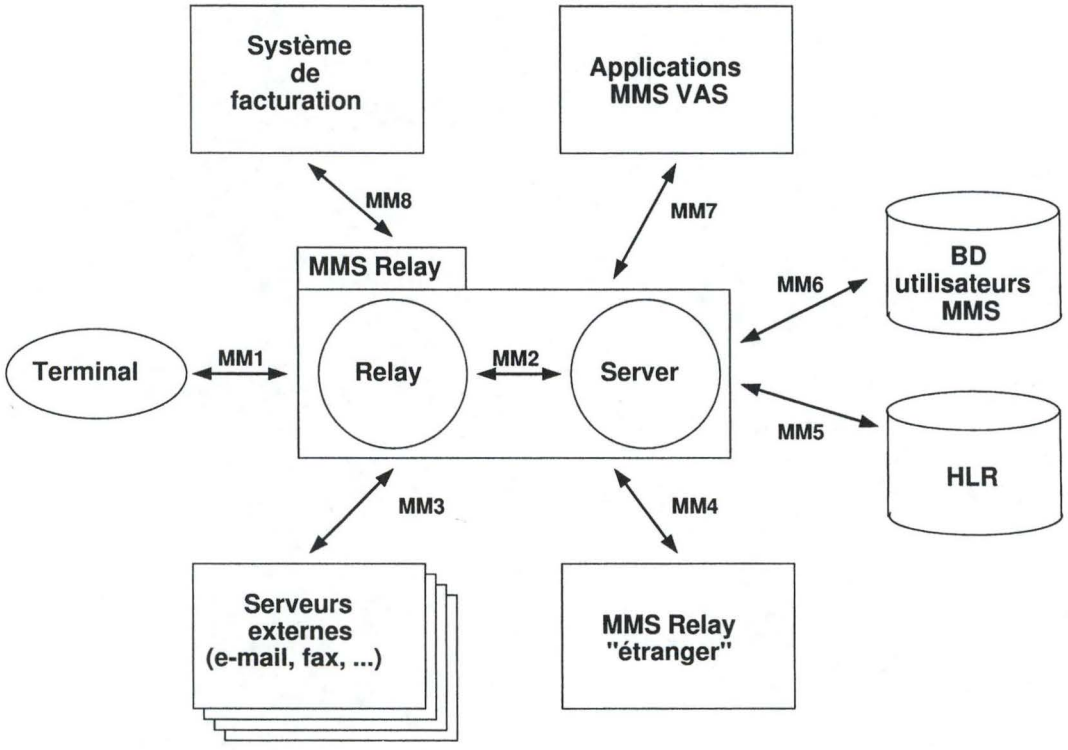



FIG. 1.3 – Architecture du MMS

(inspiré de [1, page 17])

# Chapitre 2

# La Proxy Platform

Ce chapitre sera consacré à la présentation de la *Nextenso Proxy Platform*, qui est la plate-forme de développement sur laquelle les travaux d'optimisation exposés dans ce document ont été réalisés.

Elle sera d'abord présentée dans un cadre général, puis dans le cas particulier d'une solution logicielle de traitement du trafic chez un opérateur de téléphonie mobile. Nous nous attacherons ensuite à décrire de manière plus précise la partie qui fera l'objet des optimisations proposées : le *MMS Proxy Relay*.

Ce chapitre, qui définit le cadre de travail, est essentiel à la compréhension des solutions qui seront retenues pour la réalisation des optimisations.

## 2.1 Architecture générale

La *Nextenso Proxy Platform* est une plate-forme de développement facilitant la réalisation de solutions logicielles orientées réseau. Elle s'intègre dans un flux réseau quelconque à la manière d'un proxy et permet d'agir sur celui-ci.

Elle est *scalable*[1] et permet donc d'écrire très facilement des logiciels respectant ce critère de qualité de plus en plus important. A cette fin, chaque application[2] écrite pour la *Proxy Platform* peut être instanciée sur une ou plusieurs machines. Chaque instance peut alors être lancée ou arrêtée à loisir et augmenter ainsi la capacité de traitement globale.

De nombreuses autres caractéristiques font de la *Proxy Platform* un support de développement réseau intéressant (partage automatique de la charge entre toutes les instances sur les différents hôtes, interface *web* d'administration, ...) mais elles sont malheureusement au delà de la portée de ce document. Nous nous bornerons à expliciter celles qui sont essentielles à la compréhension de

[1] N'ayant pu trouver une traduction française satisfaisante, nous avons préféré utiliser le terme anglais et prions le lecteur de bien vouloir nous en excuser.

[2] Par application, nous entendons un composant de la solution logicielle développée.

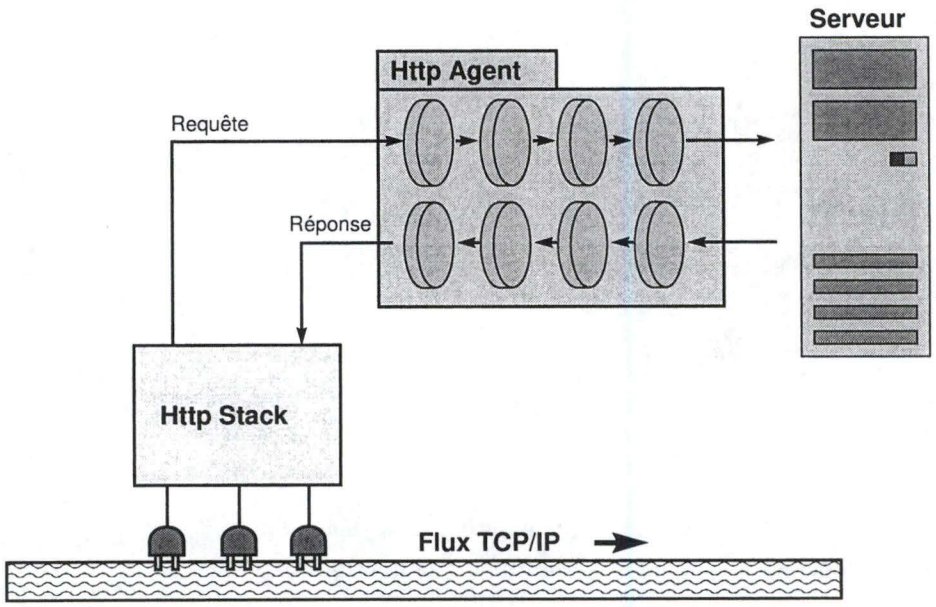



FIG. 2.1 – Illustration d'un couple *Stack* + *Agent*

la problématique.

La *Proxy Platform* abrite deux types principaux de composants : les *Stacks* et les *Callout Agents* (ou *Agents*). Différentes types de *Stacks* sont implémentées pour différents types de protocoles : *HTTP Stack*, *WAP Stack*, ... Elles permettent de gérer les connections de manière performante (à cette fin, elles sont écrites en *C*).

Le type d'application le plus intéressant est sans doute le *Callout Agent*. C'est en fait une application qui contient un ensemble de mini applications écrites en Java : les *Proxylets*. Elles permettent de traiter le flux à un haut niveau et peuvent être assemblées en chaînes de traitement afin de réaliser des opérations très complexes en découpant logiquement le problème à résoudre en opérations élémentaires.

Ces deux types de composants sont toujours utilisés en couple afin de traiter un protocole particulier (une illustration d'un couple pour le protocole HTTP est présentée à la figure 2.1). La *Stack* s'occupe de la gestion bas niveau du protocole et des connections réseau. Elle transmet à l'*Agent* les requêtes associées à ce protocole afin qu'elles puissent être traitées. Comme nous pouvons le voir sur la figure 2.2 (les symboles ressemblant à deux cylindres imbriqués représentent un couple *Stack* + *Agent*), ces couples peuvent ensuite être chaînés afin de faire interagir plusieurs protocoles. Par exemple, dans un couple dédié au protocole WAP, il suffira de générer une requête vers un couple dédié au protocole HTTP (plus exactement vers la *Stack* de celui-ci).

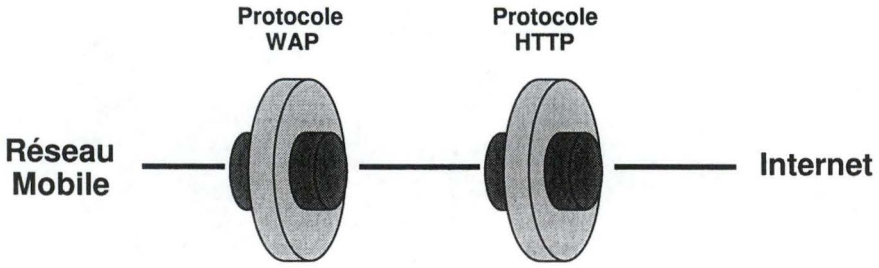



FIG. 2.2 – Illustration du chaînage de couples *Stack* + *Agent*

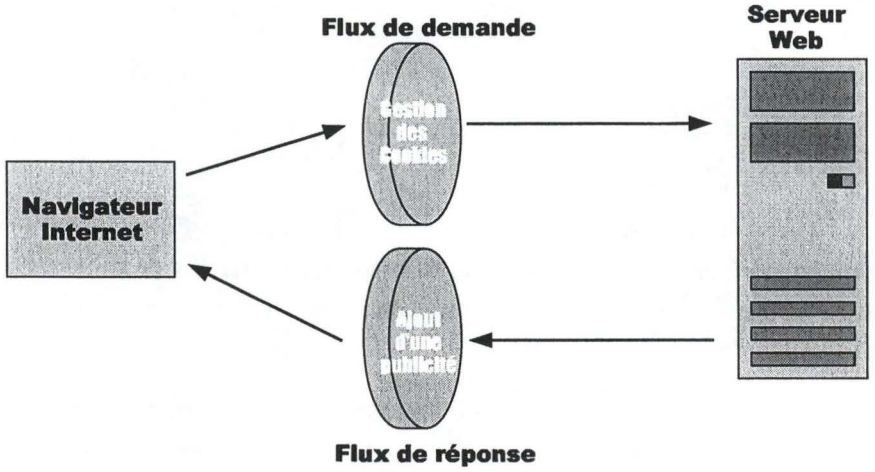



FIG. 2.3 – Exemple d'utilisation des *proxylets*

### 2.1.1 Les *proxylets* HTTP

Le concept de *proxylet* est un concept original développé par la société *Nextenso S.A.* qui donne toute sa force à la *Proxy Platform* en la rendant extrêmement modulable tout en facilitant grandement les phases d'implémentation et de maintenance. Dans cette section, nous allons traiter des *proxylets* fonctionnant sur le protocole HTTP.

Une *proxylet* est un morceau de code écrit en Java permettant de modifier le flux de données entre le client et le serveur. Chaque *proxylet* va réaliser une petite partie de la solution à implémenter. Le résultat final sera obtenu en chaînant les *proxylets* l'une derrière l'autre.

Comme nous pouvons le voir sur la figure 2.3, les *proxylets* sont chaînées sur deux flux de données : le flux de demande et le flux de réponse. Cette séparation permet une découpe plus logique du problème et une modularité plus importante. La figure 2.4 illustre cette modularité : il a été aisé d'introduire une *proxylet* pour traduire le contenu d'une page web demandée dans une autre langue.

Remarquons que cette chaîne est dynamique : on peut appliquer un contrôle sur le flux. En

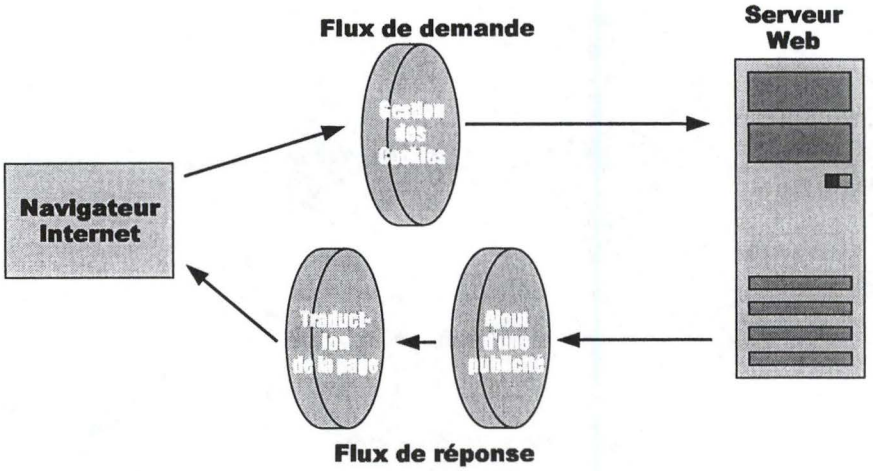



FIG. 2.4 – Exemple d'utilisation des *proxylets*

effet, durant l'exécution, chaque *proxylet* a le choix entre transmettre le flux à la *proxylet* suivante, à la première *proxylet* du flux de réponse (et ainsi court-circuiter le flux de demande) ou bien de court-circuiter la totalité de la chaîne. De même, chaque *proxylet* peut décider de traiter ou non le flux.

Les chaînes de *proxylets* sont rassemblées dans des contextes (attention, chaque contexte ne peut contenir qu'une seule instance de chaque *proxylet* par flux - demande et réponse) qui peuvent ensuite être assignés à une instance de *Callout Agent*.

Notons que les *proxylets* sont multi-tâches et que la gestion des *threads*[3] est automatiquement gérée par le *Callout Agent*. Selon les actions que la *proxylet* doit effectuer sur le flux de donnée[4], elle pourra être exécutée par un *thread* pré-existant ou bien par un nouveau, créé spécialement afin de ne pas pénaliser le reste du flux. Ce comportement est dynamique et peut être adapté à chaque requête.

Il existe deux types de *proxylets* : les *streamed proxylet* et les *buffered proxylet*. Le premier type, plus performant, permet de traiter les données d'une même requête[5] au fur et à mesure de leur disponibilité (n'oublions pas que nous sommes dans un environnement réseau). Cependant, pour certaines opérations, ce type de comportement ne sera pas possible et il faudra alors se tourner vers le deuxième type de *proxylet*. Pour celui-ci, le *Callout Agent* attendra d'avoir reçu la totalité des données d'une requête pour commencer son traitement par la *proxylet*.

Cette description des *proxylets* est assez sommaire mais permettra néanmoins de saisir les problèmes rencontrés lors des optimisations apportées au logiciel s'appuyant sur cette architecture et qui seront mis en évidence au chapitre 3.

[3] Encore une fois nous prions le lecteur de bien vouloir nous excuser pour l'utilisation de ce terme anglais.
[4] En fait, selon que celle-ci risquent de bloquer ou non le flux.
[5] Par requête, nous entendons un ensemble cohérent de données (par exemple, un MMS).

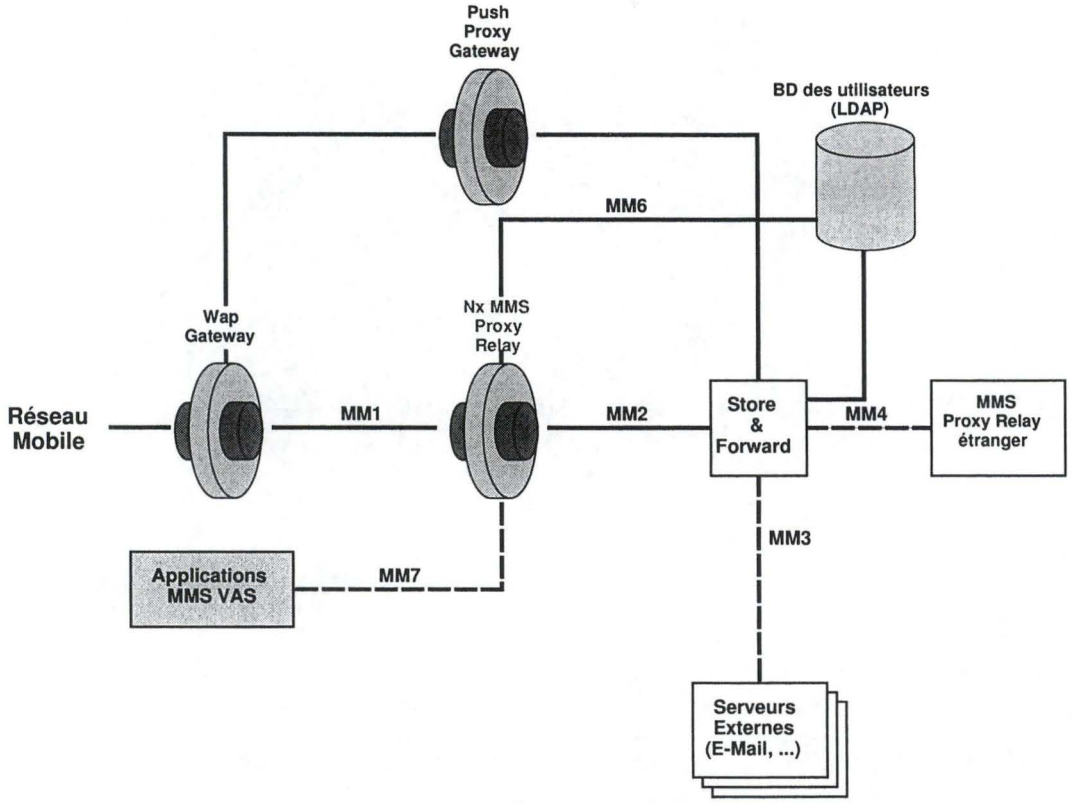



FIG. 2.5 – Architecture de la Proxy Platform

Les lignes en pointillés représentent les connections qui se font typiquement à travers l'Internet (inspiré de [18, page 8])

## 2.2 Application à la téléphonie mobile

L'environnement de travail sur lequel se base ce document concerne l'application de la *Proxy Platform* à une solution logicielle de traitement de MMS pour opérateur de téléphonie mobile.

Notons que dans la suite de ce document, afin d'éviter des lourdeurs de notation, toute référence à la *Proxy Platform* devra être comprise, par abus de notation, comme l'implémentation de cette solution logicielle se basant sur la *Proxy Platform*, et non le concept générique de *Proxy Platform*

Nous n'entrerons pas ici dans les détails de l'architecture, les optimisations présentées concernent uniquement le *MMS Proxy Relay*. Dans ce cadre, il est tout de même important de pouvoir le situer dans son contexte d'utilisation sur la *Proxy Platform*.

Comme expliqué plus haut, chaque *Stack* transmet les requêtes[6] à un *Agent* de type correspondant (p.ex. *HTTP Agent* pour une *HTTP Stack*) qui est implémenté au moyen de *buffered proxylets*.

[6]Requête doit ici être compris dans le cadre d'un protocole de communication.

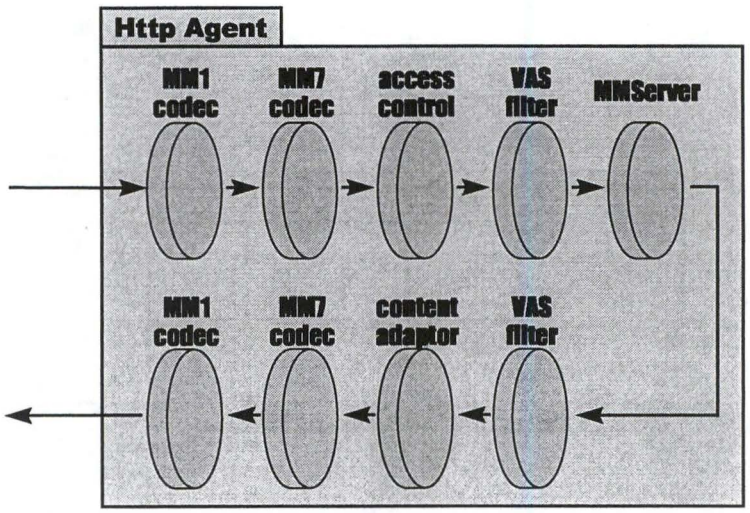



FIG. 2.6 – Schéma des proxylets de l'*HTTP Agent* du *Nextenso MMS Proxy Relay*

C'est à nouveau la *Stack* qui se chargera de la bonne transmission du résultat. Une illustration de ce principe est présentée à la figure 2.1.

Sur la figure 2.5 se trouve l'architecture générale dans laquelle se situe le *MMS Proxy Relay*. Attention, il peut exister plusieurs instances des couples *Stack* + *Agent* sur une ou plusieurs machines mais dans un souci de clarté, le schéma ne tient pas compte d'un déploiement complexe. Ceci n'est pas important pour l'instant même s'il faudra en tenir compte lors de l'implémentation des optimisations choisies. Ceci sera exposé en détail dans la troisième partie du présent document.

## 2.3 Le Nextenso MMS Proxy Relay

Le *Nextenso MMS Proxy Relay* est la partie de la *Proxy Platform* qui réalise toutes les opérations nécessaires à l'acheminement et au stockage des MMS.

Le stockage proprement dit, qu'il soit temporaire ou permanent, est en fait délocalisé dans le module *Store & Forward*[7] (voir figure 2.5). C'est d'ailleurs ce dernier qui se chargera d'envoyer les notifications via, par exemple, le *Push Proxy Gateway*.

Dans ce document, nous nous intéresserons à l'*HTTP Agent* qui forme le coeur du *Nextenso MMS Proxy Relay* et dont on peut voir une représentation sur la figure 2.6. Remarquons que dans cet *Agent*, le flux ne continue pas jusqu'à un serveur mais est entièrement traité par des *proxylets*.

Etudions plus en détail les différentes *proxylets* présentées sur cette figure :

**MM1 Codec** Cette *proxylet* est responsable de l'encodage/décodage des trames de type *WAP-209* qui proviennent ou sont à destination du canal *MM1*.

[7] Ce composant correspond en fait à l'entité *Server* présentée dans le chapitre 1.

**MM7 Codec** A l'instar du *MM1 codec*, celui-ci est responsable de l'encodage et du décodage des trames *SOAP* qui transitent par le canal *MM7*.

**access control** Ici, les informations d'identification de l'utilisateur à l'origine de la trame sont vérifiées par consultation de la base de donnée accessible par le canal *MM6*. Pour pouvoir poster un nouveau message, il faut évidemment que l'utilisateur soit enregistré et actif dans cette base de donnée. Notons tout de même que les trames issues du canal *MM7* ne font pas l'objet d'une vérification dans cette *proxylet* ; elle aura lieu dans le *VAS Filter*

**VAS filter** Cette *proxylet* a deux fonctions : d'une part, elle effectue le contrôle d'accès sur les trames originaires du canal *MM7*[8] et d'autre part, elle dialogue avec les VAS.

**content adaptor** Comme nous l'avons vu au chapitre 1, il est nécessaire d'adapter le contenu des messages qui sont récupérés par un téléphone portable (via le canal *MM1*) afin que celui-ci puisse l'afficher correctement. C'est à cette *proxylet* de s'acquiter de cette tâche.

**MMServer** Cette *proxylet* assure le lien entre le module *Store & Forward* et l'*Agent*. Elle permet, via ce dernier, de stocker, transférer ou récupérer un message.

Le lecteur trouvera dans l'annexe A une illustration, dans le cadre de la *Proxy Platform*, du cas d'utilisation principal présenté au chapitre 1.

Après cette description des différents éléments de la *Proxy Platform*, nous pouvons enfin définir la problématique proprement dite, qui sera développée au chapitre suivant grâce aux notions présentées jusqu'à maintenant.

[8] Nous sommes donc dans le cas d'utilisation d'un VAS postant un message vers un utilisateur final.

# Chapitre 3

# La problématique

La plupart des opérateurs demandent des systèmes capables de traiter, pour le MMS, le même nombre de messages par seconde que pour le SMS. Mais la taille d'un MMS est sans commune mesure avec celle d'un SMS. A titre d'exemple, avec des MMS de 100kb (les derniers terminaux sont capables de traiter des MMS de plus de 300kb), un *MMS Relay* doit, pour assurer le routage de 1000 MMS/seconde, avoir une capacité de traitement de pratiquement 100Mb/seconde.

La capacité des machines actuelles ne permet de réaliser de telles performances qu'en utilisant de nombreuses machines formant un ensemble difficile à administrer. Pour éviter de devoir recourir à un nombre trop important de machines, il est nécessaire d'optimiser au maximum le *MMS Relay*.

Dans cette optique, il est évident que les optimisations à apporter devraient influencer positivement les performances en charge du système. Les performances en charge d'un système représentent, d'une manière générale, la capacité de traitement de ce système lorsqu'il est soumis à un nombre important de requêtes (au sens de demande de traitement d'information). Cette capacité de traitement peut être exprimée en octets par seconde.

Dans le cadre du MMS, la charge représente le nombre moyen de transactions par seconde que le système est capable de traiter au maximum de ses possibilités. Une transaction commence à l'envoi d'un message par un expéditeur et se termine à la fin de son téléchargement par tous ses destinataires. La taille moyenne des messages a une influence très importante sur cette mesure. En effet, un système pourra traiter beaucoup plus de transactions par seconde si la taille moyenne des messages impliqués est de 10kb que si celle-ci est de 100kb. De la même manière, un nombre élevé de destinataires par message à une influence négative sur cette mesure.

Nous avons choisi de nous intéresser à cette dernière remarque pour orienter nos efforts d'optimisations. Dans les sections suivantes, nous allons tenter de montrer pourquoi l'optimisation du traitement des messages à plusieurs destinataires nous paraît être une piste susceptible de fournir des gains de performance appréciables.

Remarquons que nous ne pourrons faire appel à des optimisations qui réduiraient la facilité de maintenance de manière trop importante. Et, quoi qu'il arrive, toute optimisation qui mettrait en

péril la propriété *scalable* du système devra être écartée.

## 3.1 Les zones critiques

Pour débuter nos recherches, nous avons tout d'abord réalisé des mesures sur les performances du système au moyen d'un logiciel de type *profiler*[1]. Nous avons mesuré le temps qui était passé dans chacune des *proxylets* du *Nextenso MMS Proxy Relay* durant un test de charge qui consistait à réaliser un nombre important de transactions (envoi d'un message, notification de réception, téléchargement du message) pendant plusieurs minutes afin de pousser le système au maximum de ses possibilités.

Malheureusement, nous avons perdu les tableaux résultant de ces mesures et n'avons pu les inclure dans ce document. Nous prions le lecteur de bien vouloir nous en excuser. Cependant, il est apparu très nettement que les zone critiques d'optimisations sont les *proxylets* de type codec (MM1, MM7, ...) ainsi que la *proxylet* d'adaptation de contenu. Cette dernière étant la plus gourmande.

Nous aurions pu obtenir les mêmes conclusions simplement en raisonnant. En effet, ces *proxylets* sont les seules à traiter l'entièreté d'un message en transit et non seulement certaines parties voir uniquement les en-têtes. Si ce type de raisonnement n'est pas toujours vrai, notre connaissance des processus mis en oeuvre au sein des différentes *proxylets* nous permet de faire ce rapprochement entre quantité de données traitées et temps de traitement. De plus, les traitements réalisés au niveau de ces deux *proxylets* sont assez lourd : encodage d'un message ou bien adaptation de son contenu qui peut consister à effectuer diverses opérations sur des fichiers contenant des images, des vidéos ou du son.

Nous avons donc décidé d'orienter nos efforts sur la recherche d'optimisations sur ces *proxylets* car elles auront un impact très important sur les performances en charge du *Nextenso MMS Proxy Relay*.

## 3.2 Les cas d'utilisation

Les optimisations que nous avons tenté d'effectuer sont des optimisations plutôt conceptuelles qui peuvent être dérivées de l'analyse de certains cas d'utilisations.

Comme nous l'avions déjà suggéré dans le chapitre 1, les propriétés multi-destinataires du MMS introduisent une certaine redondance dans le traitement du message. Par exemple, en supposant que tous les destinataires possèdent tous le même type de terminal, l'opération d'encodage et d'adaptation sera redondante pour chaque récupération du message par un destinataire. Ce type de redondance entraîne une sur-utilisation du processeur qu'il pourrait être intéressant de réduire.

[1] Ce type de logiciel permet d'analyser, notamment, l'utilisation de la mémoire et du processeur par les différentes parties d'un programme durant son exécution.

S'il est vrai que cette supposition n'est pas très réaliste, tout le monde pourra aisément voir que ce type de redondance pourrait faire l'objet de nombreuses optimisations et qu'il est intéressant de pousser quelque peu ce raisonnement.

Considérons donc ce cas d'utilisation plus réaliste : très courante au moyen des e-mail, les services de type liste de diffusion ou groupe de discussion ont également un franc succès au moyen des SMS. Pour s'en convaincre, il suffit de constater à quel point notre quotidien est inondé de publicités pour ce type de service : recevoir chaque jour son horoscope, des informations boursières, la météo, .... On peut donc gager que ce succès sera encore plus grand avec les MMS puisque ces derniers se rapprochent de l'e-mail et permettent d'utiliser un contenu beaucoup plus riche.

Prenons, par exemple, le cas d'une liste de diffusion qui compterait un millier d'abonnés. On peut raisonnablement supposer que ces abonnés ne possèdent pas tous un type de terminal différent. En les groupant par type de terminal, nous retombons sur la situation présentée plus haut.

Toujours dans cette optique, nous pourrions imaginer d'éliminer cette redondance par l'utilisation d'une sorte de cache qui permettrait de sauver les MMS dont le contenu a été adapté et qui ont été encodés pour un type particulier de terminal afin d'éviter de répéter plusieurs fois ces deux opérations.

C'est cette piste de réflexion que nous avons décidé d'employer pour réaliser l'optimisation du *Nextenso MMS Proxy Relay*. Nous nous bornerons donc à développer cette piste tout au long du reste de ce document.

Par ce biais, il y a une forte probabilité d'améliorer les performances globales du système car nous pourrons réduire la redondance de traitement présente au niveau de la totalité des *proxylets* les plus gourmandes du *Nextenso MMS Proxy Relay*.

Notons que le choix de cette piste est assez personnel. Même s'il est motivé par les faits que nous avons tentés de vous exposer dans ce chapitre, il pourrait exister d'autres pistes d'optimisation qui permettraient de réaliser également un gain de performance appréciable. Par exemple, tenter d'optimiser les processus eux-mêmes : l'adaptation de contenu et l'encodage des messages. Cependant, ce type d'optimisation est plus compliqué car, outre les optimisations d'implémentation, il aurait également convenu d'optimiser les algorithmes réalisant ces opérations. L'investissement en temps nous paraissait beaucoup plus important par rapport au gain que l'on pouvait en espérer. De plus, concernant l'adaptation de contenu, une partie des opérations associées est réalisée au moyen de logiciels spécialisés qu'il ne nous paraissaient pas évident d'optimiser.

Afin de nourrir notre réflexion, nous allons maintenant faire le point, dans la partie suivante, sur les techniques actuelles en matière de développement de caches.

## Deuxième partie

# La mémoire cache

# Chapitre 4

# La mémoire cache

## 4.1 Définition

La définition la plus générale (du point de vue informatique) qui peut être faite d'une mémoire cache[1] est la suivante :

> *"la mémoire cache est une zone de mémoire dans laquelle on peut stocker des données plus ou moins temporairement afin d'y accéder rapidement."*

Remarquons que le terme *zone de mémoire* doit être compris dans le sens le plus générique. En effet, il peut tout aussi bien s'agir d'une partie de la mémoire centrale de l'ordinateur que d'une mémoire spécialisée séparée de la mémoire centrale, d'une mémoire persistante ou d'une mémoire distribuée entre plusieurs machines.

Il en va de même pour le terme *donnée* qui désigne aussi bien un octet qu'une page de mémoire ou une structure complexe.

A l'origine, la mémoire cache a été développée pour servir de "tampon" entre la mémoire centrale et l'unité de calcul (processeur) de l'ordinateur. En effet, les temps d'accès[2] à la mémoire centrale ne permettaient pas d'utiliser au mieux les capacités de calcul du processeur. Ce type de mémoire, extrêmement rapide, est utilisé pour stocker les données en cours d'utilisation ainsi que la partie du code de l'application utilisée actuellement par le processeur.

Malheureusement, ce type de mémoire est beaucoup plus onéreux que la mémoire centrale et, par conséquent, seule une quantité limitée peut être utilisée. Il a donc fallu développer des algorithmes permettant de gérer au mieux cette capacité tout en assurant un niveau confortable de performance.

Par la suite, une idée a été d'utiliser une partie de la mémoire centrale pour servir de mémoire cache pour le disque dur (dont les temps d'accès et taux de transferts sont beaucoup plus faibles

[1] Dans la suite de ce document, nous utiliserons indifféremment la dénomination *mémoire cache* ou *cache*.
[2] Notons que dans ce document, le terme *accès* désigne aussi bien la lecture que l'écriture.

que la mémoire centrale).

Avec le développement de l'Internet et des applications distribuées, un nouveau cadre d'utilisation est apparu. Une partie de la mémoire centrale d'une machine peut être utilisée pour stocker temporairement des informations provenant d'autres machines (et ainsi réduire les temps d'attentes conséquents à l'utilisation d'un réseau).

Ces différentes considérations nous permettent de donner une autre définition de la mémoire cache, plus adaptée à notre contexte :

> *"Une mémoire cache est une zone de mémoire utilisée pour améliorer les performances d'accès à une autre zone de mémoire (beaucoup) plus lente et de (beaucoup) plus grande capacité."*

Afin de faciliter la lecture de la suite de ce document, nous prendrons comme convention de nommer cette zone de mémoire plus lente la *mémoire cible*.

### 4.1.1 Principes fondateurs

Intéressons nous un peu plus aux principes qui ont motivé le développement de la mémoire cache. Il est maintenant clair que le but principal de l'utilisation d'une mémoire cache est de réduire le temps d'attente nécessaire pour l'accès à une donnée en vue de lui appliquer un traitement quelconque.

Mais étant donné que la donnée provient d'un type de mémoire plus lente, comment l'utilisation d'un "intermédiaire" supplémentaire pourrait-il réduire ce temps ? On trouve dans la littérature trois[3] principes empiriques fondamentaux qui permettent de justifier cette utilisation : la localité spatiale, la localité temporelle et la séquentialité.

Le principe de la localité spatiale consiste en le fait que, si un accès à une donnée a lieu, il est fort probable que les données *voisines*[4] fassent l'objet d'un accès dans un futur proche.

Un exemple assez intuitif de ce phénomène concerne la lecture d'un fichier sur le disque dur. Lorsqu'un accès aux premiers secteurs contenant le fichier a lieu, il parait assez logique que les secteurs contenant la suite du fichier soient lus dans un futur très proche. Dans ce contexte, le *voisinage* concerne la proximité physique des secteurs d'un disque dur.

Dans le contexte d'un site web, par exemple, le *voisinage* d'une page concernera plutôt l'ensemble des pages qui sont référencées par un hyperlien dans cette page.

Le principe de la localité temporelle suppose que, lorsqu'un accès à une donnée a lieu à un moment particulier, il est fort probable qu'un nouvel accès à cette même donnée se produise dans un futur plus ou moins proche.

[3] Notons que certains auteurs ne citent que les deux premiers.

[4] La sémantique de ce terme dépendra fortement du contexte d'utilisation de la cache.

Assez intuitivement, on comprend que cela s'applique bien à un programme informatique notamment à cause de la structure des langages impératifs. Par exemple, la condition d'une boucle sera vérifiée plusieurs fois et donc les données associées à cette condition seront accédées plusieurs fois durant un certain laps de temps.

Il apparaît moins évident que ce principe puisse s'appliquer à d'autres situations et pourtant des études ont prouvé qu'il était toujours valable. Par exemple, on trouvera dans [19] une étude sur le nombre de consultations de pages web qui confirment ce principe.

Ce principe induit les notions de *donnée chaude* et *donnée froide*[5] représentant respectivement une donnée qui est fortement utilisée pour le moment (ou qui le sera dans un futur proche) et une donnée qui est peu, voir pas utilisée pour l'instant.

Le principe de séquentialité est en fait une conséquence de la structure des programmes informatiques combinée avec les deux principes précédents. Il s'applique donc moins à d'autres contextes, c'est la raison pour laquelle ce principe n'est pas cité par tous les auteurs.

La plupart des programmes informatiques sont constitués d'un nombre très important de boucles. A l'intérieur de ces boucles, les mêmes données sont souvent accédées. Dès lors, des algorithmes de gestion de caches pourront se baser sur ce principe pour améliorer les performances des boucles.

### 4.1.2 Indicateurs

Lorsqu'un accès à la mémoire cible doit avoir lieu, une première tentative est effectuée au moyen de la mémoire cache. Cependant, comme nous l'avons vu plus haut, le rapport entre la taille de la mémoire cache et de la mémoire cible est inférieur à un. Cela implique qu'il est impossible de placer dans la mémoire cache l'entièreté des données *pouvant* être contenue dans la mémoire cible et que, par conséquent, cette tentative peut échouer.

On qualifiera de *cache hit* (ou plus simplement *hit*) la réussite de cette tentative (c'est-à-dire que la donnée recherchée se trouve dans la mémoire cache). Dans le cas contraire, on dira qu'il y a eu un *cache miss* (ou *miss*).

Il existe plusieurs raisons pour qu'un *cache miss* se produise. Arun Chauhan et al. [7] proposent le classement suivant des principales causes de cette situation :

**Compulsory miss** C'est la première fois que cette donnée est demandée alors qu'elle ne se trouve pas dans la cache. (Ce type d'échec est très courant quand la politique de placement est de type *premièr accès* - nous y reviendrons dans la section suivante).

**Capacity miss** Le nombre de données actuellement demandées est plus important que celui que peut contenir la mémoire cache. Dans ce cas, il y a toujours des *données chaudes* qui ne peuvent être stockées dans la mémoire cache.

**Conflict miss** Comme nous le verrons dans la section suivante, à cause du mode d'adressage

[5] Ces notions sont introduites dans [21] sous la dénomination *hot & cold blocks* et *warm & cold blocks* dans [14].

de certaines mémoires caches, il se peut que deux données distinctes (par exemple A et B) provenant de la mémoire cible entrent en compétition pour la même adresse dans la mémoire cache. Il se produit ce type d'échec lorsque l'on essaie d'accéder à la donnée B alors que c'est la donnée A qui occupe actuellement cette adresse dans la mémoire cache.

Anant Agarwal et al. [2] introduisent un autre classement, partant du point de vue du programme utilisateur, des raisons pouvant causer un *cache miss* :

**Start-up effect** C'est un effet qui résulte du démarrage d'un nouveau programme. Le temps que celui-ci accède à toutes ses données de travail, de nombreux *cache miss* vont se produire. Il peut également être observé lorsqu'un programme change brutalement sa phase d'exécution (et donc les données sur lesquelles il travaille).

**Nonstationary behavior** C'est l'effet qui résulte de l'évolution de l'exécution du programme au cours du temps. De cette évolution découle que les données utilisées commencent à changer peu à peu, ce qui induit un certain nombre de *cache miss* sur ces nouvelles données.

**Intrinsic interference** Comparable au *conflict miss* du classement précédent, c'est le type d'interférence qui apparaît lorsqu'un programme utilise simultanément plusieurs données entrant en compétition pour la même adresse dans la mémoire cache.

**Extrinsic interference** Ce type d'interférence regroupe les cas où deux programmes distincts (nous sommes donc dans un contexte multi-tâches) utilisent simultanément des données en compétition pour la même adresse. Typiquement, les *cache miss* en résultant n'apparaîtront qu'au moment des changements de contexte (c'est-à-dire aux moments où le processeur est attribué à un autre programme).

Afin de pouvoir quantifier les notions de *cache hit* et *cache miss*, deux indicateurs ont été introduits par la littérature : il s'agit du *hit rate* et *miss rate*[6] respectivement.

Le *hit rate* est la probabilité qu'une donnée faisant l'objet d'un accès se trouve effectivement dans la cache et est défini par :

$$hit\ rate = \frac{\#\ cache\ hit}{\#\ total\ d'accès}$$

De même, le *miss rate* exprime la probabilité complémentaire et est défini par :

$$miss\ rate = 1 - hit\ rate$$

Comme nous l'avons vu, le but de l'utilisation d'une cache est de minimiser les temps d'accès aux données de la mémoire cible. Au vu de ces indicateurs, il apparaît clairement que, pour atteindre ce but, il faut maximiser le *hit rate* (ou minimiser le *miss rate*).

[6]Dans la littérature ils sont parfois aussi appelés par *hit ratio* et *miss ratio* respectivement.

En fait, il faut impérativement que le *hit rate* ne soit pas inférieur à une valeur minimum (dépendante du contexte) car un système contenant une mémoire cache qui ne respecterait pas cette propriété serait moins performant que le même système dépourvu de cette mémoire cache. En effet, à chaque *cache miss* est associée une certaine pénalité (appelée, dans la littérature, *miss penalty*) qui exprime le temps supplémentaire qu'il sera nécessaire de dépenser pour l'accès à la donnée cible.

Tout en cherchant à maximiser le *hit rate*, il est donc également utile de réduire le *miss penalty* afin d'augmenter les performances globales du système.

La maximisation du *hit rate* dépend fortement du contexte d'utilisation et de très nombreux algorithmes ont été développés pour faire face à beaucoup de situations différentes. Cependant, un certain nombre de règles générales de design permettent de s'assurer un bon *hit rate*. La section suivante va présenter en détail ces différentes méthodes ainsi que leur importance dans la valeur effective de cet indicateur.

## 4.2 Design

Nous allons maintenant passer en revue les différents points critiques dont il faut tenir compte lors de la réalisation d'une mémoire cache (que ce soit une mémoire cache matérielle ou logicielle) et qui influenceront fortement les performances de la cache. Ils doivent donc être soigneusement étudiés en fonction du contexte d'utilisation.

### 4.2.1 Associativité

La première question qui se pose à propos de l'utilisation d'une mémoire cache est de savoir de quelle manière les données à mettre dans la cache vont être adressées. Nous disposons de l'adresse de la donnée dans la mémoire cible mais nous ne pouvons l'utiliser telle quelle car, comme nous l'avons vu, la taille de la mémoire cache (et donc son espace d'adressage) est toujours inférieure à celle de la mémoire cible.

L'associativité d'une mémoire cache définit le rapport qui lie l'adresse d'une donnée dans la mémoire cible et son adresse dans la mémoire cache.

Il existe trois grandes familles d'associativités : associativité simple[7], associativité multiple[8] et associativité complète[9].

Les mémoires caches à associativité simple (voir figure 4.1) utilisent une fonction de *hashing*[10] afin de rendre compatible l'adresse de la mémoire cible avec l'espace d'adressage de la mémoire cache. Si la mémoire cache dispose d'un espace d'adressage de $N$ adresses, la fonction

[7] Dénommées *direct-mapped caches* ou *1-way associative caches* dans la littérature.
[8] Dénommées *set-associative caches* ou *n-way associative caches* dans la littérature.
[9] Dénommées *fully-associative cache* dans la littérature.
[10] Définie en fonction du contexte d'utilisation, il s'agit le plus souvent d'une opération de "modulo".

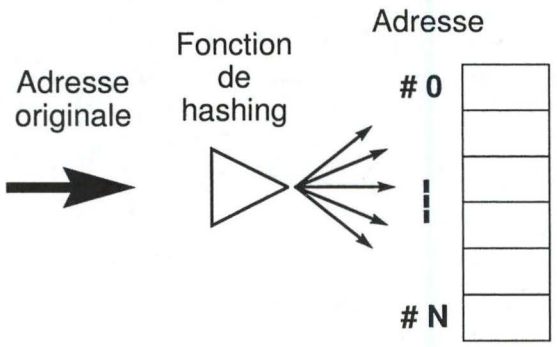



FIG. 4.1 – Organisation d'une cache avec associativité simple

$f(adresse) = adresse \ \% \ N$ pourra, par exemple, être utilisée.

Evidemment, cela entraîne des conflits d'adresses car plusieurs données possédant des adresses distinctes dans la mémoire cible peuvent obtenir la même adresse dans la mémoire cache. Il faut donc mémoriser l'adresse originale (ou tout autre moyen de différenciation) de la donnée lorsqu'elle est placée dans la mémoire cache afin de pouvoir vérifier, lors d'un accès, que c'est bien la bonne donnée qui se trouve dans la cache.

Lors d'accès fréquents à deux données dont les adresses sont conflictuelles, sachant qu'elles ne peuvent se trouver simultanément dans la cache, il risque de se produire un grand nombre de *cache miss* dégradant fortement les performances de la cache.

La pire situation est celle où les accès à ces deux données sont entrelacés : accès *un*, accès *deux*, accès *un*, accès *deux*, ... Il est assez probable que l'algorithme placera toujours dans la cache la mauvaise donnée au mauvais moment, provoquant ainsi une suite de *cache miss* très importante.

Pour contrer ce phénomène, une associativité multiple (voir figure 4.2) peut être utilisée. Au lieu d'adresser la mémoire cache par donnée, on va l'adresser par lignes. Ainsi, à chaque adresse de la mémoire cache, deux ou plusieurs données pourront être insérées[11].

Le premier désavantage est un temps d'accès un peu plus long. En effet, après avoir converti l'adresse de la mémoire cible, il faudra encore effectuer une à plusieurs comparaisons pour sélectionner la donnée correcte (encore une fois, cette comparaison pourra avoir lieu sur base de l'adresse originale ou sur tout autre élément discriminant des données).

Le second désavantage est que l'on utilise la place disponible dans la mémoire cache encore plus mal que dans le cas de l'associativité simple. Une ligne de la cache ne pourra servir qu'à stocker des données dont l'adresse dans la mémoire cache est la même. Si peu de données possédant cette propriété sont utilisées, beaucoup d'espace de stockage sera inutilisé et donc gaspillé.

Pour répondre à ce dernier désavantage, on peut utiliser une mémoire cache à associativité

[11] Une mémoire cache dont les lignes permettent d'insérer deux données est appelée *2-way associative*, ...

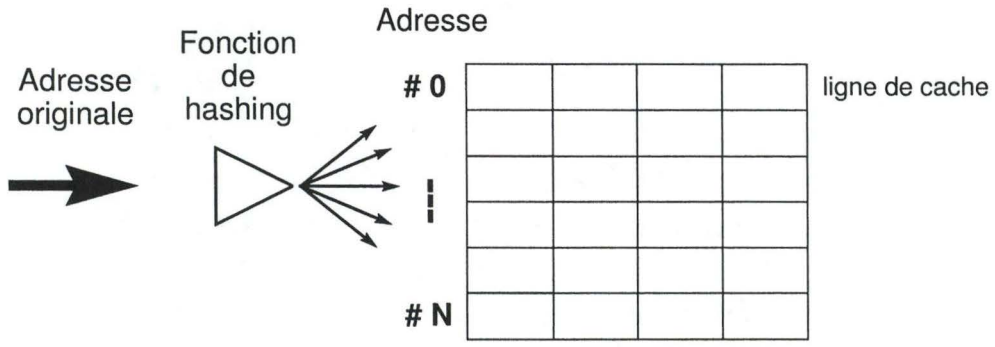



FIG. 4.2 – Organisation d'une cache avec associativité multiple

complète. Il s'agit en fait d'un cas particulier d'associativité multiple où la mémoire cache ne possède qu'un seul ensemble de données. La fonction de *hashing* n'est donc pas utilisée. Au lieu de cela, les données sont chaînées dans la cache et, lors d'un accès, tous les éléments présent dans la cache sont comparés. Le discriminant pourra ici aussi être l'adresse dans la mémoire cible mais il est préférable d'en choisir un qui permette de trier les éléments dans la cache afin de diminuer le temps d'accès.

Les temps d'accès sont très longs (surtout en cas de *miss*), ce qui confine ce type d'associativité à des mémoires caches de très petites tailles (pour limiter le nombre de comparaisons nécessaires).

Le niveau d'associativité affecte fortement le *hit rate*. D'une manière générale, plus l'associativité est élevée, meilleur sera le *hit rate* [11] (car le taux d'occupation de la mémoire cache est meilleur). Cependant, cela a un coût non négligeable car, comme nous venons de le voir, plus l'associativité est élevée et plus le temps d'accès à la mémoire cache est long. Il faut donc équilibrer correctement ce paramètre en fonction du contexte.

### 4.2.2 Cache multi-niveaux

L'organisation d'une cache en plusieurs niveaux consiste à placer plusieurs mémoires caches devant la mémoire cible. Tout se passe comme si chaque niveau de mémoire cache avait pour mémoire cible la mémoire cache du niveau inférieur, ainsi de suite jusqu'à la mémoire cible d'origine.

C'est d'ailleurs cette méthode qui est utilisée au niveau de la mémoire cache des processeurs modernes[12]. Comme nous le voyons à la figure 4.3, le processeur est couplé avec une mémoire cache de niveau un. Elle est extrêmement rapide (le temps d'accès est synchronisé sur le cycle d'horloge du processeur) mais sa taille est extrêmement réduite. Donc pour réduire le *miss penalty* associé à cette cache, on place juste derrière une autre mémoire cache (de niveau deux), qui est un peu plus lente mais de capacité plus importante. Elle permet d'avoir, en quelque sorte, une seconde

[12] voir [11].

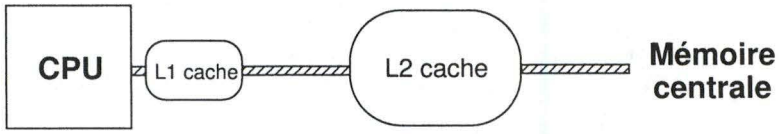



FIG. 4.3 – Organisation des deux niveaux de mémoire cache d'un processeur

chance d'obtenir la donnée cible relativement rapidement.

Ce paramètre n'a pas d'influence sur le *hit rate* de la cache de premier niveau mais il réduit fortement le *miss penalty* en offrant une ou plusieurs chances supplémentaires de trouver la donnée avant de se résoudre à consulter la mémoire cible.

### 4.2.3 Politique de placement

La politique de placement doit décider quelles données placer dans la cache tout au long de l'utilisation de celle-ci. Il s'agit uniquement de décider quelles données choisir, indépendamment de la place disponible. C'est la politique de remplacement qui décidera de la manière de trouver de la place s'il n'y en a plus suffisamment.

Dans la plus grande majorité des cas, la politique choisie est celle de l'accès[13] : Chaque donnée faisant l'objet d'un *cache miss* sera insérée dans la cache. Bien que très répandue, cette politique est très simple. Cependant, il semble que peu de recherches aient été effectuées sur l'amélioration de ce type de politique. Les chercheurs ont plutôt tenté de concentrer leurs efforts sur les politiques de remplacement où bon nombre des désavantages introduit par la politique de placement par accès peuvent de toute façon être résolu.

Zhifeng Cheny et al. [9] proposent néanmoins une analyse plus fine des politiques de placement en introduisant la métrique suivant : la *idle distance* qui mesure la durée que passe une donnée dans la cache avant d'être accédée pour la première fois. Les auteurs affirment qu'une bonne politique de placement devrait minimiser cette valeur au maximum.

Ces recherches ont mené au développement de la *eviction-based placement policy* qui s'applique au contexte des caches à plusieurs niveaux. Les données choisies pour entrer dans un niveaux de la cache sont celles qui ont été retirées du niveau supérieur (par effet de la politique de remplacement). Cette politique a été spécialement optimisée pour une utilisation par un système distribué.

Ce paramètre a un effet important sur le *hit rate* dans la mesure où il influence le taux d'occupation de la mémoire cache par des données *pertinentes*[14]. Cependant, il est extrêmement difficile

[13] Dénommée *access placement based policy* dans la littérature.

[14] Par perninentes, nous entendons les données pour lequelles le fait d'être placé en cache améliore les performances globales du système utilisant la cache.

de savoir *a priori* quelles sont les données pertinentes ce qui explique l'utilisation répandue de la politique par accès. En effet, c'est au cours du temps qu'une donnée passe dans la cache que sa pertinence (c'est-à-dire de son degré d'utilisation) peut être évaluée, permettant de savoir si elle doit s'y trouver ou non.

### 4.2.4 Politique de remplacement

La politique de remplacement consiste à savoir quelles données présentes dans la cache doivent être enlevées afin de faire de la place à d'autres données plus pertinentes.

L'influence de ce paramètre sur le *hit rate* est aussi importante que la politique de placement. Seulement, dans la situation d'une donnée qui a déjà passé un certain temps en mémoire cache, beaucoup plus d'informations sont disponibles pour évaluer sa pertinence. C'est la raison pour laquelle la majorité des efforts de recherches se sont portés sur le développement de nouvelles politiques de remplacement de plus en plus efficaces et adaptées à chaque type d'utilisation d'une cache.

Il s'agit certainement du paramètre le plus critique lors du développement d'une cache. Etant donné l'importance de celui-ci, nous avons décidé de lui consacrer le chapitre suivant afin de pouvoir présenter en détail les différentes alternatives possibles.

### 4.2.5 Politique d'écriture

La politique d'écriture consiste à savoir quoi faire en cas d'accès en écriture à une donnée selon qu'elle soit présente dans la mémoire cache ou non.

Si la donnée est présente en cache, il est possible d'effectuer la modification dans la mémoire cache uniquement (*write back*). Dans ce cas, on écrira la modification dans la mémoire cible lorsque la donnée sera évincée de la mémoire cache. Cela suppose l'utilisation d'un indicateur de modification associé à chaque donnée dans la cache : on ne réécrit que les données modifiées. Cette méthode est performante mais présente des risques d'incohérence en cas de défaillance du système.

A l'opposé, les modifications peuvent être effectuées à la fois dans la mémoire cache et dans la mémoire cible (*write through*). Beaucoup plus sûre, cette politique engendre néanmoins un traffic mémoire plus important et augmente tous les temps d'écriture.

Dans le cas où la donnée n'est pas présente dans la cache, la politique du *write allocate* consistera à la placer d'abord dans la cache avant sa modification selon une des deux politiques présentées plus haut bien que, dans ce cas, la politique de *write through* n'ait pas beaucoup de sens.

Ce paramètre n'influence que très marginalement le *hit rate*, son influence se limitant aux durées des écritures. Ce n'est donc pas un paramètre crucial de design.

### 4.2.6 Victim cache

Le concept de *victim cache* [11], [7] (ou *ghost cache* [16]) a été introduit pour pallier aux erreurs d'appréciation des différentes politiques de remplacement. Il s'agit d'utiliser une petite cache à associativité complète organisée en FIFO[15] dans laquelle sont placées toutes les données évincées de la cache principale.

De cette manière, si une donnée à été évincée de la cache principale alors qu'il s'agissait d'une donnée pertinente, il est possible de la récupérer rapidement.

S'il est vrai que ce paramètre n'a aucune influence sur le *hit rate*, il réduit fortement le *miss penalty* des données évincées de la cache par erreurs. Ce type d'eviction erronée est tout à fait normal à cause du caractère approximatif inhérent à tout algorithme de gestion d'une cache (il est en effet impossible de connaître exactement la pertinence d'une donnée).

[15] First In First Out

# Chapitre 5

# Politiques de remplacement

Ce chapitre fait un tour d'horizon des différentes politiques de remplacement qui ont été développées au cours des dernières années. Il ne se veut certainement pas exhaustif compte tenu du nombre extrêmement élevé de politiques existantes. Cependant, nous tenterons de grouper les principales politiques par familles de même type.

Chacune des politiques présentées ici tente de maximiser le *hit rate*. Le point de référence est l'algorithme *optimal* suivant : on élimine de la cache les données qui seront accédées le plus loin dans le futur. C'est évidemment un algorithme théorique qui ne peut être implémenté en pratique mais dont la littérature a montré qu'il était optimal.

Remarquons que nous ne présenterons dans ce document aucune valeur numérique de *hit rate*. En effet, chaque auteur fait ses propres comparaisons sur des échantillons distincts. Il est donc difficile de comparer ces chiffres de manière objective. N'ayant pas la possibilité de réaliser nous même ce type de tests comparatifs, nous avons préféré substituer des conclusions plus générales à de telles valeurs.

Attention, le *hit rate* et le *miss penalty* ne sont toutefois pas les seuls paramètres à prendre en compte. En effet, la complexité d'implémentation, qui a comme conséquence directe le temps nécessaire pour obtenir un accès à une donnée est également importante. Nous présenterons donc plusieurs politiques offrant un *hit rate* similaire mais dont les temps d'accès varient fortement ; ce qui peut avoir un intérêt non négligeable selon le contexte d'utilisation.

Nous avons décidé de reprendre le classement fait par [14] en ajoutant une catégorie supplémentaire : les politiques dites dynamiques. Nous classerons donc les différentes politiques présentées ici entre politiques basées sur : la fréquence et/ou le moment du dernier accès, l'environnement d'utilisation (propriétés spécifiques des données dans un environnement particulier), les régularités (p.ex. séquences ou boucles) et les politiques dynamiques.

Avant de procéder à ce classement, revenons brièvement sur les toutes premières politiques de remplacement qui avaient été utilisées. Il ne s'agissait pas à proprement parler de politiques de remplacement au sens où elles n'avaient pas été développées spécialement dans ce but. Il s'agit

des politiques *random*, où les données à remplacer sont choisies au hasard et *FIFO*, où c'est la plus vieille des données se trouvant dans la cache qui est évincée au profit d'une nouvelle.

Il est évident que ces politiques donnaient de très mauvais résultats en terme de *hit rate* et que les chercheurs ont rapidement tenté de mettre au point de nouvelles politiques plus efficaces.

Notons enfin que les politiques de remplacement permettent de décider *quelle donnée* retirer de la cache et non *quand* retirer la donnée. Pour plus de clarté, nous présenterons la plupart des algorithmes dans la situation où il faut faire de la place dans la mémoire cache.

## 5.1 Politiques basées sur les accès

La majorité des politiques de remplacement basées sur les accès utilisent le moment du dernier accès ou bien la fréquence des accès. Nous commencerons par présenter les politiques basées sur un seul de ces facteurs pour nous intéresser ensuite à celles qui combinent les deux.

### 5.1.1 Dernier accès

Il semblerait que la plus ancienne politique de remplacement ([16] situe son apparition aux alentours de 1965 - voir plus tôt encore) soit la politique du *Least Recently Used* (LRU). Elle est basée sur le moment du dernier accès à une donnée : lorsqu'il faut choisir une donnée à retirer de la mémoire cache pour faire de la place à une nouvelle, c'est celle qui n'a plus été accédée depuis le plus de temps qui est choisie.

Pour l'implémenter, une file de priorité dans laquelle on classe les données par ordre décroissant du dernier accès est souvent utilisée. Ainsi, il suffit de choisir le dernier élément de la file lorsqu'il faut enlever une donnée de la cache. Cette implémentation augmente quelque peu le temps d'accès car chaque *cache hit* peut entraîner la modification de la position de la donnée dans cette file.

Bien que cette politique soit d'une simplicité déconcertante (autant conceptuellement que dans la façon de l'implémenter - ce qui est d'ailleurs un de ses grands avantages), elle donne, en pratique, d'excellents résultats au niveau du *hit rate*. En l'absence d'informations détaillées sur les données qui seront manipulées par la cache, c'est un très bon choix de base.

Nous allons d'ailleurs voir que de nombreuses recherches ont été effectuées pour tenter d'améliorer cette politique et réduire ses désavantages (au moins autant que de recherches développant des politiques totalement originales).

De nombreux auteurs ont identifiés, en fonction du contexte, plusieurs désavantages liés à l'utilisation de cette politique. Nous présentons ici une compilation de ces principaux désavantages que le lecteur pourra trouver dans [5][1], [14], [21] et qui nous servira de base pour analyser les

[1]Nous avons omis le deuxième désavantage présenté par cet auteur car celui-ci peut être résolu assez facilement en jouant sur l'implémentation de la politique car il ne s'applique qu'au contexte de la mémoire virtuelle.

améliorations apportées par les politiques développées sur base de la politique LRU.

1. Comme nous l'avons vu, lorsqu'un *cache hit* a lieu, la donnée concernée peut avoir à changer de position dans la file de priorité. Dans un environnement multi-tâches où plusieurs tâches peuvent tenter de déplacer une donnée simultanément dans cette file, il faut sérialiser ces différentes opérations derrière un verrou, ce qui handicape fortement les performances de ce genre de système.
2. La gestion de cette file de priorité impose, dans le meilleur des cas, une complexité logarithmique pour le temps d'accès à une donnée.
3. Cette politique ne tient aucun compte de la fréquence des accès. Ainsi, si une donnée est accédée très régulièrement mais avec des intervalles assez longs (plus longs que la moyenne des temps des derniers accès des données présentes dans la cache), cette donnée sera très probablement évincée de la cache alors qu'il serait plus logique de l'y laisser.
4. Elle ne tient pas compte des régularités de type boucles ou accès séquentiels. Pire, elle peut être polluée[2] par un programme qui passerait en revue toutes les données d'une liste par exemple.
5. Si une donnée a été fortement utilisée pendant un moment mais plus maintenant, il faudra tout de même attendre qu'elle redescende toute la file de priorité avant de quitter la cache et faire ainsi de la place pour une autre donnée.

Pour faire face au premier désavantage lié à la politique LRU, la politique CLOCK [5] a été développée en 1968. Elle permet d'éliminer la sérialisation des accès liés au verrou global. Au lieu d'utiliser une file de priorité, les données sont insérées dans la cache au moyen d'une liste circulaire (c'est-à-dire tel que l'élément suivant le dernier élément de liste soit le premier élément).

Chaque donnée placée dans la cache reçoit un *page reference bit* qui est mis à zéro. Lorsqu'un accès a lieu sur une des données présentes dans la cache, ce bit est mis à un. La sélection d'une donnée à remplacer dans la cache se fait en se déplaçant dans la liste circulaire à la manière des aiguilles d'une horloge : pour chaque donnée où le *page reference bit* est à un, il est remis à zéro ; lorsque ce bit est à zéro, la donnée est évincée de la cache.

Bien que cette politique ne tienne pas compte de l'heure exacte du dernier accès à une donnée, elle enlève bien de la cache les éléments qui n'ont plus été accédés depuis longtemps. En pratique, le *hit rate* de cette politique est comparable à celui du LRU tout en offrant de meilleures performances dans un environnement multi-tâches.

Song Jiang et Xiaodong Zhang [12] présentent la politique *Low-Interference Recency Set* (LIRS). Cette politique peut être résumée de la manière suivante : les auteurs définissent la valeur *Inter-Reference Recency* (IRR) attachée à une donnée comme le nombre d'accès à des données distinctes entre deux accès consécutifs à celle-ci. Ils supposent que, quand une donnée a une grande valeur de IRR, cette dernière risque d'être à nouveau assez grande dans un futur proche. Pour cette raison, les données possédant le plus grand IRR sont enlevées de la cache en premier.

[2] C'est à dire que des données non pertinentes peuvent tout de même être introduites dans la cache.

Outre les désavantages "classiques" de la politique LRU (principalement le problème des accès séquentiels et des boucles), les auteurs de cette politique ont voulu résoudre un autre problème apparaissant, par exemple, lors de l'utilisation de la politique LRU dans le cadre des bases de données. Etant donné la taille de l'index d'une grosse base de donnée, et malgré le fait que la probabilité d'un accès à l'index soit 100 fois plus élevée que l'accès à une donnée (nous mentionnons ici le contenu de la base), la politique LRU traitera indifféremment l'index et les données de la base alors qu'il serait plus judicieux de laisser plus de chance à l'index de rester dans la cache. La politique LIRS est une tentative de résolution de ce problème de manière générique.

Jusqu'à présent, nous avons considéré que les données à placer dans la mémoire cache étaient homogènes du point de vue de la taille. C'est à dire que les données à mettre en cache sont toutes de la même taille. Mais dans le cas du web, par exemple, les pages à mettre en cache n'ont pas toutes la même taille. Dans ce cas, la politique de remplacement ne doit pas simplement libérer *une place* dans la cache mais suffisamment d'espace pour placer le nouvel élément (ce qui peut impliquer de retirer plusieurs éléments de la cache).

A cette fin, une politique basée sur LRU a été développée : la politique du *Size-adjusted LRU* (résumée dans[8]). Elle évalue les données du point de vue de leur taille et du moment de leur dernier accès.

### 5.1.2 Fréquence

La principale politique de remplacement basée sur la fréquence est la politique du *Least Frequently Used* (LFU) (résumée dans [16]). Lorsque qu'une donnée est insérée dans la cache, on lui assigne un compteur qui sera augmenté à chaque accès. Ce sont les données avec le compteur le plus bas qui seront évincées de la cache pour faire de la place à de nouvelles (là encore, l'implémentation se fait, dans la majorité des cas, par l'utilisation d'une file de priorité mise à jour à chaque accès).

Pour éviter que des données qui ont été accédées un grand nombre de fois ne restent indéfiniment dans la cache même si elles ne sont plus actuellement utilisées, il faut régulièrement réduire (diviser) les compteurs de toutes les données présentes dans la cache par un facteur fixe (ce facteur fait partie des paramètres de configuration de la cache qui utilise ce type de politique de remplacement).

Cette politique partage les deux premiers désavantages de la politique LRU. De plus, elle ne tient absolument pas compte de l'historique des accès (c'est-à-dire à quel moment ont eu lieu les accès) et elle ne s'adapte pas bien aux changements dans le flux de données.

Contrairement à ce que son nom pourrait laisser penser, la politique LRU-K (résumée dans [15], [16]) est basée sur la fréquence (ou plus exactement la densité) des accès à une donnée présente dans la cache. Au lieu de mesurer le moment du dernier accès, on mesure le temps entre le $K^{eme}$ dernier accès et maintenant.

Plusieurs désavantages sont liés à cette politique. Comment évaluer une donnée qui n'est pas dans la cache depuis assez longtemps pour pouvoir consulter la $K^{eme}$ dernière référence (celle-ci n'étant pas disponible) ? Il faut traiter ce cas comme un cas particulier ce qui induit des difficultés supplémentaires d'implémentation.

La plus simple possibilité est de définir la variable *Correlated Information Period* (CIP) qui définit le temps minimum qu'une donnée qui n'a pas été consultée suffisamment de fois doit rester dans la cache avant de pouvoir être évincée. Cependant, fixer correctement ce paramètre est relativement difficile.

Considérons maintenant la situation suivante : nous utilisons une politique LRU-3 (c'est-à-dire que nous mesurons le temps entre maintenant et la $3^{eme}$ dernière référence), les données $A$ et $B$ ont été respectivement accédées aux temps $\{7, 9, 25\}$ et $\{2, 31, 34\}$. La politique LRU-3 va préférer évincer la donnée $B$ de la cache alors que, intuitivement, la donnée $B$ est celle qui sera le plus probablement consultée dans un futur proche.

Malgré ces désavantages, cette politique donne, en pratique, d'assez bons résultats pour une valeur de $K$ peu élevée. Ses auteurs conseillent d'ailleurs d'utiliser $K = 2$ pour obtenir les meilleurs résultats.

La politique de *Frequency-Based Replacement* (FBR) [14], [15], [16], [20] développée par J. T. Robinson et M. V. Devarakonda prend en compte la corrélation entre les différents moments des accès à la donnée.

L'algorithme nécessite de maintenir une liste LRU divisée en trois sections : nouvelle, moyenne et ancienne. Chaque donnée insérée dans la cache commence par être placée dans la section nouvelle en première position. A chaque donnée correspond un compteur d'accès. Lors d'un *cache hit*, la donnée considérée est transférée en première position dans la section suivante (de nouvelle à moyenne et de moyenne à ancienne). De plus, lorsque la donnée se trouvait dans la section moyenne ou ancienne, son compteur est incrémenté. Tout comme la politique LFU, les compteurs doivent régulièrement être divisés par un facteur fixé à l'avance afin d'éviter la pollution de la cache par des données qui ne sont plus utilisées maintenant.

[8] résume la politique du *Segmented LRU*. Là encore, malgré son nom, cette politique tient compte de la fréquence des accès. La cache est divisée en deux segments organisés en LRU : le segment probatoire et le segment protégé. Lorsqu'une nouvelle donnée est insérée dans la cache, elle est placée dans le segment probatoire. Dès qu'elle est accédée, elle passe dans le segment protégé. Les données qui doivent sortir du segment protégé (par manque de place) parce qu'elles sont arrivées au bout de la file LRU sont replacées dans le segment probatoire alors que celles qui doivent quitter le segment probatoire sont enlevées de la mémoire cache.

### 5.1.3 Combinée

Une idée ayant prévalu dans bon nombre de récentes recherches est de tenter de combiner les politiques basées sur la fréquence des accès et celles basées sur le moment du dernier accès.

Donghee Lee et al. [15] ont développé une politique qui combine les politiques LRU et LFU appelée politique du *Least Recently/Frequently Used* (LRFU). A chaque donnée présente dans la cache, on associe une valeur appelée *Combined Recency and Frequency* (CRF) telle que, lorsqu'il faut faire de la place dans la cache, les données avec la plus petite valeur de CRF sont évincées en premier.

Pour calculer cette valeur, il faut définir une fonction de poids $F(t)$ représentant le poids d'un accès en fonction du temps (t représente la durée entre un accès et le moment où la fonction est appelée). La valeur de CRF à un moment précis d'une donnée peut donc être calculée comme suit : $\sum_i F(t_i)$ avec $t_1, t_2, \ldots, t_n$ les temps entre chaque accès précédent à cette donnée et ce moment. Les auteurs montrent que, pour que cette fonction combine les effets des politiques LFU et LRU, il suffit qu'elle soit de la forme $F(t) = (\frac{1}{p})^{\lambda t}$ avec $p >= 2$ et $\lambda \in \{0, 1\}$.

La valeur de $\lambda$ est un paramètre de configuration très important puisque c'est lui qui va décider si la politique penche plutôt vers le LFU ou bien le LRU. En effet, si $\lambda = 0$, la politique LRFU correspondante est en faite une politique LFU. A l'opposé, si $\lambda = 1$, la politique dégénère en une simple politique LRU.

D'après les tests réalisés par les auteurs, ce type de politique a au moins les mêmes performances que les politiques LFU et LRU mais permet, quand la valeur de $\lambda$ est bien configurée en fonction du contexte, d'obtenir de meilleures performances.

Cette politique a été étendue vers une version dynamique : la politique *Adaptive LFRU* (ALFRU) résumée dans [16] et [20]. Cet algorithme fait varier dynamiquement le paramètre $\lambda$ en fonction du flux de données à mettre en cache. Le détail de cet algorithme ne sera pas présenté ici. Notons tout de même que le paramètre $p$, qui influence fortement les performances, doit toujours être fixé à l'avance, cette politique n'est donc pas entièrement dynamique.

Dans la même optique, Sabarinathan Sayiraman et al. [21] ont développé la politique *Most Frequently-Least Recently Used* (MF-LRU) qui combine les politiques MFU[3] et LRU. Cette politique est assez compliquée à mettre en oeuvre. Son idée principale est de retirer de la cache les données qui était *anciennement* fréquemment accédées. A cette fin, les auteurs définissent le *Recency Frequency Factor* qui combine la mesure du dernier accès avec la mesure de la fréquence des accès. Tout comme la politique LRFU, de nombreux paramètres doivent être définis *à priori* de manière correcte en fonction du contexte pour obtenir de bonnes performances.

Nimrod Megiddo and Dharmendra S. Modha [16] ont développé une politique appelée *Adaptive Replacement Cache* (ARC) qui, tout comme la politique ALFRU, s'adapte dynamiquement entre fréquence et dernier accès en fonction du contexte.

[3]Non présentée dans ce document, cette politique interdit tout éviction de la cache des données les plus fréquemment utilisée, le lecteur désireux d'en savoir plus pourra consulter [17].

L'idée est de maintenir deux listes LRU $L_1$ et $L_2$. Dans la première, on place les données qui ont été accédées une seule fois *récemment*. Dans la seconde, les données qui ont été accédées au moins deux fois *récemment*. Les éléments se trouvant dans la liste $L_2$ ont donc un temps inter-accès relativement faible, et donc une fréquence élevée. Ainsi, il apparait que la liste $L_1$ est sensible au paramètre du dernier accès alors que la liste $L_2$ à la fréquence. Lorsqu'il faut enlever des éléments, ce sont les éléments les plus vieux des deux listes qui sont retirés.

Le facteur d'adaptation dépend de la taille des deux listes. Si elles sont de longueur égale, la prise en compte des deux paramètres de fréquence et d'accès est équilibrée. Sinon, la balance varie plutôt d'un côté ou de l'autre. Cet équilibre n'est pas fixé par configuration mais varie en temps réel[4] pendant l'exécution de l'algorithme, permettant ainsi de s'adapter aux changements du flux de données. Une autre conséquence est que cette politique est résistante aux accès en séquence (parcours d'une liste par exemple).

Les auteurs ont tenté de développer une politique performante mais en limitant tant que possible la complexité d'implémentation. A titre d'exemple, ils affirment que leur politique est deux fois plus rapide que la politique LRU-2 et 50 fois plus rapide que la politique LRFU (nous parlons ici du temps d'accès aux données et non du *hit rate*).

Sorav Bansal et Dharmendra S. Modha [5] ont développé, à partir de la politique CLOCK et ARC, la politique *CLOCK with Adaptive Replacement* (CAR). Leur point de départ est que ces deux politiques permettent de répondre chacune à deux désavantages de la politique LRU. En les combinant, on peut obtenir une politique avec un *hit rate* comparable mais possédant moins de désavantages.

C'est une politique relativement complexe, nous n'en présenterons ici que l'idée principale : il s'agit de maintenir deux listes CLOCK $T_1$ et $T_2$. La première contiendra les données présentant un intérêt à court terme (c'est-à-dire qui sont analysées sous l'optique du dernier accès) et la seconde les données présentant un intérêt à long terme (et sont donc analysées sous l'optique de la fréquence des accès). Comme dans la plupart des algorithmes de ce type, les données doivent d'abord passer un *certain* temps dans la liste $T_1$ avant de pouvoir passer dans la liste $T_2$ d'où les données sont plus difficilement évincées. Les auteurs utilisent un algorithme permettant de définir dynamiquement la taille de ces deux listes.

Comme dans la politique ARC, les auteurs ont utilisé deux *cache hits* successifs dans un *court* laps de temps (et donc la notion de fréquence) pour mesurer la notion d'utilité à long terme. C'est évidemment une limitation qu'ils ont tenté de réduire en étendant l'algorithme précédent en *CAR with Temporal filtering* (CART). Ce nouvel algorithme utilise une *fenêtre temporelle locale*[5] de taille variable pour mesurer les *cache hits* sur les données sur un plus grand laps de temps et prédire ainsi mieux les données présentant un intérêt à long terme. Le lecteur trouvera dans [5] ces deux algorithmes présentés en détails.

[4] L'explication de la manière utilisée pour atteindre ce but est au delà des perspectives de ce document. Nous invitons le lecteur désireux d'obtenir plus d'information à consulter [16].

[5] Traduction littérale de l'auteur

Dans l'optique du stockage en mémoire cache de données à taille variable, Kai Cheng et Yahiko Kambayashi [8] ont développé la politique *LRU Size-adjusted and Popularity-aware* (LRU-SP) qui combine les politiques *Segmented LRU* et *Size-adjusted LRU*.

A chaque donnée, on assigne le rapport $\frac{\#d'accès}{taille\ de\ la\ donnée}$. La cache est divisée en plusieurs segments LRU tels que les données y sont classées par valeur de ce rapport : un segment pour la valeur 1, un autre pour les valeurs de 2 à 3, un autre pour les valeurs de 4 à 7, ... Si une donnée accumule suffisamment d'accès, elle peut passer au segment supérieur (et passer ainsi plus de temps dans la cache). Notons que plus la taille d'un objet est grande et plus il lui sera difficile de passer dans les segments supérieurs. Les données qui arrivent au bout d'un des segments LRU sont retirées de la cache (et ne passent donc pas au segment inférieur). De plus, les données qui ont été retirées de la cache redémarrent au segment minimum en cas de réinsertion.

## 5.2 Politiques basées sur l'environnement d'utilisation

Les politiques de remplacement basées sur l'environnement d'utilisation sélectionnent les données à retirer de la cache en se basant sur des informations qui sont fournies par l'utilisateur (par utilisateur, nous entendons ici le programme qui utilise la cache). Ce type de politique se base sur l'idée que les programmes manipulant des flux de données connaissent un certain nombre de propriétés particulières associées à leur flux de donnée qui leur permettent de mieux prédire la probabilité qu'une donnée soit accédée dans un futur proche.

Ce type de politique est surtout réservée aux implémentations génériques de cache au niveau du système d'exploitation ou du moteur d'une base de donnée par exemple.

La politique du *Application-Controlled File Caching* (ACFC) présentée par P.Cao et al. dans [6] a été spécialement développée pour être utilisée par le noyau de *Linux* comme politique de remplacement de la mémoire cache des accès disques. Elle fournit une politique standard par défaut à toutes les applications (ce qui permet d'éviter toute modification des applications existantes). Cependant, chaque application peut, à loisir, modifier en partie la politique de remplacement pour obtenir de meilleures performances.

Une application ne pourra pourtant pas modifier la politique comme elle le désire car il ne faut pas oublier que la mémoire cache d'accès au disque est partagée par plusieurs applications et il ne faut pas qu'une application ait la possibilité de modifier (par effet de bord) les performances d'une autre application.

Il existe d'autres politiques de ce type mais elles sont au delà de la portée de ce document (elles ne pourront en effet pas être utilisée pour résoudre la problématique présentée à la première partie). Cette section n'a d'ailleurs été incluse que par souci de complétude.

## 5.3 Politiques basées sur des régularités

Les politiques basées sur des régularités tiennent compte des spécificités des programmes utilisant la mémoire cache. Bien qu'elles aient l'air semblables, il ne faut pas confondre cette catégorie avec celle de la section précédente : ici, il s'agit de principes généraux sur les programmes utilisant des caches comme la prise en compte des boucles ou des accès séquentiels par exemple.

Jong Min Kimy et al. [14] ont développé une politique qui exploite ce type de régularités : la politique du *Unified Buffer Management* (UBM). Cet algorithme va détecter les accès qui ont lieu lors de l'exécution d'une boucle ou bien lors du parcours séquentiel d'un fichier et utilise une partie réservée de la cache spécialement pour les données accédées dans ce cadre. De cette manière, le reste de la cache n'est pas *pollué* par ces données.

Pour commencer, un module de détection, classe les données voulant entrer dans la cache en trois groupes : à accès séquentiel pour les données consécutives qui sont accédées l'une après l'autre, à accès en boucle pour les données qui sont accédées plusieurs fois à intervalles réguliers et le reste des données dans le dernier groupe.

A chacun de ces trois groupes est attribuée une politique de remplacement particulière : le premier groupe utilise la politique MRU, le deuxième groupe une politique qui choisit les données dont l'intervalle d'accès est le plus long et enfin le troisième groupe utilise une politique de type LRU (ou LRU-K, LRFU selon la configuration).

La taille de ces trois groupes dans la mémoire cache varie dynamiquement en fonction de l'utilisation. Un algorithme spécifique choisit d'allouer ou non de la place dans un des groupes en se basant sur le gain marginal de *hit rate* que l'introduction d'une donnée *supplémentaire* d'un type particulier pourrait rapporter par rapport à un autre type[6].

Theodore Johnson et Dennis Shasha [13] proposent la politique *2 Queues* (2Q). Au lieu d'éliminer le plus rapidement possible les données non pertinentes de la cache, elle va tenter d'empêcher ces données d'entrer dans la cache. Pour cela, la mémoire cache est divisée en trois sections : $A1in$, $A1out$ et $Am$. Lorsqu'une donnée est placée en cache, elle commence par entrer dans la section $A1in$ qui est en fait une file *FIFO*. Quand elle quitte cette file, la donnée est retirée de la cache mais son adresse est mémorisée dans la section $A1out$ qui est également une file *FIFO*. Cette division de $A1$ permet d'allonger la longueur de la file sans consommer trop de mémoire et laisser ainsi plus de temps à une donnée de prouver qu'elle est pertinente.

Lorsqu'un accès sur une donnée présente dans $A1$ a lieu, elle est placée dans la section $Am$ qui est organisée en LRU et représente en fait la partie principale de la cache. La taille des différentes sections est un paramètre de configuration dont les auteurs recommandent les valeurs suivantes : $A1in$ devrait faire 25% de la taille totale de la cache et $A1out$ devrait pouvoir contenir les adresses de 50% des données pouvant être stockées dans la cache.

D'autres politiques de ce type ont été développées comme par exemple les politiques *Early*

[6] Le lecteur pourra consulter [14, page 6] s'il désire voir en détail comment ce gain marginal est estimé

*Eviction LRU* (EELRU) et SEQ citée par [14]. La politique SEQ tente de détecter les données faisant l'objet d'un accès séquentiel et leur applique la politique MRU. Malheureusement, cette politique ne tient pas compte des boucles.

La politique EELRU utilise l'algorithme LRU standard mais, s'il détecte que de trop nombreuses données récemment accédées sont retirées de la cache (par exemple en cas d'accès séquentiel ou de boucles), il court-circuite la politique LRU pour passer temporairement sur un algorithme basé sur des prédictions mathématiques qui, bien que moins performant, évite cet inconvénient de la politique LRU.

## 5.4 Politiques dynamiques/combinées

Plutôt que de tenter de développer une nouvelle politique de remplacement, les politiques de la catégorie dynamiques/combinées vont combiner l'utilisation d'un grand nombre de politiques que nous avons présentées au cours des sections précédentes par exemple. Au lieu de tenter de résumer les caractéristiques de toutes ces politiques dans une sorte de super politique, elles vont utiliser un moteur de décision qui va dynamiquement décider d'appliquer telle ou telle technique en fonction des caractéristiques des données à stocker dans la cache.

Ari et al [4] ont développé une politique fondée sur ce principe appelée *Adaptive Cache using Multiple Expert* (ACME). Cet algorithme utilise un ensemble de politiques de remplacement statiques. A chaque fois qu'une donnée doit être remplacée, l'algorithme demande à chaque politique de voter pour savoir quelle donnée sera évincée. Pour permettre cela, une *cache virtuelle* est assignée à chaque politique. Celle-ci ne contient aucune donnée mais uniquement des informations de référence de la donnée (et prend donc peu de place mémoire).

Au départ, le vote de chaque politique est équivalent mais, au fur et à mesure du temps, le nombre de voix attribué à chaque politique va évoluer en fonction de ses performances. A cette fin, l'algorithme mesure en permanence le *hit rate* de toutes les *caches virtuelles.*

Au final, la cache physique[7] contient des données dont l'organisation ne correspond à aucune politique en particulier (sauf dans le cas où les propriétés des données donneraient un poids très important à une politique qui leur serait particulièrement adaptée).

Selon le même ordre d'idées, Ganesh Santhanakrishnan et al. [20] ont développé une politique combinée basée sur un type très particulier de politiques que nous n'avons pas présentées ici, les politiques appelées *Greedy Dual-size* qui sélectionnent les données à remplacer en se basant sur le coût de récupération des données quand elles ne se trouvent pas dans la cache (c'est un type de politique qui convient parfaitement aux applications fonctionnant à travers l'Internet).

Cette politique s'appelle *A Goal-Oriented Self-Tuning Caching Algorithm* (GD-GhOST). Elle se base sur trois politiques *Greedy-Dual size* dont nous n'exposerons ici que le point commun :

[7] Le terme *physique* est ici utilisé par contraste avec le terme *virtuel* et ne spécifie absolument pas que ce type de mémoire cache doit être réalisée au moyen d'une mémoire physique dédiée.

toutes les politiques de ce type assignent à chaque donnée une valeur (appelée $H$) représentant le coût de récupération de cette donnée et sélectionne celle dont la valeur de $H$ est la plus faible pour être remplacée. Cette valeur est calculée de la façon suivante : $H = \frac{coût(donnée)}{taille(donnée)}$ où le coût de récupération est estimé de façon différente selon l'algorithme utilisé.

La politique GD-GhOST va combiner ces trois valeurs de $H$ pour construire sa propre valeur de $H$ et sélectionner la donnée à évincer. Tout comme la politique ACME, elle commence par offrir le même poids à chacun des trois algorithme et va ensuite le faire évoluer en fonction des performances. La différence est que cette évolution n'a pas lieu en temps réel mais à intervalles réguliers (qui sont déterminés automatiquement par l'algorithme en fonction des performances).

Voilà qui conclut notre tour d'horizon des différentes techniques de caching existantes ainsi que les points clés à prendre en compte lors de l'élaboration d'une mémoire cache. Dans la partie suivante, nous allons nous attacher à tenter de développer une solution pour répondre à la problématique présentée dans la partie précédente. Nous informons cependant le lecteur que, pour des raisons de temps et de coût, les techniques les plus évoluées présentées dans ce chapitre n'ont malheureusement pas pu être utilisées pour répondre à la problématique bien qu'elles nous semblaient prometteuses en théorie.

# Troisième partie

# Solution

# Chapitre 6

# Construction de la solution

Dans cette partie, nous allons tenter de répondre à la problématique présentée dans le chapitre 3. Ce chapitre va tout d'abord présenter la démarche qui a été suivie pour répondre à cette problématique. Nous allons détailler tous les aspects de la découpe du problème, étape par étape. Chacune des sections suivantes décrit donc un aspect particulier du problème et expose la réflexion qui a prévalu à sa prise en compte dans la solution finale. Le niveau d'abstraction de ce chapitre est très élevé (sans pour autant oublier le fait que l'implémentation doit être réalisée sur la *Proxy Platform*). Dans le chapitre suivant, nous exposerons les spécifications techniques du problème ainsi que les principaux algorithmes utilisés pour finir par les points clés de l'implémentation.

## 6.1 La politique de placement

Comme nous l'avons brièvement expliqué à la fin du chapitre 3, l'idée retenue pour optimiser le *Nextenso MMS Proxy Relay* est d'utiliser une cache pour éviter de recourir plusieurs fois à des traitements similaires sur les mêmes messages[1]. Cependant, comme nous l'avons montré, l'utilisation d'une mémoire cache ne peut profiter qu'à certains messages. Le problème suivant se pose alors : comment choisir ces messages ? Nous avons démontré, par exemple, que l'envoi de messages à destination de plusieurs utilisateurs pourrait être optimisé par une cache. Mais peut être en existe-t-il d'autres ? Et si non, il pourrait tout de même être intéressant d'affiner quelque peu notre moyen de sélection.

Pour répondre à ces différentes interrogations, et dans un souci d'un développement ouvert sur l'avenir, facilement modifiable, nous avons décidé de développer un langage permettant de définir, par configuration, des règles de sélection. L'idée est la suivante : chaque message qui transite par le *Nextenso MMS Proxy Relay* est comparé aux différentes règles définies. S'il correspond à au moins une de ces règles, il sera inséré dans la mémoire cache.

Etant donné que le but est de stocker les messages dont le contenu a déjà été adapté et qui ont

[1] Pour rappel, nous utilisons indifféremment la dénomination *message* et *MMS* tout au long de ce document.

déjà été encodés, chaque message sera comparé aux règles au moment de son téléchargement par un des destinataires.

La politique de placement que nous utilisons pour notre cache est donc une politique de placement *sélective* par accès. Les *cache hits/misses* auront lieu au moment de la demande de téléchargement du MMS par un destinataire. Elle est sélective au sens où l'insertion d'un message lors d'un *cache miss* dépendra du résultat de l'application des règles.

La définition (syntaxique et sémantique) de ce langage pourra être trouvée dans l'annexe B. Afin de permettre une configuration flexible, plusieurs variables, qui seront évaluées au moment de la comparaison d'un message avec une règle, peuvent être utilisés : la taille du message, le nombre de destinataires *locaux*[2] du message, le nombre de messages actuellement dans la cache, le taux d'occupation de la cache (du point de vue de la taille), la date d'expiration du message ainsi que le canal dont il est issu.

L'utilisation de la variable qui compte uniquement le nombre de destinataires locaux (c'est-à-dire qui sont clients chez l'opérateur utilisant le *Nextenso MMS Proxy Relay*) se justifie par le fait que les messages à destination d'un autre opérateur sont immédiatement envoyé vers le *MMS Relay* de cet opérateur. Il est évident que seuls les messages à destination de clients locaux seront téléchargés via ce *Nextenso MMS Proxy Relay*.

Le lecteur attentif remarquera que les règles permettent de définir l'une des deux actions suivantes : *CACHE* et *PRECACHE*. La première correspond à l'insertion du message dans la cache au moment du téléchargement telle que nous venons de la décrire alors que la seconde permet de *pré-charger* la cache avec un message non-encodé. A la fin du développement, il s'est avéré que ce dernier mécanisme ne présentait qu'un faible intérêt pratique. Compte tenu de sa complexité, nous avons choisi de ne pas le présenter dans ce document bien qu'il ait été implémenté dans la solution finale.

## 6.2 Le stockage des MMS dans la cache

Maintenant que nous savons comment choisir les MMS à mettre dans la cache, il faut savoir de quelle manière nous allons les stocker. La première idée est de stocker les messages encodés dans la cache. Cependant, comme nous l'avons vu dans le chapitre 3, il sera nécessaire d'adapter le contenu ainsi que ré-encoder un même message une fois pour chaque groupe de destinataires utilisant le même type de terminal.

Afin d'accélérer ce processus, il est intéressant de stocker une copie du message non-encodé dans la cache et ainsi éviter de devoir le récupérer au travers du canal *MM2* (voir figure 1.3 page 25) à chaque fois qu'un nouveau type d'adaptation est nécessaire.

La solution que nous avons choisie est de stocker les messages de manière hiérarchique : toutes

[2]Notons que, dans la suite de ce document, toute allusion aux destinataires d'un MMS fera référence aux destinataires *locaux* de ce MMS.

les copies du même message (adapté à un type de terminal particulier) présentes dans la cache sont rattachées à une copie non-encodée de celui-ci appelée le *parent*.

Intéressons nous maintenant au problème de l'adressage des messages. Comme nous l'avons vu dans le chapitre 1, le téléchargement des MMS est effectué au moyen d'une URL particulière. Cette URL peut donc nous servir à identifier une *famille* (au sens hiérarchique) de message, plus exactement le parent de cette famille. Pour choisir entre les versions encodées du message, nous allons également utiliser une URL, celle du *UAProf* du type de terminal associé. Elle sera de plus facile à récupérer car elle est passée en paramètre de toute communication entre un terminal et le *MMS Relay* (voir chapitre 1).

Notons que l'adressage par URL (pour le parent) ne suffit pas. En effet, le protocole du canal *MM7*, notamment, utilise le paramètre *Message ID* pour identifier le message sur lequel doit être exécuté une action. Ce paramètre fait partie du protocole MMS (qui n'a pas été présenté techniquement dans ce document) et est utilisé dans beaucoup de situations : c'est en effet l'identifiant principal d'un MMS. Sa valeur est fixée par le premier *MMS Relay* par lequel passe le message, la contrainte étant que cette valeur soit unique pour tous les messages empruntant ce *MMS Relay*.

Les deux actions spéciales du protocole *MM7* dont nous devons tenir compte (*MM7-Cancel* et *MM7-Replace*) dans le cadre de la cache nous obligent à utiliser également le *Message ID* comme second identifiant d'une *famille* de messages.

Remarquons que le stockage dans la cache ne doit pas être "sûr", au sens de la tolérance aux pertes de données. En effet, l'utilisation de la cache ne se substitue pas à la méthode habituelle de stockage des MMS (via le canal *MM2*) qui, elle, est résistante aux pertes de données. Il ne serait bien sûr pas admissible de concevoir un système pouvant "égarer" un message pour lequel le client a été facturé. Ainsi, au moindre problème au niveau de la cache, le système se retournera vers le système de stockage normal. L'avantage est de pouvoir utiliser des techniques de stockage plus performantes (en terme de temps d'accès) au niveau de la cache.

## 6.3 La gestion des instances

Les accès aux MMS contenus dans la cache se font au niveau des *HTTP Agents* du *Nextenso MMS Proxy Relay*. Nous avons vu, au chapitre 2, que ces *HTTP Agents* sont indépendants et peuvent même être déployés sur des machines différentes. La mémoire cache ne peut être située au niveau d'un seul *HTTP Agent* car elle ne pourrait alors pas être utilisée par les autres.

De plus, la répartition automatique de la charge de travail entre les différents *HTTP Agents* implique que ceux-ci ne peuvent utiliser chacun leur propre cache car il se peut qu'un message qui serait stocké dans la mémoire cache d'un *HTTP Agent* particulier devienne, plus tard, nécessaire dans un autre. L'avantage de la cache serait alors fortement diminué (voir nul si le nombre de *HTTP Agents* est important) tout en consommant beaucoup plus de mémoire.

Une solution aurait été d'utiliser un système de mémoire distribuée entre les différents *HTTP*

*Agents*. Ce mécanisme est malheureusement complexe à mettre en oeuvre et pas aussi performant qu'une mémoire centralisée (tant au niveau des temps d'accès que du trafic réseau supplémentaire généré). C'est pourquoi nous avons choisi d'utiliser une mémoire cache centralisée dans un module séparé avec lequel les différents *HTTP Agents* peuvent communiquer à travers le réseau.

De cette manière, la propriété *scalable* du système est conservée. Notons que la solution qui sera implémentée place une limite supérieure à cette propriété au niveau de l'utilisation de la cache. Cette difficulté pourrait être contournée en limitant l'utilisation de la cache ou bien en développant tout de même une mémoire distribuée spécialisée pour implémenter la mémoire cache, qui apparaîtrait toujours centralisée du point de vue des *HTTP Agents*.

Afin de permettre cette communication, nous avons développé un protocole de communication réseau destiné à être utilisé entre la cache et ses clients. La spécification complète de ce protocole pourra être trouvée dans l'annexe C.

## 6.4 La politique de remplacement

Comme nous l'avons vu dans le chapitre 4, la politique de remplacement est le paramètre qui influencera le plus les performances de la cache. Malheureusement, la technologie du MMS est relativement récente et nous n'avons pu trouver aucune source de statistiques d'utilisations à son propos. De même, il faut considérer que les utilisateurs de ce type de données sont des humains et non des programmes. Nous ne disposons par conséquent d'aucune base solide pour conduire une réflexion sur la meilleure politique à adopter.

Cependant, nous possédons une information très importante à propos des MMS : le nombre de destinataires. Nous savons, qu'après qu'un MMS ait été téléchargé par tous ses destinataires, il ne sera plus jamais accédé. Cela nous permet, de pouvoir décider avec certitude qu'un tel MMS n'est plus pertinent dans la cache et doit en être retiré (c'est d'ailleurs le comportement qui prévaut au niveau du stockage proprement dit du MMS au niveau du *MMS Server*).

Pour conduire notre réflexion, nous avons décidé de nous baser sur les algorithmes de mémoire cache qui sont utilisés sur l'Internet. Car les MMS sont tous de tailles différentes et le domaine qui utilise le plus des algorithmes de cache où les données sont à tailles variables est celui de l'Internet.

J. E. Pitkow et M. M. Recker [19] ont développé un modèle permettant de prévoir les accès aux pages d'un site Internet en se basant sur des travaux en psychologie portant sur la manière dont la mémoire humaine ordonne et retient des informations. Ils ont ensuite confronté les prévisions de ce modèle aux statistiques d'utilisation d'une palette de sites Internet sur une durée de trois mois et ont obtenu des similitudes très intéressantes. De ce modèle, ils ont tiré une ébauche de politique de remplacement, basée sur la fréquence et le moment des accès aux pages.

En nous basant sur leurs résultats et sur la théorie présentée dans le chapitre 5, nous avons élaboré une politique particulière tenant compte de la fréquence (en faisant un rapport entre la fréquence des accès et le nombre total de destinataires du message) et le moment du dernier accès.

Le principe de cette politique est le suivant : tout comme dans le modèle de Pitkow et Recker, le taux d'occupation de la cache sera maintenu (dans la mesure du possible) en dessous d'une certaine valeur, fixée par configuration (typiquement, 80%). De cette manière, le temps d'insertion d'un élément reste très faible car, même en cas de forte charge, le pourcentage de place libre dans la cache devrait offrir une marge suffisante. Ce choix se justifie également par le fait que l'on peut utiliser une quantité relativement importante de mémoire pour réaliser la cache ; la place n'est donc pas un facteur aussi critique que le temps. A chaque message présent dans la cache, on associe une valeur, appelée *score* qui évalue la pertinence du message à se trouver dans la cache.

Cette valeur se calcule de la façon suivante :

```
if local_rcpts - activity > 25% of local_rcpts
    then score <- activity
    else score <- local_rcpts - activity

if last_access between 10 min and 1 hour
    then score <- score * 2
if last_access between 1 min and 10 min
    then score <- score * 4
if last_access between 20 seconds and 1 min
    then score <- score * 8
if last_access between 0 and 20 seconds
    then score <- score * 16

if score < 0
    then score <- 0
```

où `local_rcpts` représente le nombre de destinataires du MMS, `last_access` le moment du dernier accès à ce MMS et `activity` le nombre de fois où ce MMS a été accédé depuis son insertion dans la cache.

Malgré le fait que nous ne disposions d'aucune statistique sur le MMS, nous avons supposé, pour mettre au point le mode de calcul du *score*, qu'il était raisonnable de penser que le temps que mettrait un destinataire pour demander le téléchargement d'un message ne devrait, en moyenne, pas excéder l'ordre d'une heure.

Périodiquement (cet intervalle étant un paramètre de configuration), un algorithme de type *garbage collector* calcule le *score* de chaque message présent dans la cache et élimine les éléments indésirables ainsi que, si le taux d'occupation de la cache est trop important, tous les messages dont le *score* est inférieur de 25% à la moyenne des *scores* des éléments de la cache. Sont considérés comme indésirables tous les messages sans parent (cette situation peut se produire à la suite d'une éviction de ce dernier par l'algorithme), les messages expirés (c'est-à-dire ceux que leurs destinataires ne sont pas venu télécharger avant que la durée de validité n'ait expiré) ainsi que les

messages que tous les destinataires ont déjà téléchargé.

Cette politique doit évidemment être testée en pratique, ne fût-ce que pour trouver les réglages optimaux des paramètres de configuration. Des tests en conditions réelles d'utilisation permettront de se faire une idée plus précise de ses performances.

# Chapitre 7

# La solution

Sur base des réflexions du chapitre précédent, nous exposons ici le détail des points importants de la solution finale : son architecture détaillée, les structures de données utilisées (au sens algorithmique) ainsi que son intégration dans l'architecture de la *Proxy Platform*.

## 7.1 Les nouvelles proxylets

La solution est incorporée, au niveau de l'*HTTP Agent* du *Nextenso MMS Proxy Relay*, au moyen de deux *proxylets*, appelées *Cache Req* et *Cache Resp* et placées sur le flux de requêtes et de réponses respectivement. Elle traitent le flux de données et communiquent avec la mémoire cache proprement dite au moyen de connections réseaux, comme nous le verrons dans la section suivante.

### 7.1.1 La *proxylet Cache Req*

La *proxylet Cache Req* a pour rôle de filtrer toutes les demandes de téléchargement d'un MMS. Lorsqu'une telle demande se produit, elle vérifie si ce message est présent dans la cache. S'il est présent, deux situations sont alors possibles : le MMS correspondant est présent dans la cache et encodé pour le type de terminal qui a émis la demande de téléchargement ou bien il s'y trouve mais pour un (ou plusieurs) autre(s) type(s) de terminal (terminaux).

Dans le premier cas, le MMS encodé est immédiatement envoyé au terminal qui a émis la demande. Dans le second, une copie du MMS original est extraite de la cache et placée au début du flux de réponse (pour que son contenu soit adapté et qu'il soit encodé). Si il satisfait aux règles, il sera ensuite inséré dans la cache avant son envoi définitif.

Notons que, dans les deux cas, la *proxylet* doit informer le *Server* qui stocke les MMS car c'est celui-ci qui prend en charge la suite de la transaction avec le terminal. Pour rappel, le terminal doit encore envoyer une confirmation sur le succès du téléchargement (voir chapitre 1). Il est donc

nécessaire que le *Server* soit informé qu'un téléchargement est en cours pour compenser le fait que le flux d'information n'aille pas jusqu'à lui dans cette situation. Il faut également que la *proxylet* génère les informations nécessaire au module de facturation.

### 7.1.2 La *proxylet Cache Resp*

La *proxylet Cache Resp* filtre les MMS envoyés en réponse à un téléchargement. Dans ce cas, elle compare le message à chacune des règles de sélection définies. Si aucune ne correspond, rien ne se passe. Dans le cas d'une correspondance, le message est placé dans la cache (si nécessaire, son parent est créé). Notons que, dans le cas improbable, qu'il n'y ait plus de place dans la cache, aucune erreur n'est générée. Dans tous les cas, le message poursuit sa route normalement vers le terminal du destinataire.

### 7.1.3 Cas d'utilisation

La figure 7.1 illustre un exemple d'utilisation de la cache. La phase I montre l'envoi initial du MMS destinés à plusieurs destinataires locaux (dans l'exemple, par un client de l'opérateur). La cache n'entre en jeu à aucun moment et tout se déroule donc comme d'habitude. La phase II décrit le *premier* téléchargement de ce même MMS par un destinataire. Juste avant d'être envoyé au terminal, il sera inséré dans la cache (la comparaison avec les règles ayant fonctionné). La phase III illustre le téléchargement du MMS par d'autres destinataires possédant le même type de terminal que celui utilisé lors de la phase II : le MMS encodé est extrait de la cache au niveau de la *proxylet Cache Req* et envoyé directement au destinataire. Notons qu'il existe une phase IV (non représentée sur la figure), qui consisterait en le téléchargement du MMS par des destinataires possédant un terminal différent : dans ce cas, comme expliqué plus haut, une copie du MMS original serait placée au début du flux de réponse.

## 7.2 Le module *Rules Matcher*

La nécessité d'avoir un algorithme d'application des règles aux messages extrêmement efficace s'est tout de suite démarquée car il ne faut en aucun cas que ce traitement ralentisse la progression à travers le *Nextenso MMS Proxy Relay* des messages qui ne seront pas amenés à être insérés dans la cache. En effet, il ne s'agit pas, afin d'optimiser certains messages, d'introduire un effet négatif sur les performances du traitement des autres messages.

Il faut également tenir compte du fait que les règles sont spécifiées au moyen d'un langage particulier qu'il faut "compiler" au moment de l'initialisation du module ainsi qu'à chaque changement des paramètres de configuration.

A cette fin, nous avons développé un algorithme qui transforme les règles dans un format comparable à la notation polonaise inversée pour les calculatrices. De cette manière, l'application

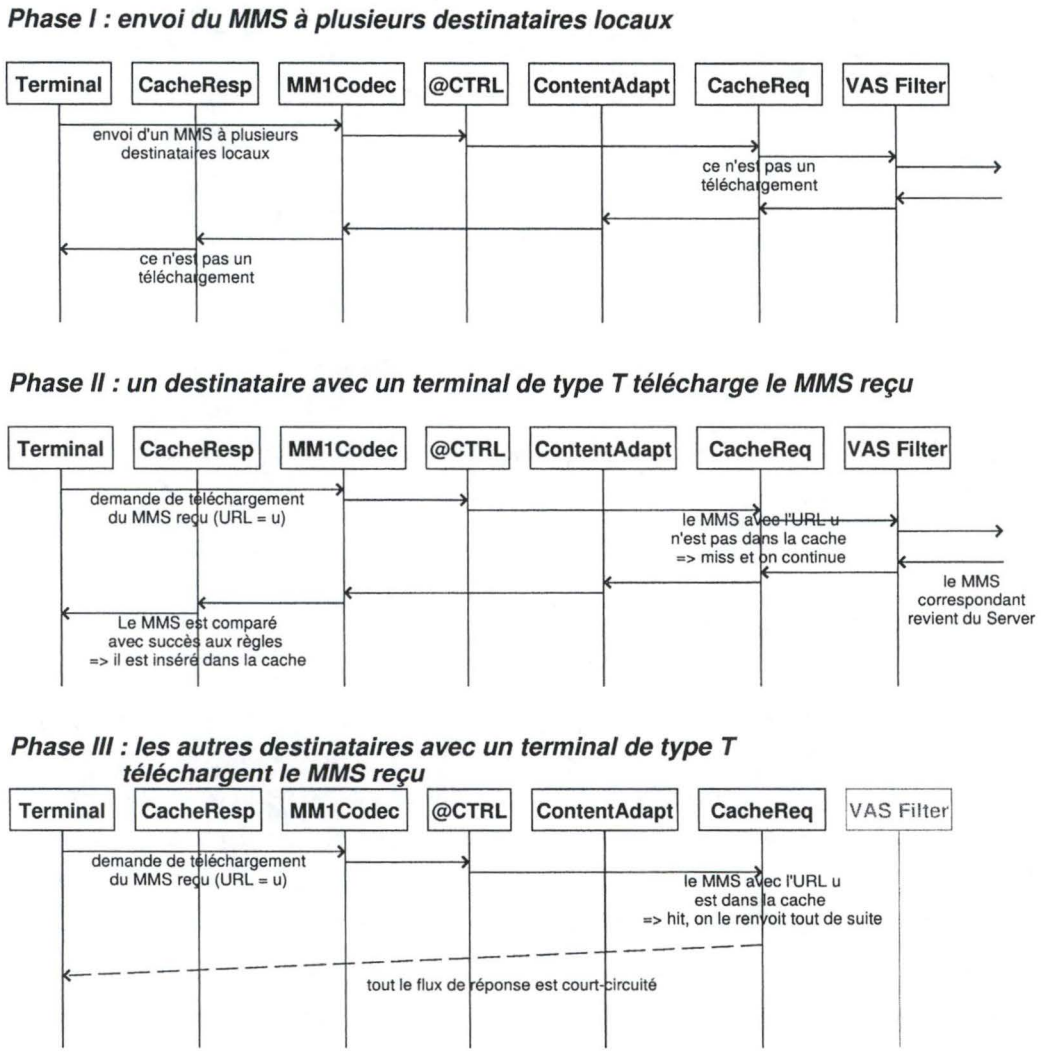



FIG. 7.1 – Exemple de cas d'utilisation mettant en oeuvre la cache

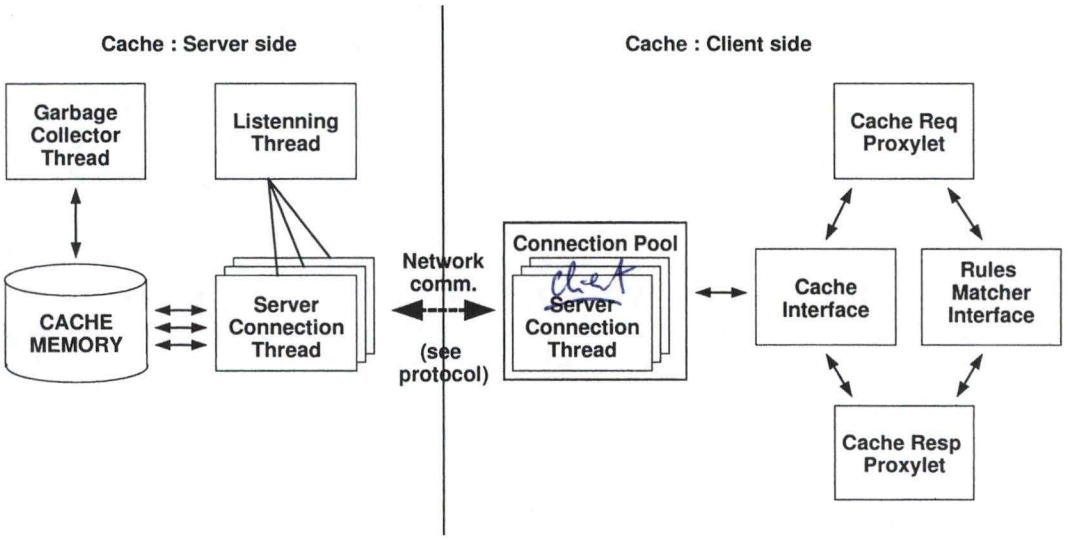



FIG. 7.2 – Architecture générale de la cache

des règles est effectuée au moyen d'une pile. Cette méthode permet l'application des règles en une seule passe et donc une complexité linéaire sur la longueur de la règle.

## 7.3 Architecture générale de la cache

Sur la figure 7.2, nous pouvons voir une vue d'ensemble de l'architecture de la cache. La partie serveur sera implémentée en *C* pour de meilleures performances. Une tâche est utilisée pour surveiller les connections en provenance des clients. Quand une nouvelle connection est ouverte par un client, une tâche dédiée est utilisée pour la servir. C'est elle qui écrira et lira directement dans la mémoire cache. Il faudra donc protéger l'écriture des données au moyen de verrous. Nous y reviendrons plus en détail dans la section 7.6.

Sur la partie droite, on peut voir l'architecture du client, qui est inclus dans l'*HTTP Agent* du *Nextenso MMS Proxy Relay*. Comme nous l'avons vu, le traitement de chaque MMS à l'intérieur de l'*HTTP Agent* est assuré par une tâche. Une seule connection ne peut donc suffire puisqu'il peut y avoir plusieurs messages traités en même temps. A cette fin, un ensemble de connections utilisables par chaque tâche est utilisé. Lorsqu'une tâche doit communiquer avec la cache, elle peut demander une de ces connections persistantes, qu'elle libérera après usage pour permettre aux autres tâches de s'en servir.

Afin d'augmenter l'évolutivité de la solution, nous avons choisi d'implémenter la communication avec la mémoire cache derrière une interface (au sens Java). Ainsi tout changement apporté au protocole de communication sera transparent pour les *proxylets* concernées (une version locale de la cache avait d'ailleurs été également développée sur cette interface). La même logique a été

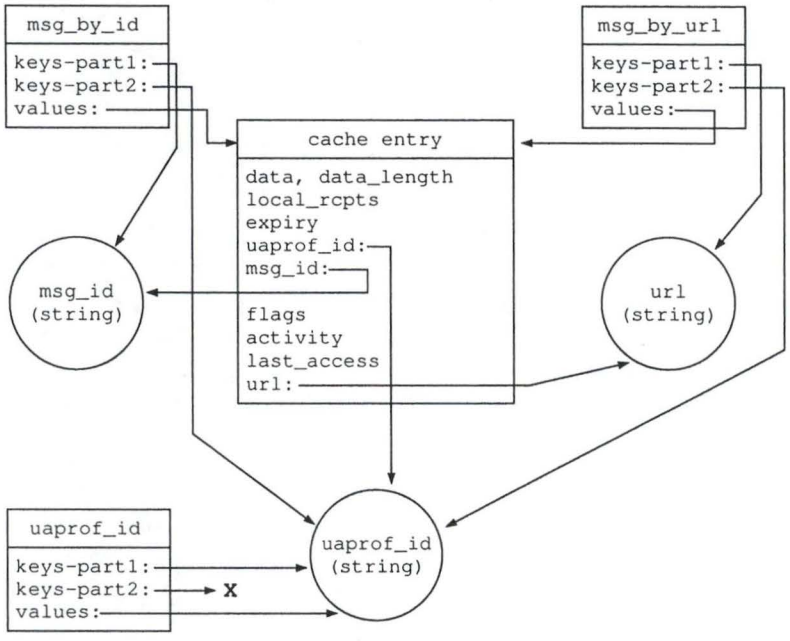



FIG. 7.3 – Structure de la mémoire cache

appliquée au module qui gère la comparaison des règles.

## 7.4 Structure de la mémoire cache

La figure 7.3 présente la manière dont vont être stockés les MMS dans la cache. Les deux tables principales, `msg_by_id` et `msg_by_url`, correspondent en fait aux index de la cache. Elles vont permettre d'accéder rapidement à un message s'il est présent ou répondre rapidement s'il ne l'est pas. Chaque MMS présent dans la cache est ainsi référencé dans *chacune* des deux tables.

Les identifiants (clés) de ces tables sont en fait composée de deux parties (`keys-part1` et `keys-part2`). La première fait référence à l'URL ou au *Message ID* tandis que la seconde partie fait référence au profil du terminal auquel correspond l'encodage du mémoire. Le seul élément d'une famille (ensemble de messages référencés par la même URL et le même *Message ID*) dont la seconde partie de la clé est nulle est la copie de l'original (donc non encodée) du message.

La troisième table d'index permet de retrouver rapidement la référence d'un profil de terminal particulier (*UAProf*). En effet, les clés sont formées de chaînes de caractères qui prennent beaucoup de place en mémoire. Il serait inutile de stocker plusieurs fois la même chaîne de caractères alors qu'il suffit de le faire une seule fois et de réutiliser plusieurs fois sa référence. Lorsqu'un message encodé est inséré dans la cache, l'algorithme tente tout d'abord de récupérer une référence au profil correspondant via cette table. S'il ne peut la trouver, il crée une nouvelle entrée

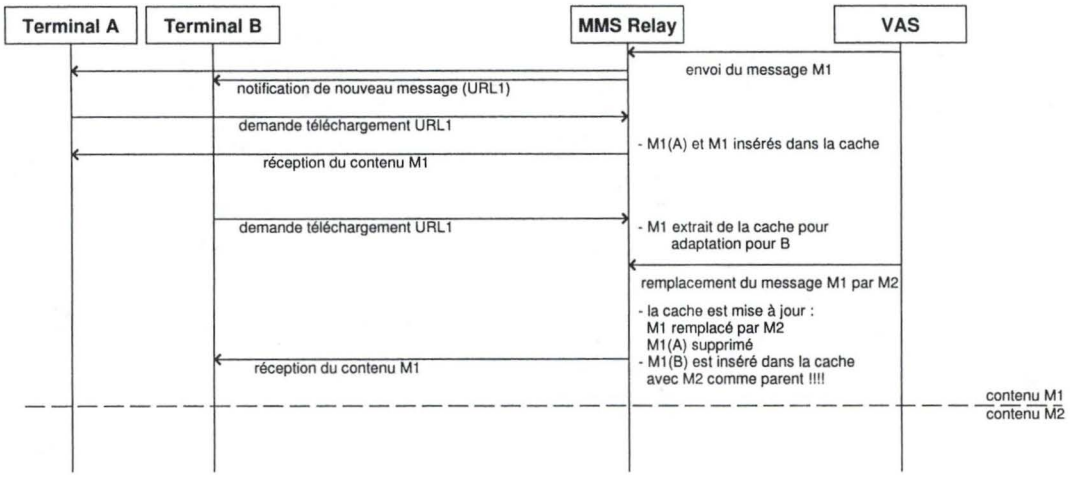



FIG. 7.4 – Illustration du problème *MM7-Replace*

contenant la chaîne identifiant ce terminal et utilise sa référence (qui pourra ainsi être réutilisée par d'autres messages).

Le même mécanisme a été utilisé, à plus petite échelle, pour les URL et les *Message ID* qui sont également des chaînes de caractères qui sont utilisées plusieurs fois dans les tables d'index ainsi que dans l'entrée de la cache contenant le message proprement dit.

La table `cache_entry` est divisée en deux sections : les informations statiques et les informations dynamiques. Les premières sont fixées au moment de l'insertion du message dans la cache et il n'est donc pas nécessaire d'employer un système de verrou pour y accéder. Il s'agit du message proprement dit (une suite d'octets), la longueur de ce dernier, la date d'expiration du message ainsi que le *Message ID* et la référence au type de terminal. La seconde contient les informations susceptibles d'être modifiées tout au long de la présence du message dans la cache (il faudra donc veiller à empêcher plusieurs tâches d'y accéder en même temps) : il s'agit des attributs (`flags` - leur signification sera abordée plus tard), l'activité (fréquence des accès), le moment du dernier accès et l'URL.

Notons que l'URL se trouve dans cette partie pour des raisons liées à l'implémentation des règles de *pre-caching* qui n'est pas exposée dans ce document (voir chapitre 6).

## 7.5 Le cas du *MM7-Replace*

Le *MM7-Replace*, action spécifique au protocole du canal *MM7* pose un problème à la gestion de la cache. Via celle-ci, un *VAS* peut remplacer le contenu d'un message qu'il aurait envoyé à plusieurs destinataires. Le *MMS Relay* doit, dans ce cas, remplacer l'ancien message par le nouveau pour tous les destinataires qui n'ont pas encore téléchargé ce message. Cela implique une

gestion particulière de ce type de cas : tous les messages encodés de la famille correspondante sont supprimés de la cache et le contenu du parent est remplacé par le nouveau.

Il existe cependant un cas non trivial (présenté sur la figure 7.4). Supposons qu'un VAS envoit un MMS, $M_1$, à plusieurs destinataires. Un premier destinataire vient télécharger le message. Ce dernier, satisfaisant aux règles de sélections, est alors inséré dans la cache. Un autre destinataire, possédant un terminal de type différent ($B$), vient ensuite télécharger le message. Comme une version encodée pour ce terminal n'est pas présente en cache, c'est la copie d'origine qui est placée au début du flux de réponse. A ce moment précis, le *VAS* décide de remplacer le message $M_1$ par le message $M_2$. La famille de ce message est donc épurée de tous ces éléments encodés et le contenu du parent remplacé pendant le temps nécessaire à l'adaptation et à l'encodage du message $M_1$ que le deuxième destinataire est venu récupérer. Ce message ($M_1$) encodé, possédant le même *Message ID* que le message $M_2$, sera inséré dans la cache avec comme parent $M_2$.

A partir de cet instant, tous les destinataires utilisant un terminal de type $B$ risquent de télécharger le message $M_1$ au lieu du message $M_2$ qui vient d'être remplacé.

Ce problème est extrêmement complexe car, au delà de la présentation qui en est faite ici, il n'existe aucune garantie sur la sérialisation de ces différentes transactions (téléchargement et remplacement), ce qui complexifie encore d'avantage le problème. Au moment de la rédaction de ce document, nous n'avons malheureusement toujours trouvé aucune solution *efficace* et *complète* à ce problème. Cette question reste donc en suspend.

## 7.6 L'implémentation

L'implémentation de la partie serveur de la cache (et donc de la mémoire cache proprement dite) est réalisée au moyen d'un *Monitorable Server* et est donc écrite en *C*. C'est de cette partie de l'implémentation que traitera cette section. Il ne s'agit en aucun cas de présenter tous les détails de l'implémentation mais simplement de commenter les solutions originales qui ont été développées dans le cadre de cette implémentation. Essentiellement, nous nous bornerons à parler du stockage des messages, le reste de l'implémentation ne présentant pas de difficulté particulière.

Les différentes tables d'index sont réalisées au moyen de grosses tables de *hashage* optimisées pour le stockage des chaînes de caractères. La fonction de *hashage* utilisée est fortement inspirée de celle utilisée en Java. Afin d'optimiser les accès concurrentiels, les tables sont protégées par un système de verrous lecture/écriture qui permet à plusieurs tâches de lire simultanément dans la table alors que seule l'écriture fait l'objet d'une zone d'exclusion.

Afin de réduire encore cette zone d'exclusion qui est critique pour les performances de la cache, chaque MMS stocké possède son propre verrou pour protéger ses données dynamiques. Lors d'un accès à un message, le verrou de la table d'index est libéré aussitôt que la référence au message est récupérée. La demande de verrouillage du message proprement dite ayant lieu après.

Cela pourrait induire un problème car le circuit d'attente des verrous utilisés n'est pas protégé

contre la destruction du message. Ainsi, si un message est détruit (retiré de la cache) pendant qu'une tâche est en attente, celle-ci subira une erreur de segmentation. Pour éviter ce problème, les messages qui doivent être détruit sont marqués comme inutilisables (au moyen du paramètre `flags`) et les références supprimées des tables d'index. La tâche qui tente d'effacer le message va ensuite libérer son verrou et en refaire une demande. Etant donné la manière dont les verrous sont implémentés, elle ne récupérera le verrou que lorsque toutes les autres tâches en attente sur celui-ci l'auront obtenu. A ce moment, elle pourra détruire sans risque le message (les références ayant été supprimées, il ne peut exister de tâche encore en attente sur ce verrou).

Notons qu'une tâche qui obtiendrait un verrou sur un message marqué comme étant supprimé considérera ce message comme inexistant et relâchera immédiatement le verrou.

En ce qui concerne l'implémentation du *garbage collector*, notons qu'un système spécifique a été introduit dans les tables d'index afin de permettre un parcours séquentiel efficace des messages présents dans la cache. A cette fin, un curseur se déplace d'entrée en entrée dans la cache. Si un message est inséré *avant* la position courante du curseur, ce message ne sera traité qu'à la prochaine boucle du curseur.

Pour éviter de surcharger le système avec des calculs inutiles, le parcours des différents messages contenus dans la cache n'a lieu que lorsque le taux d'occupation devient critique par rapport à la valeur maximum fixée par configuration. Un passage forcé est néanmoins exécuté périodiquement pour permettre de tenir à jour la valeur moyenne de *score* de la cache.

## 7.7 Les tests

Outre les tests unitaires qui ont été réalisés pour vérifier la correction de la solution, il aurait fallu également réaliser des tests de charges pour comparer les bénéfices apportés par la solution implémentée.

Malheureusement, il ne nous a pas été possible de mener ces tests à bien durant la période pendant laquelle les différents outils concernés étaient à notre disposition. De plus, pour simuler des tests suffisamment réalistes, il nous aurait fallu développer un logiciel spécialisé. Nous ne pourrons par conséquent pas présenter de tels résultats dans le présent document.

Cependant, des tests préliminaires réalisés pendant les dernières phases de développement ont montré que l'utilisation de la cache ne dégradait pas les performances en charge des messages qui ne satisfont pas aux règles de sélection, même si aucun message ne satisfait à ces règles. Cela signifie que, au pire, le système ne sera pas moins performant qu'avant. C'était un point très important qui a été pris en compte lors du développement, d'où l'intérêt particulier qui a été porté à la réalisation du module *Rules Matcher*.

Même en l'absence de chiffres, il est possible de raisonner sur les performances qui pourraient être atteintes. Premièrement, il découle du raisonnement présenté dans le chapitre 3 que la totalité du temps processeur qui sera utilisé pour traiter les téléchargements d'un message destiné à plu-

sieurs destinataires sera plus faible s'il transite par la cache dans des circonstances optimales (en supposant raisonnablement que l'implémentation est performante). Ensuite, même si un tel message ne transite pas par la cache, le surcoût occasionné par les traitements de sélection au niveau de la cache est nul. Donc, dans le pire des cas, les performances du système pourvu de la cache sont équivalentes à celles du système dépourvu de cache. Par conséquent, il est raisonnable de penser que les performances avec utilisation de la cache augmenteront d'un facteur dépendant du nombre de messages satisfaisant aux règles de sélection (pour peu que ces dernières soient judicieusement choisies).

La quantification exacte de cette augmentation ne pourra être effectuée que par la mise en place de tests sur des échantillons réels de messages avec simulation du comportement des utilisateurs.

Cependant, nous pouvons tout de même présenter les grandes lignes d'un protocole de test par simulation qui pourrait être utilisé pour évaluer les gains de performance apporté par nos optimisations. Il s'agirait de recréer, à petite échelle (un millier d'individus) et de manière aussi réaliste que possible, un réseau d'utilisateurs s'échangeant des messages de manière aléatoire.

Une base de données de MMS types pourrait être construite de manière automatique au moyen d'un ensemble de fichiers textes, images, sons et vidéos, la taille des messages générés s'étendant de quelques octets à 50 kilo-octets. La distribution des différentes tailles de messages est difficile à évaluer par manque de sources statistiques. Il faudra donc supposer, par exemple, qu'il est plus réaliste de générer plus de messages de petites tailles que de grandes tailles.

Un logiciel simulant un millier d'utilisateurs possédant chacun un terminal choisi aléatoirement parmi un ensemble d'une centaine d'éléments se chargerait alors d'effectuer des envois de MMS choisis au hasard dans la base de données à un nombre aléatoire de destinataires du réseau. Ici aussi, il parait plus intuitif de supposer qu'il faudrait envoyer plus de messages vers un petit nombre de destinataires ou un seul destinataire bien que nous n'ayons pu trouver de source statistique sur ce sujet. Il devrait également s'occuper d'effectuer les téléchargements des messages reçu. Pour tenter d'introduire le facteur humain, il faudrait utiliser un délai aléatoire entre le moment de la réception de la notification et la demande de téléchargement proprement dite.

Il suffirait alors de laisser tourner ce test plusieurs heures avec et sans la présence de la cache pour pouvoir avoir une estimation relativement réaliste des gains de performances qui peuvent être espérés dans des conditions réelles d'utilisation.

Cependant, comme nous l'avons mentionné plus haut, la réalisation de tests de cette envergure risque de nécessiter le développement d'outils spécialisés.

# Conclusion

Nous avons tenté de sensibiliser le lecteur sur les enjeux de l'optimisation de l'organe central de la technologie MMS : le *MMS Relay*. Ceux-ci sont d'ordre économique car plus une solution logicielle est optimisée et moins la structure matérielle à mettre en place pour la supporter est onéreuse. Au moment de choisir un produit entre différents concurrents, un opérateur prendra cet aspect financier très au sérieux.

Au cours d'un stage réalisé auprès de la société *Nextenso SA*, nous avons été amené à réaliser des optimisations sur le *Nextenso MMS Proxy Relay*, une implémentation de *MMS Relay* basée sur la technologie *Nextenso Proxy Platform* également développée par cette société.

En nourrissant notre réflexion de certaines propriétés des MMS ainsi que de cas d'utilisation pratiques, nous avons développé une solution d'optimisation utilisant les avantages de l'architecture particulière de la *Nextenso Proxy Platform*. Cette solution se base sur l'utilisation d'une mémoire cache.

Après avoir parcouru de nombreuses recherches en la matière, nous avons tenté de dériver une politique de gestion de la cache efficace dans le contexte du MMS. Nous avons également développé un langage de définition de règles permettant de sélectionner avec précision les MMS qui peuvent profiter des avantages de la cache. Ceci rend notre solution très adaptable et configurable. Une mini base de donnée a également été imaginée afin de permettre un accès efficace aux MMS stockés dans la mémoire cache.

Malheureusement, nous n'avons pas été en mesure de réaliser des tests comparatifs pour mesurer l'impact réel de notre optimisation. Il pourrait être intéressant de réaliser les tests que nous proposons à la fin du chapitre 7 comme approfondissement de ce travail. De plus, si les résultats n'étaient pas ceux escomptés (bien qu'il soit fort probable qu'ils soient positifs, le pourcentage de variation des performances reste incertain), ils pourraient servir de base pour améliorer la solution.

Remarquons également que nous avons délibérément privilégié la piste de la mémoire cache. Ce type d'optimisation permet d'augmenter les performances sans devoir modifier le code existant. Toujours dans une optique d'amélioration de ces performances, il pourrait donc être intéressant d'étudier la possibilité d'optimiser l'implémentation existante.

Pour conclure, nous pouvons dire que, bien que les machines actuelles deviennent de plus en plus puissantes, l'optimisation reste un enjeu important des systèmes devant supporter de fortes

charges de travail.

# Bibliographie


[1] 3rd Generation Partnership Project. 3gpp ts 23.140 v6.2.0. Technical report, 3GPP, 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE, Juin 2003. 3rd Generation Partnership Project ; Technical Specification Group Terminals ; Multimedia Messaging Service (MMS) ; Functional Description ; Stage 2 (Release 6).

[2] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transaction on Computer Systems*, 7(2) :184–215, May 1989.

[3] OMA Open Mobile Alliance. Mms conformance document v1.2. Technical report, OMA, 2002.

[4] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. Acme : Adaptive caching using multiple experts. In Carleton Scientific, editor, *Proceedings of the Workshop on Distributed Data and Structures (WDAS)*, 2002.

[5] Sorav Bansal and Dharmendra S. Modha. Car : Clock with adaptive replacement. In *Proceedings of the 3nd USENIX Symposium on File and Storage Technologies (FAST'04)*, pages 187–200, San Francisco, California, March 2004.

[6] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 171–182, 1994.

[7] Arun Chauhan, Henrik Weimer, and Shilpa Lawande. Quantitative comparison of set-associative and victim caches : Proposal. October 1989.

[8] Kai Cheng and Yahiko Kambayashi. Lru-sp : A size-adjusted and popularity-aware lru replacement algorithm for web caching. In *Proceedings of the 24th IEEE Computer Society International Computer Software and Applications Conference (Compsac'2000)*, 2000.

[9] Zhifeng Cheny, Yuanyuan Zhouy, and Kai Li. Eviction based cache placement for storage caches. In *Proceedings of 2003 USENIX Annual Technical Conference*, pages 269–282, June 2003.

[10] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One : Format of Internet Message Bodies. Technical report, November 1996. Note that this RFC obsoletes RFC1521, RFC1522, and RFC1590.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*, chapter 5. 3rd edition, may 2002.

[12] Song Jiang and Xiadong Zhang. Lirs : An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Performance Evaluation Review - International Conference on Measurement and Modeling of Computer Systems*, pages 31–43. ACM Sigmetrics, June 2002.

[13] Theodore Johnson and Dennis Shasha. 2q : A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on VLDB*, pages 439–450, 1994.

[14] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong San Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, pages 119–134, october 2000.

[15] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong San Kim. *LRFU : A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies*, pages 1352–1360. december 2001.

[16] Nimrod Megiddo and Dharmendra S. Modha. Arc : a self tuning, low overhead replacement cache. In *USENIX File & Storage Technologies Conference (FAST)*, march 2003.

[17] Nagi N. Mekhiel. Multi-level cache with most frequently used policy : A new concept in cache design. In *ISCA International Conference on Computer Applications in Industry and Engineering*, Honolulu, Hawaii, December 1995.

[18] Nextenso. Mms proxy relay 2.0 reference guide. Juillet 2003.

[19] James E. Pitkow and Margaret M. Recker. A simple yet robust caching algorithm based on dynamic access patterns. In *Proceedings of the 2nd International Multimedia Conference*, 1997.

[20] Ganesh Santhanakrishnan, Ahmed Amer, Panos K. Chrysanthis, and Dan Li. Gd-ghost : A goal-oriented self-tuning caching algorithm. In ACM Press, editor, *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1141–1145, 2004.

[21] Sabarinathan Sayiraman, Senthil Kumar Dayalan, and Shanmugavel Mani Subbiah. A framework for mf-lru (most frequently-least recently used) replacement policy. In *9th International Conference on High Performance Computing (HiPC 2002)*, Bangalore, India, December 2002.

# Annexes

# Annexe A

# Cas d'utilisation principal du MMS

La figure A.1 (page suivante) représente le cas d'utilisation principal du MMS tel qu'il avait été présenté dans le chapitre 1 (cf. figure 1.2 page 21) dans le cadre de la *Nextenso Proxy Platform.*

L'expéditeur est symbolisé par le *Terminal A* et le destinataire par le *Terminal B*. Le symbole *@CTRL* représente la *proxylet* d'*access control* (pour rappel, la description de ces différentes *proxylets* peut être trouvée dans le chapitre 2).

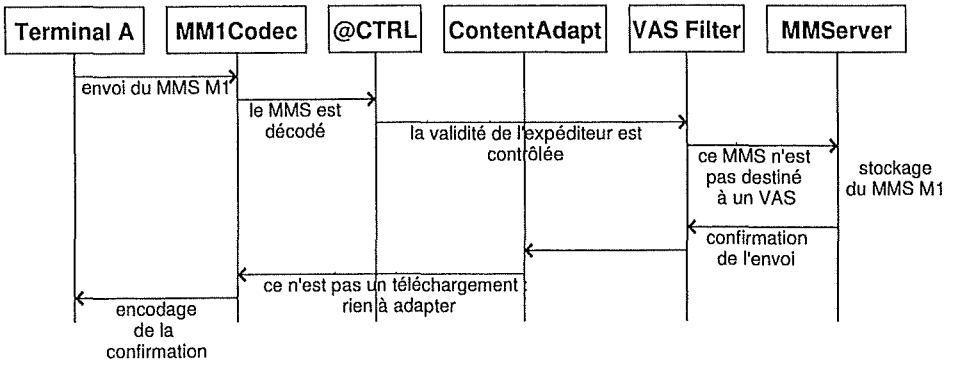
La phase I illustre l'envoi du MMS par l'expéditeur. Après avoir été décodé par le *MM1 Codec*, le message est vérifié par l'*access control* afin de s'assurer, notamment, que l'expéditeur est bien autorisé à envoyer un message en utilisant ce *MMS Relay*. Après avoir stocké le message, le *MMServer* retourne une confirmation d'envoi au *Terminal A*. Durant cette phase, le *VAS Filter* et le *Content Adaptor* n'ont pas eu de rôle à jouer.

Dans la phase II, le *MMServer* envoi la notification de réception d'un nouveau message (contenant l'URL nécessaire pour demander son téléchargement) au destinataire au moyen du *Push Proxy Gateway*.

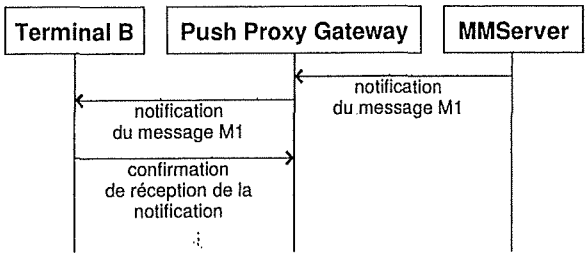
Comme indiqué dans la phase III, le destinataire demande ensuite le téléchargement du message qu'il vient de recevoir. Cette requête spécifique n'as pas besoin d'être décodée. Le *MMServer* récupère alors le MMS demandé et le retourne au destinataire. Remarquons que le *Content Adaptor* va s'assurer que le message est bien compatible avec les caractéristiques techniques du *Terminal B* et modifier le contenu du MMS en conséquence si nécessaire.

Enfin, dans la phase IV, le *Terminal B* confirme la réussite du téléchargement. Comme il s'agissait de l'unique destinataire, le *MMServer* peut alors effacer le MMS concerné.

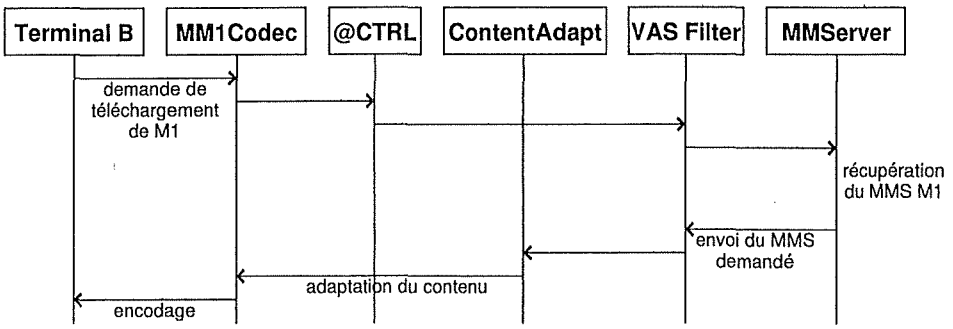
***Phase I : envoi du MMS***




***Phase II : le destinataire reçoit la notification***




***Phase III : téléchargement par le destinataire***




***Phase IV : confirmation du téléchargement***

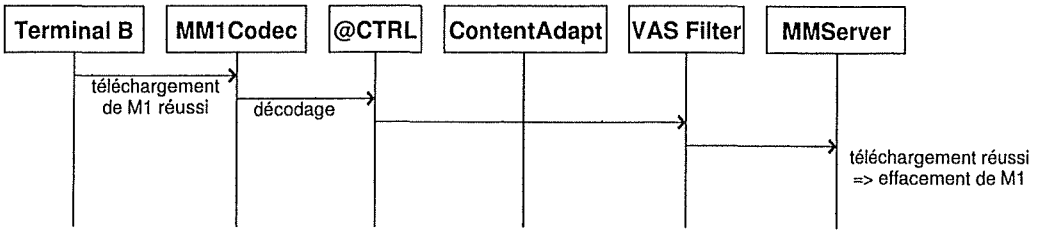



FIG. A.1 – Illustration du cas d'utilisation principal du MMS dans le cadre de la *Nextenso Proxy Platform*

# Annexe B

# Définition du langage permettant la sélection des MMS

Ce langage permet de définir les règles qui serviront à sélectionner quels MMS seront insérés dans la cache. Sa syntaxe est présentée au format BNF[1]. L'élément de base est `RULES_DEF`.

## B.1 Syntaxe

```
LETTER          ::= [a-zA-Z_]
DIGIT           ::= [0-9]
NUMBER          ::= DIGIT+
VARNAME         ::= LETTER(LETTER|DIGIT)*
OPERAND         ::= NUMBER|VARNAME
LOG_OPERATOR    ::= "<"|"<="|"="|">="|">"
BOOL_OPERATOR   ::= "|"|"&"

RULE_TYPE       ::= "CACHE"|"PRECACHE"
LOG_EXPR        ::= OPERAND LOG_OPERATOR OPERAND
BOOL_EXPR       ::= LOG_EXPR|LOG_EXPR BOOL_OPERATOR LOG_EXPR|
                    "(" BOOL_EXPR ")"

RULE_DEF        ::= RULE_TYPE ":" BOOL_EXPR+
RULES_DEF       ::= (RULE_DEF CR LF)+
```

[1] Backus-Naur Form

## B.2 Sémantique

| *Variable* | *Description* | *Opérateurs autorisés* | *Type* |
|---|---|---|---|
| EXPIRY | Nombre de secondes restante avant l'expiration du MMS | $<, <=, =, =>, >$ | Seconde |
| RCPTS_COUNT | Nombre de destinataires **locaux** du MMS | $<, <=, =, =>, >$ | Entier |
| MSG_SIZE | Taille du MMS | $<, <=, =, =>, >$ | Octet |
| MSG_COUNT | Nombre de secondes restante avant l'expiration du MMS | $<, <=, =, =>, >$ | Entier |
| OCCUPATION | Taux d'occupation de la cache | $<, <=, =, =>, >$ | Pourcentage |
| FROM | Canal source du MMS (MM*1*, MM*3*, ...) | $=$ | Entier |

Le résultat de tous les opérateurs du tableau suivant est de type booléen.

| *Opérateurs* | *Description* | *Priorité* |
|---|---|---|
| $(,)$ | | 1 |
| $<, <=, =, =>, >$ | Opérateurs de comparaison logique | 2 |
| & | Opérateur logique ET (à utiliser avec des valeurs booléennes) | 3 |
| \| | Opérateur logique OU (à utiliser avec des valeurs booléennes) | 4 |

# Annexe C

# Définition du protocole de communication avec la cache

## C.1 Syntaxe

```
General codes values

0-63          : status code
64-127        : commands code
128-191       : value types
191-255       : special tags

Primitive types definitions

LONG = signed long integer value on 32 bits (big endian)
BYTE = unsigned char value on 8 bits

Response types definitions

OK                  ::= 0
INVALID_PROTOCOL    ::= 1
CACHE_IS_FULL       ::= 2
ALREADY_IN_CACHE    ::= 3
NOT_FOUND           ::= 4
MALFORMED_REQ       ::= 5
INTERNAL_ERROR      ::= 6
ALREADY_EXPIRED     ::= 7
```

```
Commands definitions

CLOSE                ::= 64
PUT                  ::= 65
GET                  ::= 66
REMOVE               ::= 67
REPLACE              ::= 68
SET_URL              ::= 69
CHECK                ::= 70
GET_INFO             ::= 71

Value types definitions

LONG_ID              ::= 128
BYTE_ARRAY_ID        ::= 129

Special tags definitions

PROTOCOL_ID          ::= "CSP"
END_TAG              ::= 0xFF

Grammar definition

VERSION ::= LONG          (int.int - e.g. 1.0)

LONG_T ::= LONG_ID LONG BYTE_ARRAY_T ::= BYTE_ARRAY_ID BYTE*

DATA ::=
      LONG_T
    | BYTE_ARRAY_T

COMMAND ::=
      PUT
    | GET
    | SET_URL
    | REMOVE
    | REPLACE
    | CHECK
    | GETINFO

RESPONSE_TYPE ::=
```

```
      OK
    | INVALID_PROTOCOL
    | CACHE_IS_FULL
    | ALREADY_IN_CACHE
    | NOT_FOUND
    | MALFORMED_REQ
    | INTERNAL_ERROR
    | ALREADY_EXPIRED

REQUEST ::= COMMAND (DATA)* END_TAG
RESPONSE ::= RESPONSE_TYPE (DATA)* END_TAG
```

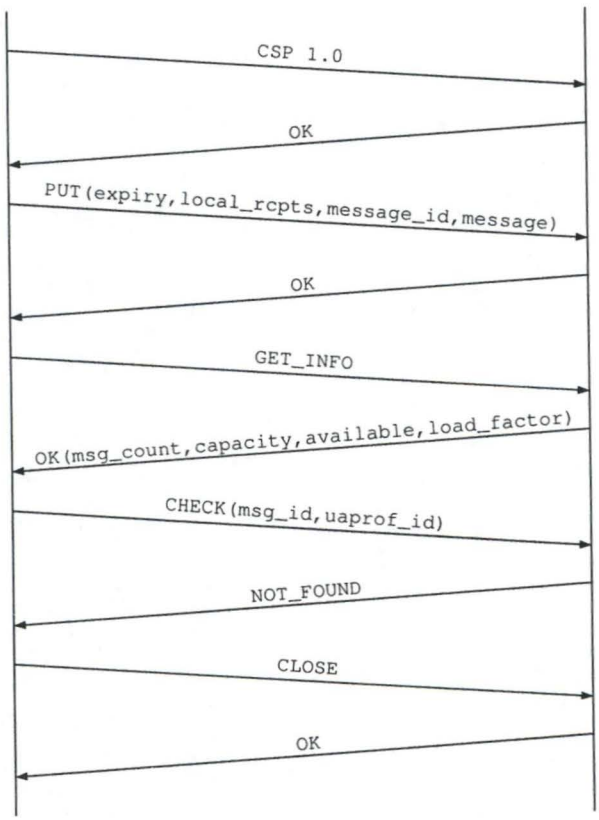
## C.2 Sémantique

1. Remarks
   - Strings **must be** null terminated, even when they are encoded as byte arrays.
   - To optimize implementation, all PDU[1] (except for the init sequence) must be prefixed with the length of the PDU on 32 bits
2. Opening sequence
   After opening a new connection to the server, the client must send the following sequence :
   `PROTOCOL_ID VERSION END_TAG`
   If the server support this version, it must return `OK END_TAG` and be ready for incoming requests Otherwise it should return `INVALID_PROTOCOL END_TAG` and immediately close the connection
3. Closing sequence
   The server should treat any incoming request from the client until it receives a `CLOSE` command. It must then respond with `OK END_TAG` and close the connection after it had sent a response for all pending requests.
4. Commands syntax and semantic :

```
PUT: PUT (message_expiry:LONG_T) (local_rcpts:LONG_T)
(message_id:BYTE_ARRAY_T) (message:BYTE_ARRAY_T) [(url:BYTE_ARRAY_T)
(uaprof_id:BYTE_ARRAY_T)] END_TAG

-> RESPONSE : OK END_TAG
              CACHE_IS_FULL END_TAG
              ALREADY_IN_CACHE END_TAG
              ALREADY_EXPIRED END_TAG
```

[1] Protocol Data Unit

```
GET: GET (url:BYTE_ARRAY_T) [uaprof_id:BYTE_ARRAY_T] END_TAG
     GET (message_id:BYTE_ARRAY_T) [uaprof_id:BYTE_ARRAY_T] END_TAG

-> RESPONSE : OK (message:BYTE_ARRAY_T) END_TAG
              NOT_FOUND END_TAG

CHECK: CHECK (url:BYTE_ARRAY_T) [uaprof_id:BYTE_ARRAY_T] END_TAG
       CHECK (message_id:BYTE_ARRAY_T) [uaprof_id:BYTE_ARRAY_T] END_TAG

-> RESPONSE : OK END_TAG
              NOT_FOUND END_TAG

REMOVE: REMOVE (url:BYTE_ARRAY_T) END_TAG
        REMOVE (message_id:BYTE_ARRAY_T) END_TAG

-> RESPONSE : OK END_TAG
              NOT_FOUND END_TAG

REPLACE: REPLACE (message_id:BYTE_ARRAY_T) (message:BYTE_ARRAY_T) END_TAG

-> RESPONSE : OK END_TAG
              NOT_FOUND END_TAG

SET_URL: SET_URL (message_id:BYTE_ARRAY_T) (url:BYTE_ARRAY_T) END_TAG

-> RESPONSE : OK END_TAG
              NOT_FOUND END_TAG

CLOSE: CLOSE END_TAG

-> RESPONSE : OK END_TAG

GET_INFO : GET_INFO END_TAG

-> RESPONSE : OK (msg_count:LONG_T) (capacity:LONG_T) (available:LONG_T)
(load_factor:LONG_T) END_TAG
```




pour améliorer la lisibilité, la taille des PDUs et les END_TAG n'ont pas été indiqués.

FIG. C.1 – Exemple de diagramme de séquence