heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

IICT
INSTITUTE FOR INFORMATION AND COMMUNICATION TECHNOLOGIES

Mémoire de diplôme

Implémentation du protocole *Proxy Mobil IP* pour *IPv4*

Projet de diplôme TR 2008

Proposé par : l'HEIG-VD institut IICT. Pour le projet européen Dustbot

Professeur responsable : Robert Stephan
Collaborateur Scientifique : Doswald Alistair
Diplômant : Hercher Yann

Date : 15 décembre 2008
Lieu : 1400 Yverdon

---

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

iICT
INSTITUTE FOR INFORMATION AND COMMUNICATION TECHNOLOGIES

A B S T R A C T

Mobil IP est une extension pour *IPv4* et une partie intégrale de *IPv6*. De nombreuses *RFCs* ont déjà été écrites sur le sujet mais la mobilité au niveau de la couche *IP* de base sont définis dans *RFC 3344* pour *MIPv4* et dans *RFC 3775* pour *MIPv6*. Une personne désirant travailler sur la mobilité *IPv4* ou *IPv6* doit en premier lieu être familière avec un de ces protocoles. Par contre, de nombreuses additions ont été faites aux protocoles pour rajouter des fonctionnalités, de la sécurité, ou pour optimiser le temps de réponse. Deux extensions récentes ajoutent de nouvelles fonctionnalités à mobile *IP*, mais elles ne sont pas encore implémentées. Cependant, ces protocoles ne sont décrits pour l'instant que par des drafts et ne sont donc pas des protocoles finaux, comme *Proxy Mobile IP* par exemple.

C A H I E R D E S C H A R G E S

Proxy Mobile IP (*PMIP*) : le but de *Proxy Mobile IP* est de permettre à des stations qui n'implémentent pas de client mobile *IP* de pouvoir malgré tout accéder à des fonctions de mobilité. Pour *IPv6* l'intérêt est surtout présent pour les clients qui ne disposent que de peu de ressources, étant donné que *MIPv6* devrait normalement être implémenté par toutes les stations capables d'utiliser *IPv6*. Par contre, pour *IPv4* il est beaucoup plus utile puisque les clients *mobileIPv4* ne sont que très peu déployés.
Le *Draft* pour *Proxy Mobile IPv4* est : draft-leung-mip4-proxy-mode-04 et celui de *Proxy Mobile IPv6* : draft-sgundave-mip6-proxymip6-02.
Il n'existe pas actuellement d'implémentation *open source* de *Proxy Mobile IPv4* ou *v6*. Le but du travail de diplôme est de développer une implémentation de ces protocoles.

T A B L E D E S M A T I E R E S

AVANT PROPOS

Pour bien cibler le projet, un chapitre est consacré à l'explication des protocoles *MIP* (*Mobile IP*) et *proxy MIP*. Une comparaison entre ces deux protocoles est faite afin de bien comprendre l'utilité du *Draft* proposé par Monsieur K. Leung. Le projet contient une pré-analyse du protocole avant d'être développé en *Java*. Pour cela un chapitre théorique de génie logiciel reprend point par point chaque élément du code à implémenter. Cela permet de bien planifier le travail et d'éviter des erreurs de conception. Pour chaque module important, du code sera associé à un chapitre qui contient un bref résumé de sa fonctionnalité, les itérations importantes de programmation, les choix faits selon le respect du *Draft* et les problèmes rencontrés. Le chapitre suivant contient le laboratoire expérimental et les tests effectués ainsi que les problèmes rencontrés.

Pour finir, le document contient un chapitre avec les améliorations et développements futurs qui n'ont pu être réalisés dans la vision de ce travail de diplôme ainsi qu'une conclusion sur le travail effectué contenant une autocritique. Ce projet reprend la suite du cours PPD (Pré Projet de Diplôme) de fin de troisième année section Télécommunication Réseau (TR) de la Haute École d'Ingénierie et de Gestion du Canton de Vaud (HEIG-VD).

Règles typographiques

Ce document a été rédigé en respectant les conventions typographiques suivantes :

- Verdana 10 est utilisé pour le texte en général.
- `Courier 11` est utilisé pour le code.
- Les termes étrangers ainsi que les termes importants et abréviations apparaissent en *Verdana italique*.
- Les références bibliographiques mentionnées dans le mémoire figurent entre crochets [].
- Chaque abréviation se retrouvera dans une liste en fin de document, sur une feuille A3 pouvant se déplier pour être suivie avec le rapport.

Respect des Drafts et RFCs

Dans chaque chapitre du document, traitant du code de programmation, se trouve une section *MUST* qui définira ce que le code doit contenir afin de respecter le *Draft* ou le *RFC* (avec référence).

Code

Le code est régi par la licence *GPL*. Il est accompagné d'une documentation exhaustive en anglais permettant à toute personne de pouvoir reprendre le code et de le modifier ou de l'améliorer à sa convenance. Il sera développé et testé sur les systèmes d'exploitation suivants :

- *Windows XP*
- *Linux*

CHAPITRE 1

INTRODUCTION

Historiquement, les réseaux informatiques englobaient de grandes quantités de clients fixes, connectés au réseau directement par un câble ethernet. Depuis l'avènement des réseaux *Wireless*, les ordinateurs et plus particulièrement les portables *PDA* et autre *Smartphones* ont vu naître une certaine liberté dans leurs déplacements. Cette liberté est certes relativement limitée car l'utilisateur reste attaché à la même borne *Wireless*, voir en poussant un peu plus loin au même réseau. Il est donc impossible actuellement de garder une liaison non-interrompue avec un hôte distant tout en changeant de réseau.

C'est dans cette idée que le protocole *Mobil IPv4* est né. Il permet en installant un petit client sur l'élément mobile de pouvoir se déplacer tout en gardant sa liaison avec l'hôte distant ! Mais ce système, actuellement à l'état de *RFC*, nécessite une intervention au niveau du client mobile et certains appareils de type *Smartphone* ou *PDA* ne peuvent pas recevoir cette modification. C'est alors que Monsieur Kent Leung s'est décidé à reprendre ce *Mobil IPv4* et de l'améliorer en supprimant l'installation du client et de proposer ce *Draft* à *l'IETF* [*IETF*]. Ce nouveau protocole se nomme *Proxy Mobile IP*, il consiste à rester dans la même idée que son parent en réutilisant certaines parties. L'amélioration est que le soft installé sur le client mobile se déplace directement sur un élément passif du réseau (un routeur *Wireless* par exemple). Ceci entraine bien sûr quelques modifications du programme mais permet de donner la possibilité à n'importe quel client disposant d'une gestion d'adresse *IP* basique avec un client *DHCP* d'obtenir la mobilité.

CHAPITRE 2

MIP ET PROXY-MIP

2.1. *Mobil IPv4* (*MIPv4*)

2.1.1. Historique:

L'idée de mobilité *IP* permettant de conserver son adresse *IP* est apparue au début des années 90. Plusieurs *Drafts* et *RFCs* ont vu le jour jusqu'à être rendu obsolètes par le dernier document en date, le *RFC 3344* édité en 2002. Beaucoup de constructeurs dans le domaine du *networking* ont développé leur propre solution, disponible moyennant une contribution financière. Certaines personnes ou instituts mettent en ligne du code *Open Source* permettant d'installer cette technologie facilement et d'en profiter gratuitement.

2.1.2. Implémentation

Pour que le protocole *MIP* fonctionne, le réseau doit comporter au minimum deux éléments et un autre conseiller qui est, lui, optionnel.

Incontournable, le *Home Agent* (*HA*) est l'entité principale à laquelle sont connectés tous les réseaux éloignés. C'est par lui que passent toutes les informations envoyées par le client mobile en direction de l'hôte distant. Ce dernier croit discuter avec le client mobile, mais discute réellement directement avec le *HA* qui achemine les paquets dans le bon réseau éloigné dans lequel se trouve le client mobile. Le *Mobile Node* (*MN*) est la deuxième partie obligatoire et se situe au niveau du client. Cet élément peut se mouvoir entre différents réseaux tout en gardant la communication avec le *HA* via un tunnel.

Ce tunnel permet au client de se croire en réalité dans le même réseau (*Home Network* : *HN*) que le *HA*. Pour ce faire, les paquets envoyés sont encapsulés pour franchir tous les réseaux intermédiaires avant d'être désencapsulés dans le *HN* pour atteindre le *HA*. Cette méthode permet donc au client de conserver une adresse *IP* semblable à celle du *HN*. Dans le cas du *Draft* c'est un tunnel de type *GRE* (*Generic Routing Encapsulation*) qui est utilisé, voir [RFC2784].

Certains réseaux éloignés sont dotés d'un *Foreign Agent* (*FA*). Cet élément est là pour établir le tunnel *GRE* entre le *HA* du réseau *Home* et le réseau éloigné (*Foreign Network*). Son utilité n'est pas indispensable mais permet une gestion des éléments mobiles localement dans le réseau éloigné (*FN*). Il peut fonctionner de deux manières différentes. La première consiste à désencapsuler le tunnel provenant du *HA* et de fournir les données au client normalement. Cette méthode a pour effet de réduire la charge inutile de la bande passante *Wireless*. La deuxième solution est celle pour laquelle le *FA* sert uniquement à forwarder les paquets ainsi que le tunnel entre le *FA* et le *MN*. Dans ce cas, le tunnel est prolongé jusqu'au *MN* qui se débrouille pour désencapsuler les données. L'encapsulation commence toujours au niveau du *HA*, mais elle peut se terminer soit au niveau du *FA* soit à celui du *MN*. Pour connaître ce point de désencapsulation une *Care Of Address* (*CoA*) est utilisée. Elle spécifie au *HA* que le tunnel se termine à cet endroit. Il est à noter que les *FA* ne sont pas obligatoirement

liés à un seul *HA*. Au contraire, ils sont à même de pouvoir discuter avec plusieurs d'entre eux selon la volonté du client mobile.

Dans le cas du protocole *MIP*, le programme *Mobile Node* (*MN*) est obligatoirement installé sur l'ordinateur du client. Ceci peut poser certains problèmes de nos jours sur des appareiles basiques du genre *Smartphone* ou *PDA* disposants d'un contrôle faible sur le système par l'utilisateur. Le *HA* et le *FA* peuvent quant à eux se trouver dans un routeur *Wireless* ou sur un serveur avec une interface possédant un accès *Wireless*.

2.1.3. La mobilité

Le terme de mobilité est pour le client une notion qui lui permet de conserver son adresse *IP* tout le long de son chemin entre les différents sous-réseaux. Cette adresse *IP* appelée *Mobile Node Home IP Address* provient du *Home Network* (*HN*) dans lequel se trouve le *Home Agent* (*HA*). Lorsqu'un *MN* profite de sa mobilité en bougeant d'un réseau éloigné à l'autre (*Foreign Network* : *FN*), il peut rencontrer jusqu'à quatre cas de connexions différents suivant sa configuration voir Figure 1 ci-dessous.

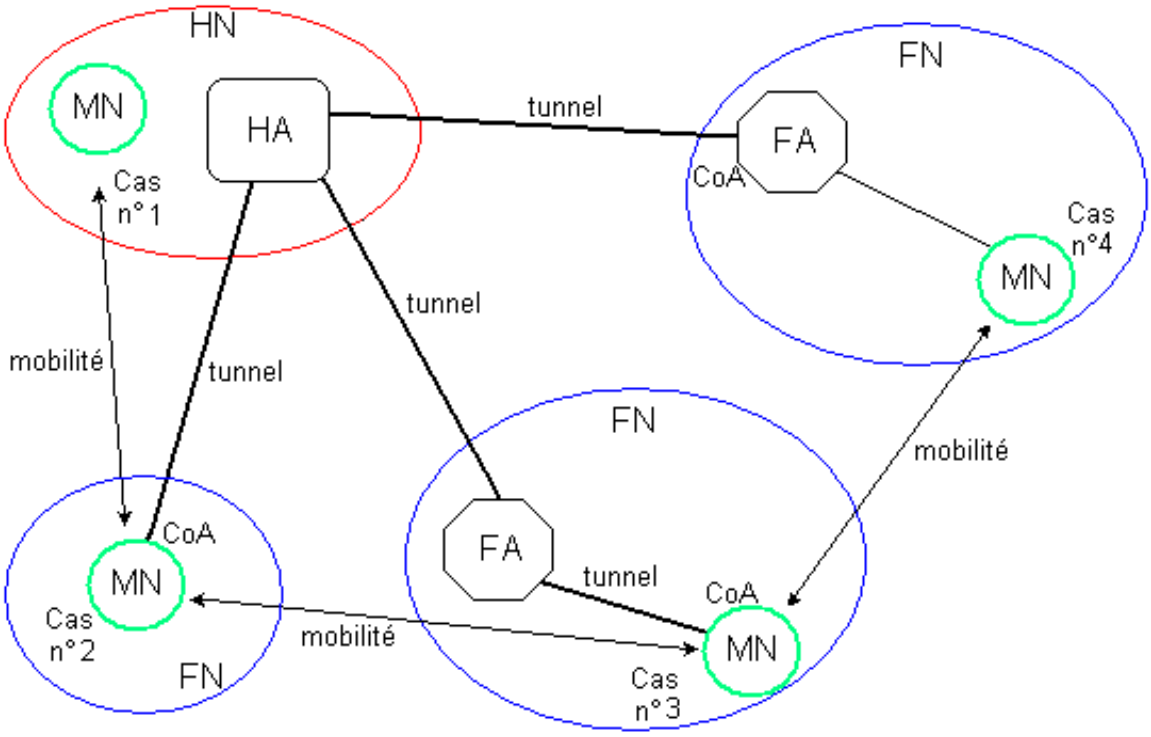


Figure 1 : Mobilité *MIPv4*

1. Le *MN* se trouve dans le réseau *HN* dans lequel se trouve aussi le *HA*. Dans ce cas, la mobilité ne s'applique pas puisque le client est dans le même réseau que son adresse *IP*. Il n'a donc pas besoin de se connecter au *HA* pour obtenir une mobilité.

2. Le *MN* se trouve dans un réseau qui ne possède pas de *FA*. Il tentera d'établir lui-même un tunnel directement avec le *HA*. Dans ce cas l'adresse *IP* précisant au *HA* la désencapsulation du tunnel, la *Care Of Address* (*CoA*), sera la même que l'adresse *IP* mobile du client.

3. Le *MN* est dans un *Foreign Network* et passe par un *FA* pour la mobilité. Ce dernier ne désencapsule pas le tunnel qui provient du *HA* mais le forward jusqu'au *MN* portant la *Care Of Address* (*CoA*).

4. Ce cas est presque identique au précédent à la différence près que la configuration du *FA* précise au *HA* que c'est lui qui porte la *CoA* et donc, désencapsule le tunnel. Les données entre lui et le client ne sont plus encapsulées. La *CoA* est l'adresse de l'interface du *FA* dans le réseau distant qui communique avec le *HA*. Cette adresse peut tout à fait être d'un réseau différent.

2.1.4. La sécurité

Ce protocole est actuellement principalement orienté vers les réseaux Wireless offrant une mobilité physique au client. Plusieurs clients peuvent se présenter au point d'accès *Wireless* (*AP*) pour tenter de se connecter sur le réseau. Une première sécurité, qui ne touche pas directement le protocole, est celle de la sécurisation de la connection *Wireless* qui interdira l'accès aux personnes non-autorisées.

Le protocole propose lui aussi plusieurs types de sécurité différents dont certains sont obligatoires et d'autres sont optionnels. Il utilise des messages de commande pour établir la connexion entre les différents éléments (*HA*, *FA* et *MN*). Ces messages servent à authentifier le client mobile et établir la connexion mobile pour lever le tunnel entre les réseaux *HN* et *FN*. Les messages envoyés au *HA* se nomment *Proxy Registration Request* (*PRRQ*) et ceux envoyés en retour au client, les *Proxy Registration Reply* (*PRRP*). Ils sont tous terminés par une signature qui englobe et assure l'intégrité des données entre le *MN* et le *HA*. Par défaut, cette signature doit être un *HMAC-MD5* (voir [RFC1321] et [RFC2104]) avec une clé secrète de 128 bits commune avec le *HA* en caractère *UTF8* ou en Hexadécimal *HEX*. D'autres types d'encryptions sont disponibles pour créer la signature (voir [RFC3344]).

Ces messages possèdent aussi un champ identifiant (*Identificator field*) de 8 octets. Il est composé de l'heure au format *Timestamp UTC* (*Universal Time Clock* voir [RFC1305]) de la machine, ainsi qu'une valeur aléatoire ajoutée dans le bit de poids le plus faible. Ce champ sert à éviter les *Replay Attacks* en se synchronisant avec le *Timestamp* du serveur.

Une sécurité non obligatoire est de paramétrer sur le *HA* tous les *FA* disposant du droit de s'y connecter. Les messages peuvent aussi contenir un champ d'authentification pour le *FA – HA* ainsi que pour le *MN – FA*. D'autres solutions sont encore disponibles. Pour cela il faut se référer au [RFC3344].

2.2. *PMIPv4* (*Proxy Mobile IPv4*)

2.2.1. Historique

L'idée du *ProxyMIP* [*DraftPMIP*] est apparue bien après le *MIP*. C'est pour pallier au problème de l'installation d'un programme (*MN*) sur le client que Monsieur K. Leung a proposé ce *Draft*. Ce dernier, sur lequel repose ce projet, date de début 2006 et est actuellement à sa version 9. Le plus gros avantage par rapport à son prédécesseur est qu'il n'y a plus besoin d'installer de programme sur l'élément mobile. Les *PDA* et autres *Smartphones* disposant d'une configuration *IP* limitée peuvent devenir eux aussi mobiles grâce à ce protocole. Un autre point fort est qu'il n'y a plus d'occupation de la bande passante par le protocole entre le client et *l'AP*. En effet les *Wireless* fonctionnent comme des *Hubs* et partagent entre leurs clients leur faible bande passante.

2.2.2. Similitudes avec le MIPv4

Le *PMIP* fonctionne avec les mêmes messages que le *MIP* ce qui les rendent compatibles dans le sens où ils peuvent se comprendre facilement. Le *HA* est une partie totalement commune aux deux protocoles car il assure le même travail. Le *FA* est quant à lui réutilisable seulement dans certains cas, sinon il devra être légèrerment modifié pour recevoir un module supplémentaire. Le time line est lui aussi identique aux deux protocoles, c'est-à-dire qu'ils passent par les mêmes phases et envoient les mêmes messages.

2.2.3. Différence avec le MIPv4

Dans l'idée générale, le *PMIP* fonctionne comme le *MIP* à la seule différence que le *MN* ne se trouve plus sur le client mais au même niveau que le *FA*. Le draft profite de ce changement de contexte pour renommer le *MN* en *PMIP Client*. Ce nouveau module est associé au *FA* et forme le *Proxy Mobile Agent* (*PMA*) au sein du *Foreign Network*. Il peut être d'un seul montant s'il vient à être développé dans son ensemble ou divisé en deux parties distinctes (*Split PMA*). Dans ce dernier cas, le *FA* d'un programme fonctionnant sur le protocole *MIP* peut être réutilisé et enverra des messages au *PMIP Client* qui simule le *MN* du protocole *MIP* mais au niveau du réseau. Les messages de commande utilisés dans le *PMIP* sont les mêmes que dans le *MIP* mais portent un nom différent. Il est donc tout à fait envisageable de faire tourner sur un réseau disposant de la mobilité les deux protocoles *PMIP* et *MIP* en parallèle mais seulement si le *PMA* est partagé.

Le but du *PMIP Client* est de trouver les informations de l'élément mobile et de négocier la mobilité de ce dernier avec le *HA*. En effet dans le cas du *MIP* les informations étaient à même le client mais maintenant le *PMIP Client* doit se débrouiller pour trouver ces informations lorsque un client se connecte. Pour ce faire, il doit se servir d'un service *AAA* (protocole d'authentification réseau voir [RFC2905]) qui détectera l'arrivée du nouveau client sur le réseau avant le protocole *PMIP*. Ce service fournit à travers le *Network Access Serveur* (*NAS*) les informations nécessaires depuis sa base de données. L'utilisation d'un *AAA* est donc obligatoire pour le *PMIP* ce qui n'était pas le cas du *MIP*.

Aussitôt que la négociation est acceptée et la mobilité mise en place, le *PMIP Client* doit simuler auprès du *HA* le client actuellement connecté. Il enverra pour cela des messages afin de maintenir et renouveler le tunnel. Le *PMIP Client* est lié avec le service qui fournit l'adresse *IP* au client mobile dont le temps de renouvellement doit être plus faible que le temps du renouvellement du tunnel. Ceci a pour effet d'avertir le *PMIP Client* que le client mobile est toujours connecté et qu'il peut renouveler le tunnel. Dernière grande différence, le *Draft* du *PMIP* requiert la désencapsulation du tunnel au niveau du *FA* dans le cas d'un *PMA* séparé.

Tableau de comparaison récapitulatif.

	PMIP	*MIP*
HA	oui	oui
FA	oui (dans le *PMA*)	pas obligatoire
MN	non	oui
PMIP Client	oui	non
PMA	oui	non
Trames standard RFC 3344	oui	oui
TimeLine standard RFC 3344	oui	oui
AAA	oui	pas obligatoire
NAS	oui	pas obligatoire
Désencapsulation tunnel par *MN* / *PMIP Client*	non (interdit)	pas obligatoire si *FA*
Désencapsulation par le *FA*	oui (obligatoire)	pas obligatoire si *MN*

2.2.4. Implémentation du service AAA dans le PMIP

L'utilisation d'un service *AAA* est obligatoire, ce qui n'était pas le cas pour le protocole *MIP*. Tout d'abord, il peut servir de première barrière de sécurité, car il est possible de le configurer de sorte à ce que chaque nouveau client qui tente de se connecter doit s'authentifier, par exemple avec un certificat ou nom et mot de passe. Mais il trouve réellement sont utilité pour fournir au *PMIP Client* toutes les informations nécessaires pour l'établissement de la connexion mobile. En effet, comme le *MN* n'existe plus, le client ne possède aucune donnée en interne il ne peut donc pas informer autrement que par sa présence le *PMIP Client*. C'est le service *AAA* qui, par authentification, pourra depuis sa base de données fournir au *PMIP Client* les informations du client dont il a besoin pour établir la mobilité.

Le fonctionnement théorique est relativement simple (voir Figure 2 ci-dessous). Un client qui tente de se connecter à une borne *Wireless* envoie via le protocole *EAP* (*Extensible Authentication Protocol* voir [RFC3748]) un message *Start* au *Network Access Serveur* (*NAS*). Cet élément se trouve sur le point d'accès du réseau et est en charge de répondre aux requêtes *EAP* et de les forwarder à un serveur *AAA*. Le *NAS* renvoie alors un message de demande d'identification au client qui s'empresse d'y répondre en donnant un nom d'authentification. Après que la borne ait reçu cette information, il l'encapsule dans le protcole *AAA* utilisé et l'envoie au serveur *AAA*. De là, le serveur va trouver une correspondance dans sa base de données et connaître les exigences de connection du client. Ainsi va commencer un long échange entre le client et le serveur pour échanger des informations d'authentification de connexion. A cette étape, il se peut qu'un tunnel crypté soit levé entre le serveur *AAA* et le client. Le *NAS* entre deux ne fait plus que passer les données de l'un à l'autre jusqu'à ce que la connexion soit établie et acquittée par un message d'acceptation (*Accept*) provenant du serveur *AAA*. Dans le cas du *PMIP* le *NAS* doit filtrer les données puis les envoyer au *PMIP Client* lorsque le client se voit donner la permission d'accèder au réseau.

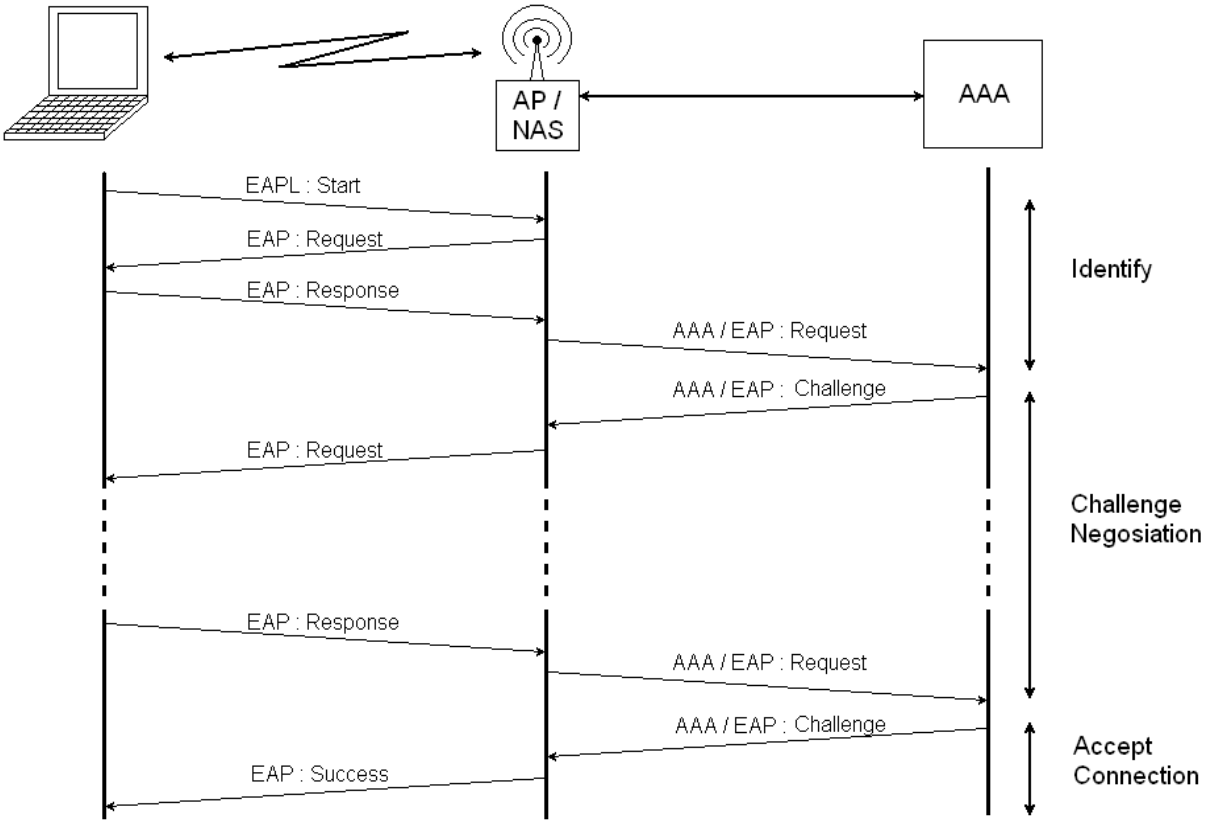


Figure 2 : *TimeLine* du protocole *AAA* / *EAP*

Pour une mobilité parfaite, le service *AAA* doit être séparé en deux parties comme pour le *FA* et le *HA* respectivement le *AAAF* (*foreign*) et le *AAAH* (*Home*). Le premier se trouve dans le réseau éloigné et c'est avec lui que communique le *NAS*. D'après l'information du client, le *AAAF* s'oriente pour *forwarder* les paquets directement vers le bon serveur *AAAH*. Ce dernier est dans le même réseau que le futur *HA* avec qui le client va communiquer. C'est pour cette raison notamment que le *Draft* conseille vivement l'utilisation du protocole *Diameter* disposant de plus d'options pour la mobilité que son prédécesseur *Radius*. De plus, ce protocole possède déjà des champs prévus pour contenir les informations relatives à la mobilité du client. Il existe des solutions open source de ce protocole sur Internet comme par exemple *OpenDiameter* [*OpenDiameter*].

2.2.5. La mobilité

Les différents types de mobilité sont moindres dans le cas du *PMIP*. En effet nous ne retrouvons que deux cas sur quatre vu précédemment dans le *MIP*. Dans le premier cas, le client mobile (*Mobile Device* : *MD*) se trouve dans le *Home Network*. Dans le deuxième cas, le client entre dans un réseau éloigné. Il passera par le *PMA* pour établir sa mobilité avec le *HA* voir Figure 3 ci-dessous.

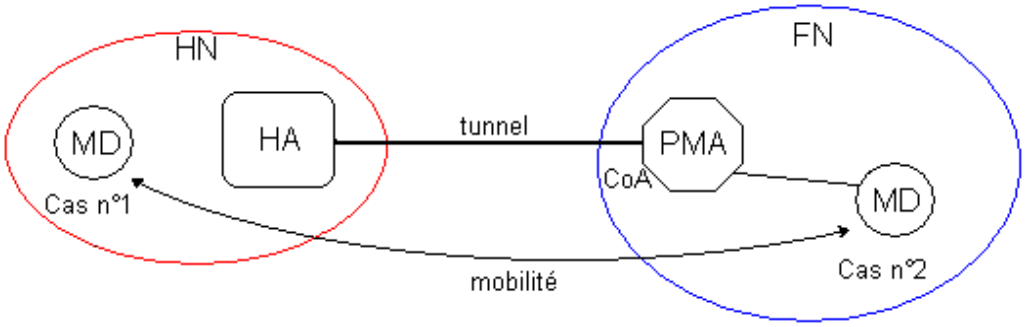


Figure 3 : Mobilité *Proxy Mobile IP* (*PMIP*)

1. Le client mobile *MD* se trouve dans le réseau *HN* dans lequel se trouve aussi le *HA*. Dans ce cas la mobilité ne s'applique pas puisque le client est dans le même réseau que son adresse *IP*. Il n'a donc pas besoin de se connecter au *HA* pour obtenir une mobilité. Ce cas est exactement le même que celui du protocole *MIP*.

2. Le *MD* se trouve dans un réseau éloigné et se connecte via un *PMA* pour obtenir sa mobilité. Le tunnel est ouvert uniquement entre le *HA* et le *PMA* car aucune information de mobilité ne circule sur le réseau *Wireless*.

2.3. Fonctionnement du protocole *PMIP*

Cette section est une traduction en français du *Draft* [*DRAFT PMIP*]. Elle explique en détail le fonctionnement de ce protocole ainsi que les différents cas de mobilité. Le protocole y est passé en revue dans sa globalité mais ne traite pas des points techniques qui seront développés plus bas dans les chapitres de développement du code.

2.3.1. Première connexion du client mobile

L'établissement d'une connexion mobile *PMIP* avec *FA* s'effectue en trois phases : *Authentication*, *Address Acquisition*, *Data Forwarding*, voir Figure 4. La première phase permet au client de s'authentifier sur le réseau via le *AAA*. La deuxième établit le tunnel avec le *HA* et retourne une adresse *IP* mobile fournie par ce dernier. La dernière phase est l'envoi et la réception des données par le client. A ce moment la mobilité est établie.

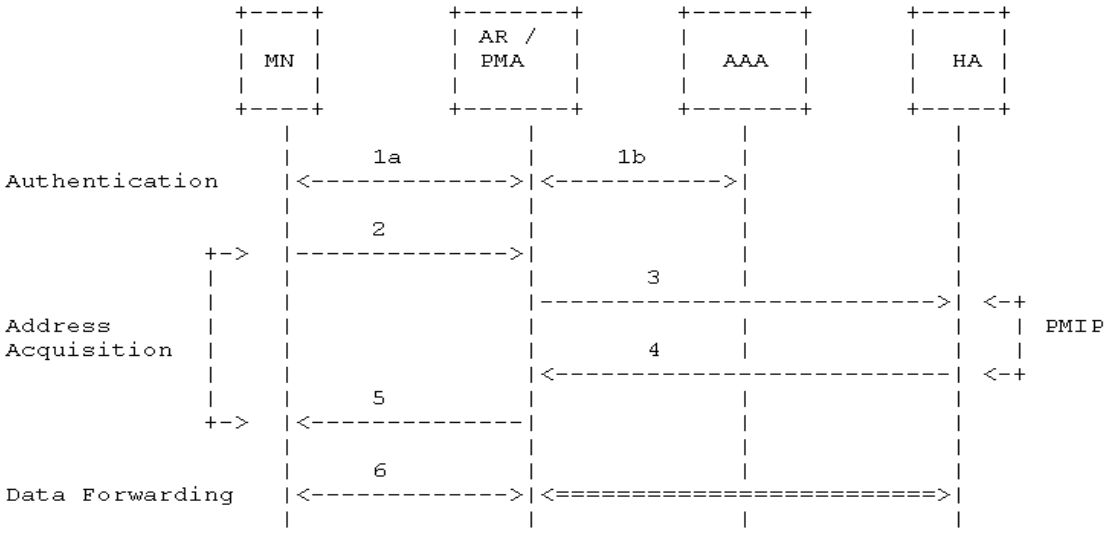


Figure 4 : Etablissement de mobilité d'un nouveau client

Authentification (*Authentication*)

1a : Le client mobile établit un lien de couche 2 (*OSI*) avec le point d'accès sans fil (*AP* : *Access Point*) et lance une demande d'authentification et d'autorisation qui transitera à travers *l'Access Router* (*AR*). Durant cette phase le *Mobile Device* (*MD* nom donné au client mobile) doit exécuter une requête *CHAP* ou *PAP* s'il est sur du *PPP* ou alors une requête *EAP* s'il est sur *Foo*. Dans ce cas, *l'AR* joue le rôle de serveur d'accès réseau *NAS* (*Network Access Server*) qui est un élément d'échange de données entre le client et le service *AAA*.

1b : Le client échange ses informations avec le serveur *AAA*, à travers *l'AR*, pour permettre son authentification et autorisation d'accès au réseau. Le serveur *AAA* renverra non seulement l'acquittement de mobilité mais aussi plusieurs paramètres comme le profil de l'utilisateur, l'adresse *IP* du *HA* et d'autres paramètres encore.

Acquisition d'adresse (*Address Acquisition*)

2 : Le *MD* émet une requête d'obtention d'adresse *IP* via *PPP*/*IPCP* ou *DHCP*. Pour le *DHCP* le client mobile envoie sa requête aux *DHCP* agents relais ou serveurs. Ces derniers ne renvoient l'adresse au client qu'après avoir terminé le signalement *PMIPv4* au *HA*.

3 : Le *PMA* envoie une requête *PRRQ* au *HA* dont il connaît l'adresse depuis l'étape 1b. Cette requête possède plusieurs champs dont l'adresse *CoA* (*Care Of Address*) du *PMA*.

4 : Le *HA* reçoit la requête et traite le contenu pour permettre la mobilité du client. Si la demande est acceptée le *HA* retourne un *PRRP* au *PMA* contenant les informations permettant de monter le tunnel. Dans ces informations, est contenue notamment la clé de sécurité du tunnel *GRE* [RFC2784].

5 : Après avoir reçu l'autorisation en retour, le *PMA* peut monter le tunnel et envoie une adresse *IP* au client. Deux cas se présentent selon le lieu où se trouve le *DHCP*. Dans le premier cas, le *DHCP* est sur l'*AR*, l'échange de données pour obtenir une adresse se passe entre le client et l'*AR*. Dans le deuxième cas, l'*AR* joue le rôle de *DHCP Relay agent*, qui fera transiter les demandes et les réponses pour le client, et le *HA* qui sera le serveur *DHCP*. Dans ce cas précis, le serveur renvoie une offre contenant l'adresse dans le champ "*yiaddr*".Echange de données (*Data Forwarding*)

6 : Le client possède maintenant une adresse *IP* lui permettant d'être mobile grâce à un chemin (tunnel) passant par le *PMA* et le *HA*.

2.3.2. Maintien de la connexion

Tant que le client ne bouge pas et reste dans le même réseau et derrière le même *PMA*, la session du tunnel est maintenue entre le *PMA* et le *HA* grâce au message d'enregistrement *PRRQ* et *PRRP*. Ces derniers contiennent le temps de vie du tunnel et ainsi par un simple envoi, peuvent le renouveler. Pour cela le *PMA* vérifie que le client est toujours connecté puis renvoie un message au *HA* (voir Figure 5) qui répondra immédiatement en acceptant ou refusant.

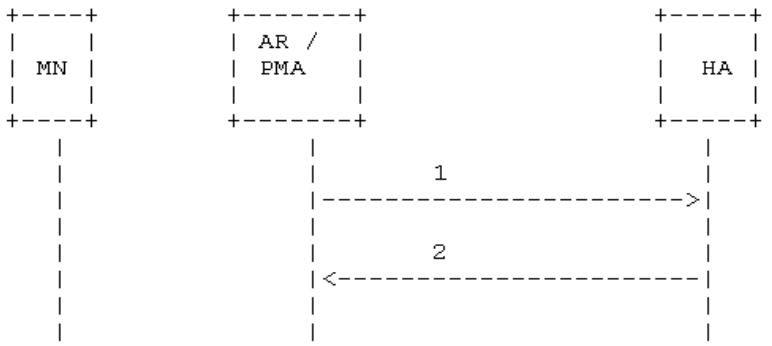


Figure 5 : Prolongement du temps de la connexion mobile

Renouvellement de connexion

1 : Avant que le temps de vie du tunnel n'expire, le *PMA* s'informe de la présence du client. Le *PMA* envoie alors une requête *PRRQ* au *HA* pour étendre la durée du bail de mobilité auprès du *HA*.

2 : Le *HA* renvoie une réponse *PRRP* pour confirmer le prolongement du bail.

2.3.3. Changement de PMA

Le changement de *PMA* se produit lorsque le client se déplace d'un réseau éloigné à un autre. Ce déplacement n'est pas à confondre avec un retour du client mobile dans le *Home Network*. Dans le cas d'un changement de *PMA* la reconnexion s'effectue en trois phases (voir Figure 6). Le time line ressemble à la première connexion d'un mobile, mais seule la phase d'obtention de l'adresse *IP* diffère, car le client mobile est censé renouveler son adresse et non en obtenir une nouvelle.

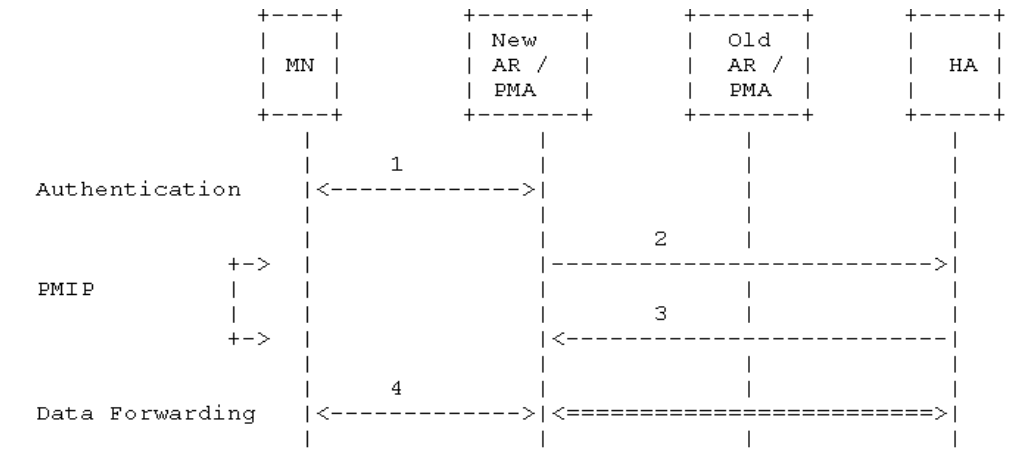


Figure 6 : Déplacement dans un autre réseau (Changement de *PMA*)

Authentification dans un nouveau réseau

1 : Le client établit un lien de couche 2 avec la borne *Wireless* et envoie une demande d'authentification/autorisation avec le nouvel *AR*.

2 : Après l'authentification le *PMA* envoie une requête *PRRQ* au *HA* contenant l'adresse du client. La requête contient l'adresse *CoA* du nouveau *PMA*.

3 : Après avoir reçu le *PRRQ*, le *HA* met à jour sa table de liens et renvoie la *Home address* dans le *PRRP*. Le *PMA* peut maintenant monter le tunnel pour l'établissement de la connexion.

4 : C'est ici que la connexion pour le mobile reprend après une déconnexion rapide de la couche *L2*.

2.3.4. Libération des ressources

La procédure de libération des ressources a lieu lorsque le client se déplace sur un autre *PMA*, dans le réseau du *HA* ou bien se déconnecte. Le *PMA* ne voyant plus de client connecté ne tentera pas de renouveler le tunnel avant le délai imparti. C'est le *HA* qui va prendre le devant (voir Figure 7) en ne voyant pas arriver de renouvellement de tunnel ou si le client s'est annoncé vers un autre *PMA*. Pour cela il envoie un message au *PMA* pour lui annoncer de libèrer toutes ses ressources concernant ce client.

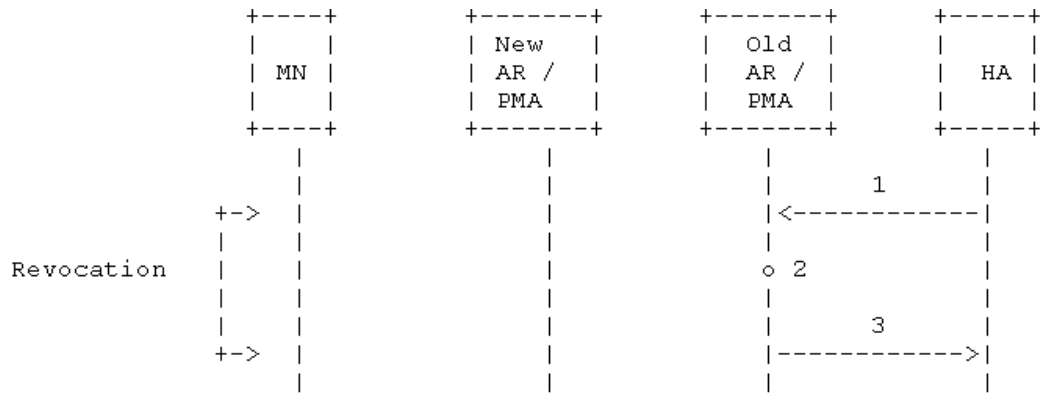


Figure 7 : Révocation du lien entre le *HA* et l'ancien *PMA*.

1 : Le *HA* détecte que le client est arrivé sur un nouveau *PMA* ou que le *TTL* de la ligne est arrivé à terme et décide de supprimer la connexion avec l'ancien *PMA*. Pour cela, il lui envoie un message de révocation (*Registration Revocation*) pour demander la libération du canal.

2 : Après avoir reçu cette demande, l'ancien *PMA* annule tout ce qui est en rapport avec la mobilité liée avec ce client.

3 : Le *PMA* renvoie alors une réponse de révoccation (*Revocation Acknowledgement*) en réponse au *HA* quand tout est terminé.

2.3.5. Communication de client à client sur le même PMA

Une communication entre deux éléments mobiles sur le même *PMA* est obligée de passer par le *HA* à travers leur tunnel respectif. Il ne peut pas y avoir de raccourci direct entre les deux clients. A l'exception faite des diffusions (*broadcast*) permettant de trouver le *DHCP* et les requêtes *ARP*. Ils doivent tous transiter par le *HA* qui déterminera à qui il doit les transférer suivant le sous-réseau de l'auteur.

2.4. Complément au cahier des charges

Le but final de se projet de diplôme est de développer, implémenter, puis tester le protocole du *Proxy Mobil IPv4* (*PMIPv4*) encore à l'état d'ébauche (*Draft*). Pour cela il est possible de reprendre des parties de programme à source libre (*Open Source*) héritées du *MIPv4*. Le développement contient donc une première analyse génie logiciel, puis la programmation du module *Proxy Mobil Client* (*Soft* client du *MIPv4* à déplacer sur le routeur d'accès pour *PMIPv4*) ainsi qu'une phase de test.

- Obtenir le plus rapidement un programme viable fonctionnant sur le protocole *PMIPv4* offrant une mobilité minimale qui devra être améliorée par la suite.
- Le développement du *Proxy Mobil Client* sera fait en langage *Java*.
- Le programme devra au minimum tourner sur les systèmes d'exploitation *Windows XP* et *Linux Ubuntu*. Ainsi que gérer des clients tournant sur les mêmes *OS*.
- Le choix du serveur *AAA* reste à la charge de l'élève, mais l'utilisation du protocole *Diameter* est fortement conseillée (recommandé par le *Draft*).
- Les éléments *MIPv4* réutilisables pour ce projet seront issus du logiciel libre *Dynamics* modifiés par Monsieur A. Doswald.
- Au final le programme créé restera sous licence libre *GPL* (*open source*) et devra être documenté en conséquence (en anglais de préférence).

CHAPITRE 3

GENIE LOGICIEL

3.1. Analyse du projet

3.1.1. Obligation

Ce projet est en premier lieu un travail de diplôme, mais reste quand même un programme open source qui peut se retrouver dans les mains d'une personne qui pourrait reprendre ce travail. Il est donc très important de le penser de la manière la plus simple et la plus compréhensible possible et cela toujours accompagné de commentaires explicites.

Le respect des *Drafts* et des *RFCs* est primordial dans ce projet car il doit respecter une certaine rigueur de convention pour pouvoir être mis en adéquation avec d'autres programmes. C'est pour cela que la vue d'ensemble ainsi que la structure du projet devra se faire en tenant compte de la hiérarchie des priorités définies par les *RFCs*. Ces derniers implémentent un système de priorité et d'obligation pour le respect des protocoles. Pour cela ils utilisent les mots : "*MUST*" (doit), "*MUST NOT*" (doit pas), "*SHOULD*" (devrait), "*SHOULD NOT*" (devrait pas), "*MAY*" (peut), "*RECOMMANDED*" (recommandé) et "*OPTIONAL*" (optionnel). Les *MUST* devront impérativement être faits et décrits au niveau du code et du rapport.

3.1.2. Structure du projet

Le réseau actuel de test s'appuie sur une architecture *MIPv4* fonctionnant sur le programme libre *OpenDynamics*. Après une analyse suivie de tests, il s'est révélé possible de travailler et de réutiliser le *HA* et le *FA* de ce programme. Nous allons donc nous pencher plus particulièrement sur la partie du *Drafts PMIPv4* qui évoque la possibilité de séparer le *PMA* en deux parties.

Le *PMA* est composé de deux entités : le *FA* et le *PMIP Client*. Ce dernier simule la présence du *MN MIPv4* au niveau de l'élément réseau en discutant avec le *FA*. D'un autre côté, il doit être à même de pouvoir redonner au client une adresse *IP* mobile dynamiquement. A cette étape, nous pouvons déjà fractionner la partie *PMIP Client*, qui est le programme à développer en deux parties importantes. La première consiste à fournir une adresse à l'élément mobile via deux méthodes décrites dans le *Draft* : par *DHCP* ou *PPP*/*IPCP*. Seule la méthode d'obtention d'adresse par *DHCP* sera implémentée dans ce projet. La deuxième partie est le développement du *PMIP Client* qui reste l'élément déterminant du protocole *PMIP*. Ce dernier a besoin pour communiquer avec le *HA* de connaître les informations du client mobile qui lui sont fournies par la *NAS* du service *AAA*. Cette partie doit être développée car il n'existe actuellement pas de solution *Java Open Source* qui propose déjà cette solution. Cela fait trois programmes distincts utilisant des protocoles différents qu'il faut mettre en adéquation autour d'une seule entité (Voir Figure 8, page suivante).

Le projet sera développé afin d'obtenir le plus rapidement possible un protocole utilisable respectant les iterations obligatoires des *Draft* et *RFC*. A partir de cette itération il sera possible de tester le protocole ainsi que de compléter le code selon les exigences moindres du *Draft*.

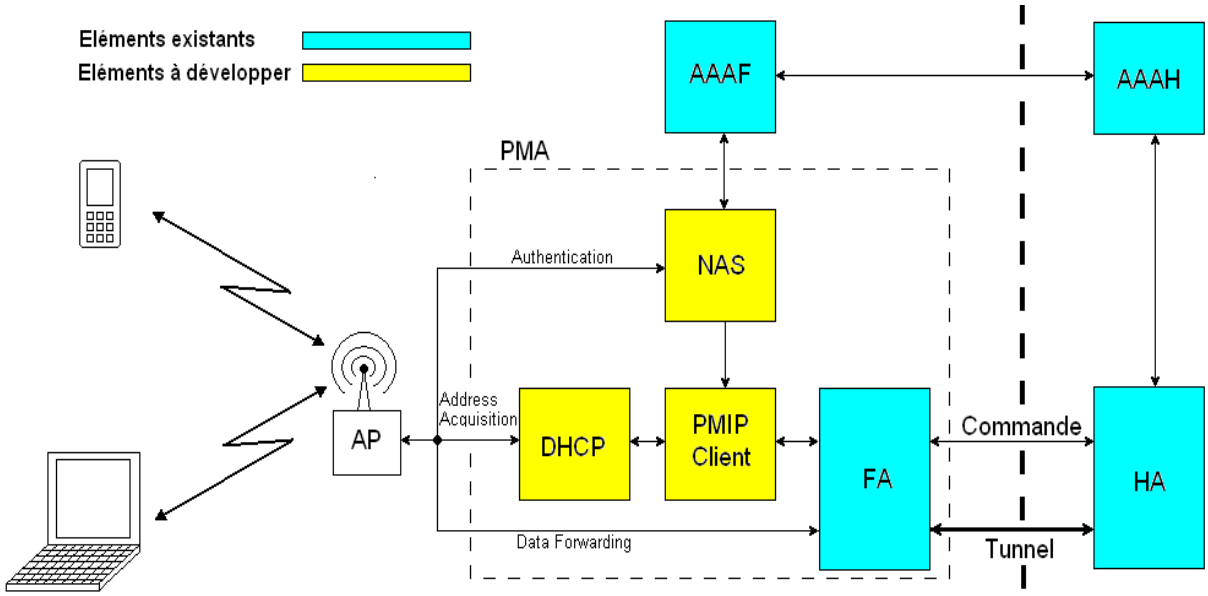


Figure 8 : Structure *PMIP* avec *PMA* Partagé (*Split PMA*)

3.2. Split PMA

3.2.1. PMIP Client

Le rôle du *PMIP Client* est de simuler un client *MIPv4* directement depuis l'élément réseau. Il doit donc posséder toutes les informations du client utiles à la négociation de sa mobilité avec le *HA*. A chaque nouveau client mobile sur le réseau doit être associé un *thread PMIP Client* possédant tous les paramètres de mobilité propres au client. Après que le *NAS* a transmis les informations au *PMIP Client*, il va lancer une nouvelle instance *thread* qui va attendre un certain délai que le client se présente par le protocole *DHCP*. Passé ce temps, le *thread* doit se terminer pour ne pas occuper de la mémoire inutilement.

Après être informé de l'arrivée du client par le *DHCP*, le *thread* va tenter de négocier la mobilité du nouveau client avec le *HA*. Si la mobilité est acceptée, il doit retourner l'adresse *IP* mobile reçue au client via le service *DHCP* sinon lui dire de se réorienter vers un autre serveur *DHCP*. A partir de ce moment, le client peut discuter avec le *HA* à travers le tunnel qui vient d'être créé.

Après cette étape, le *PMIP Client* doit assurer le renouvellement du tunnel entre le *FA* et le *HA*, jusqu'à ce que le client se décide à quitter le réseau. Après toutes ces descriptions nous pouvons nous rendre compte que l'entité principale qui doit gérer tous les messages entre eux est bien le *PMIP Client*. Tout cela montre une gestion relativement importante entre les messages envoyés et reçus, ainsi qu'un échange synchronisé de données avec la partie *DHCP*.

La gestion d'envoi et de réception des messages n'est pas la partie la plus difficile, car l'envoi se fait depuis un port libre non assigné en direction du port 434. Les réponses se font en retour sur le même port libre créé par le client, qui reste ouvert pendant tout le temps de la transaction jusqu'à la libération de la ressource. Chaque *thread* sera donc indépendant des autres. La partie la plus complexe sera toute la gestion du traitement des messages provenant du *HA*.

3.2.2. DHCP

Cette partie est relativement plus imposante et plus compliquée que le *PMIP Client*. Il peut fonctionner de deux manières différentes, soit en serveur *DHCP*, soit en *Relay DHCP* [RFC2131] qui transmet les demandes plus loin. La partie serveur sera implémentée en premier et sera complétée par la suite en relai.

Le serveur *DHCP* est en charge de recevoir en premier un nouvel arrivant ne disposant d'aucune adresse. Il doit alors avertir le *PMIP Client* de débuter une demande de mobilité auprès du *HA*. En cas de réponse négative, le serveur *DHCP* doit redonner une adresse de sa propre initiative. Si la réponse est positive, il doit transmettre au client l'adresse *IP* fournie par le *HA*. Comme pour le *PMIP Client*, chaque nouvel arrivant a un *thread* dédié pour la négociation du protocole ce qui permet de gérer plusieurs clients. Le fonctionnement est relativement simple au premier abord mais se qui va compliquer le processus, c'est que le serveur doit écouter uniquement le port 67. Lorsque le service *DHCP* reçoit un message sur ce port, il doit l'analyser et le transmettre au bon *thread DHCP*. Il faut créer une classe *dispatcher* qui va générer le *sockets* pour tous les *threads*. Ce *socket* est uniquement en écoute sur l'aiguilleur de paquet (*Dispatcher*) et peut être utilisé par les *thread* pour envoyer des paquets (voir Figure 9 ci-dessous). En résumé, une classe d'aiguillage (*Dispatcher*) doit écouter sur le port 67 et vérifier selon la *MAC* adresse des messages qui arrivent pour les transmettre au bon *thread*. Ces *threads* associés à chaque nouveau client sont chargés de commencer et gérer la négociation avec le *PMIP Client* ainsi que de terminer la procédure d'obtention d'adresse via le protocole *DHCP*. Pour cela il faut utiliser des états bloquants de concurrents comme le *CountDownLatch* en *Java*. Afin de gérer les *threads* facilement, il faut les stocker dans un tableau d'objet dont l'index est la *MAC* adresse du client afin de pouvoir les retrouver et leur transmettre les nouveaux messages. Voici un aperçu de la structure d'aiguillage des messages *DHCP*.

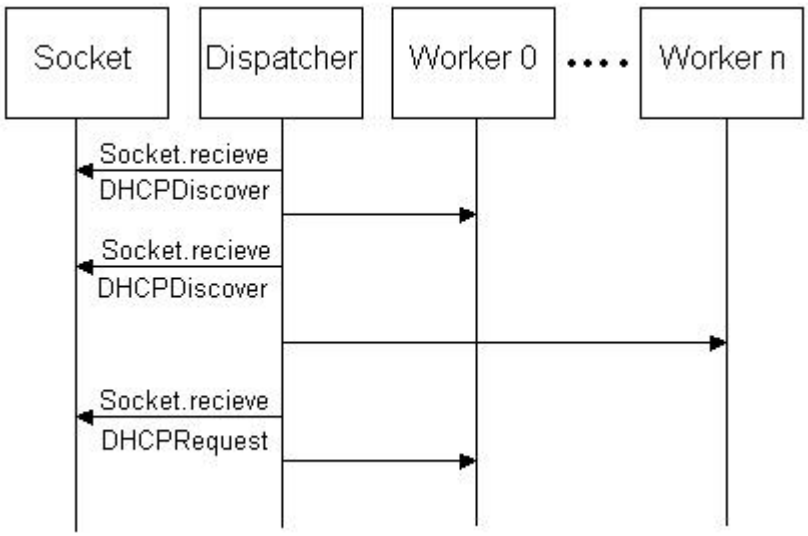


Figure 9 : Structure d'aiguillage des messages *DHCP*

La figure page-précédente représente l'aiguilleur (*Dispatcher*) écouter sur le même socket et distribuer le message au bon *thread* (*Worker*). Le système peut ainsi gérer plusieurs demandes de connexion de clients simultanément. Toute la logique de gestion des messages se trouve au niveau des *threads* qui lors de la réception du message savent comment le traiter et y répondre.

3.3. Les classes et packages

Package jPMIPClient : Contient l'information pour créer, envoyer et recevoir les messages.

- *PMIPClientSocket*: Permet de créer le *Socket* ainsi que les méthodes d'envoi et réception.
- *PMIPClientMessage*: Contient les méthodes pour la création des options des messages *PMIP Client*
- *PMIPClientExtension*: Classe abstraite de construction des extensions *DHCP* parente des classes suivantes :
  - *ExtensionTypeLengthValue*
  - *ExtensionTypeLongFormat*
  - *ExtensionTypeMobileID*
  - *ExtensionTypeSecurityAuthentication*
  - *ExtensionTypeShortFormat*
- *PMIPReadConfig* : Permet la lecture du fichier de configuration du *PMIP* et offre des méthodes d'accès à ses variables.
- *FixeSetTTL* : Permet de changer le *TTL* par défaut du système d'exploitation sur lequel se trouve le programme

Package *jDHCP* : Trouvé sur Internet [*Java DHCP*]. Contient trois classes de base dont une ajouté par le diplômant.

- *DHCPSocket* : Permet de créer le *Socket* ainsi que les méthodes d'envoi et de réception.
- *DHCPOption* : Contient les méthodes pour la création des options du message *DHCP*
- *DHCPMessage* : Construction des messages *DHCP* contenant des méthodes d'édition.
- *PMIPReadConfig* : Rajouté par le diplômant, permet de lire dans un fichier la configuration de *DHCP* et met à disposition des méthodes d'accès au variables.

Default Package : *Package* dans lequel se trouvent les classes principales des programmes *DHCP* et *PMIP* Client.

- *PMIP_Client* : Classe principale qui contient le *Main* et met en adéquation le reste des classes entre elles.

- *PMIP_Worker* : Objet créé par client et stocké dans un tableau de hachage dans la classe *PMIP_Client*. Il contient une instance thread de la classe *PMIP_Instance*. Met à dispostion des méthodes de partage de variable pour que le *thread* puisse obtenir et fournir des informations.

- *PMIP_Instance* : *Thread* créé pour chaque client afin de gérer la négociation du tunnel entre le *FA* et le *HA*.

- *DHCP_Dispatcher* : Ecoute et partage le *Socket* sur le port 67. Aiguille les paquets reçus vers les différents *workers* stockés dans une table de hachage indexés par l'adresse *MAC* des messages entrants.

- *DHCP_Instance* : *Instance thread* créée pour chaque nouveau client. Reçoit les paquets de la classe *DHCP_Dispatcher*, les analyse puis renvoie leur correspondant depuis le socket partagé de cette dernière classe.

- *NAS* : Cette classe ne joue pas réellement le rôle d'un vrai *NAS*. Elle porte ce nom pour être conforme au *Draft* mais fait office de sniffer de protocole *AAA*.

- *MN_Data_Mobile* : Objet contenant les informations mobiles du client créé par le *NAS*

3.4. Diagramme UML des classes

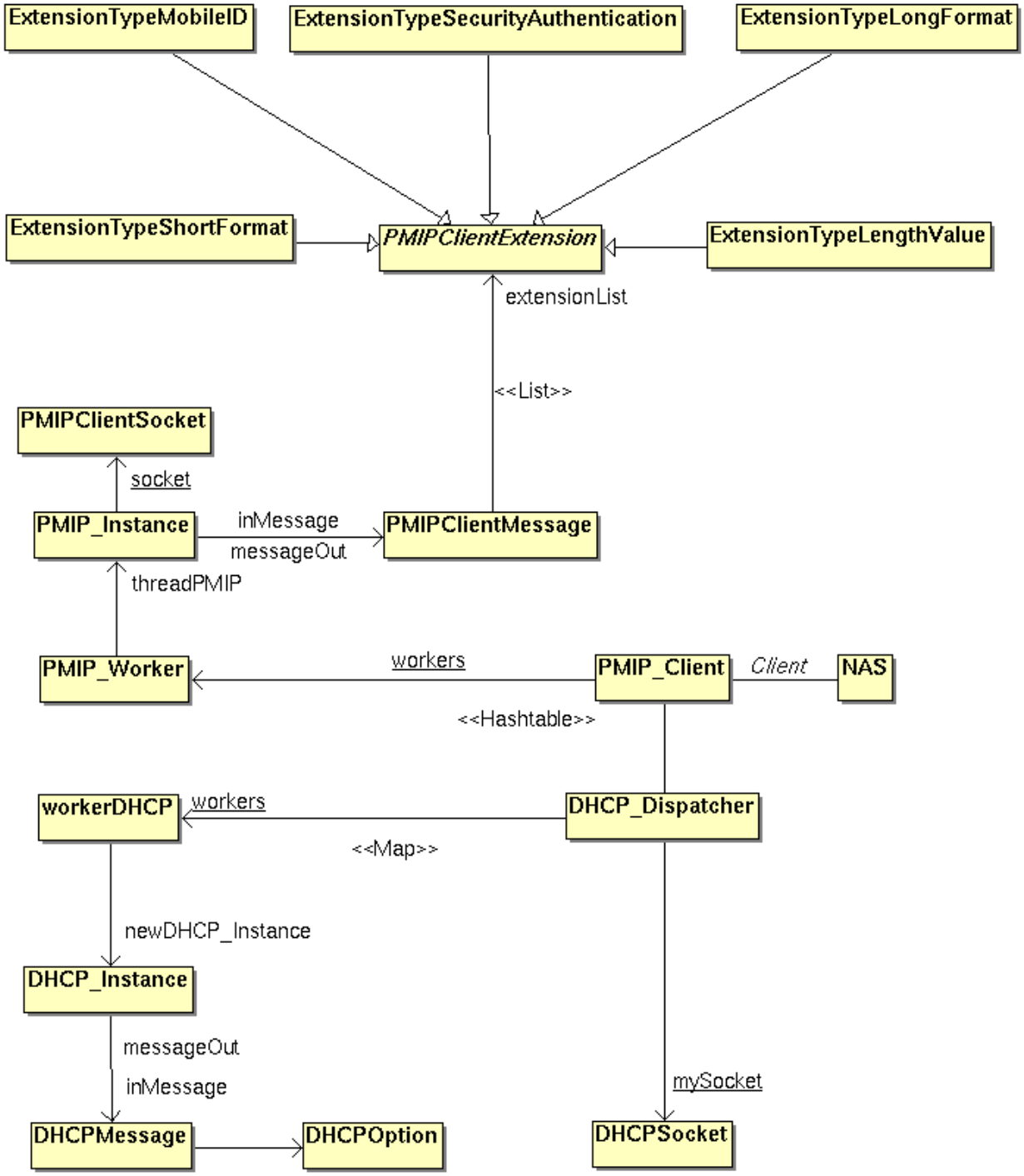


Figure 10 : UML des classes du DHCP

Seul trois classes ne sont pas représentées sur ce schéma UML. Les deux classes de lecture du fichier de configuration du DHCP et PMIP Client, ainsi que le Fixe du TTL.

3.5. Planning de développement

La planification doit se faire en tenant compte de deux dates importantes. La première est située à la fin de la sixième semaine, c'est-à-dire au milieu du travail de diplôme. Un premier document intermédiaire doit être rendu, lequel deviendra plus tard le rapport final. La deuxième est l'écheance finale du 15 décembre à midi où trois copies imprimées ainsi qu'une copie sur CD doivent être remises. Chaque semaine un rapport de travail hebdomadaire relatant le travail accompli doit être rendu le lundi.

Semaine 1
La première semaine est consacrée à l'installation et à la configuration du réseau *MIP* pour s'habituer au protocole. Il va évoluer pour atteindre la configuration requise en fin de cette semaine.

Semaine 2 (Lundi de congé)
Etude et développement du service *DHCP*. Cela va commencer par une lecture rapide du *RFC*, puis par la capture des messages *DHCP* depuis *Wireshark* sur le réseau de l'école. La deuxième étape est de créer des messages préfabriqués pour répondre aux requêtes d'un client.

Semaine 3
Fin du développement du service *DHCP* avec la gestion par *thread* de plusieurs clients simultanément. Tests de connexion avec plusieurs clients en même temps.

Début de programmation du *PMIP Client*. Pour commencer une étude des messages *MIP* avec *Wireshark* sera faite en comparaison avec la *RFC*. Ensuite, il faut créer des paquets préconstruits et tenter de discuter avec le *HA*. Si tout fontionne, il faut créer un système de *thread* pour plusieurs clients simultanés.

Semaine 4
Fin de la programmation du *PMIP Client* et accompagné par des tests. Il faut maintenant regrouper le service *DHCP* et le *PMIP Client.*

Semaine 5
Installation d'un service *AAA* simple avec mise en place et paramètrage d'un client. Lorsque la connection est réussie avec ce système d'authentification, il faut le joindre au *PMIP Client*. Le protocole devrait être fonctionnel en fin de cette semaine. Il devrait pouvoir assurer une mobilité simple qui sera améliorée par la suite.

Semaine 6
Rédaction du rapport intermédiaire qui doit contenir la structure du rapport final.

Semaine 7
Amélioration de la partie *PMIP Client* pour répondre à toutes les exigence du *Draft* et de la *RFC 3344*.

Semaine 8
Amélioration du service *DHCP* en ajoutant de service de *DHCP relay* afin de pouvoir redonner une adresse au client en cas de non obtention d'adresse mobile.

Semaine 9
Amélioration du système d'authentification *AAA* pour rendre le client totalement mobile (*AAAF* et *AAAH*).

Semaine 10

Le protocole devrait être opérationnel et doit pouvoir être testé. Les améliorations et études de tout les cas présentables dans le *Draft* doivent être effectués, comme par exemple le retour du client dans le réseau *Home*.

Semaine 11

Semaine tampon en cas de débordement du *timing*. Elle peut être consacrée à l'amélioration du protocole ou à la rédaction le rapport.

Semaine 12

Dernière semaine, consacrée uniquement à la rédaction du rapport final à rendre pour le lundi 15 décembre à midi.

C H A P I T R E 4

D H C P

4.1. Historique

Le protocole *DHCP* (*Dynamique Host Configuration Protocol*) dont la norme est définie dans la [RFC2131], a été inventé au début des années 90. Il est le successeur près de 10 ans plus tard de son aîné, le protocole *BOOTP*. Au début, l'adressage *IP* était prévu en fixe, mais rapidement avec l'évolution des réseaux cela est devenu ingérable. Il a donc fallu trouver un moyen pour donner dynamiquement une adresse *IP* aux clients.

4.2. Fonctionnement

Le fonctionnement est relativement simple, voir le *Time Line* (Figure 11 ci-dessous). Un client sans adresse qui arrive sur un réseau disposant d'un service *DHCP* envoie un message *DHCPDiscover* pour trouver le serveur. Ce dernier lui répond par un *DHCPOffer* contenant l'offre d'adressage *IP* que le client peut refuser. Si le client l'accepte, il renvoie un *DHCPRequest* au serveur pour confirmer sa demande. Le serveur répond pour terminer la transaction et pour valider l'adressage par un *DHCPAck*. Pour libérer la ressource il suffit au client d'envoyer un *DHCPRelease* au serveur.

Il existe aussi un système d'allocation de bail associé à un temps. Avant que ce dernier ne soit dépassé, le client redemande une prolongation d'adresse avec un *DHCPRequest*. En cas d'acceptation le serveur répondra par un *Ack* sinon par un *Nack*.

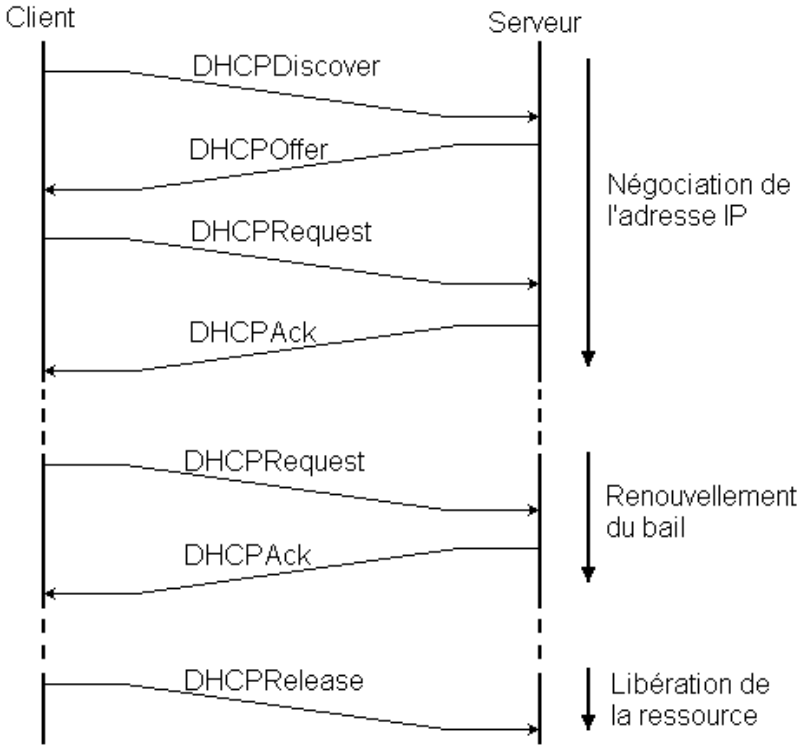


Figure 11 : Diagramme du protocole *DHCP*

4.3. Structure du message *DHCP*

Ce paquet *DHCP* (voir Figure 12) est encapsulé dans un paquet *TCP* lui-même dans un paquet *IP* qui sera ensuite émis sur le réseau. Le port du client est le 68 et celui du serveur le 67. Ce qui différenciera ces paquets c'est le champ *Xid* propre à la transaction (session) en cours ainsi que le champ *chaddr* qui contient toujours l'adresse *MAC* du client.

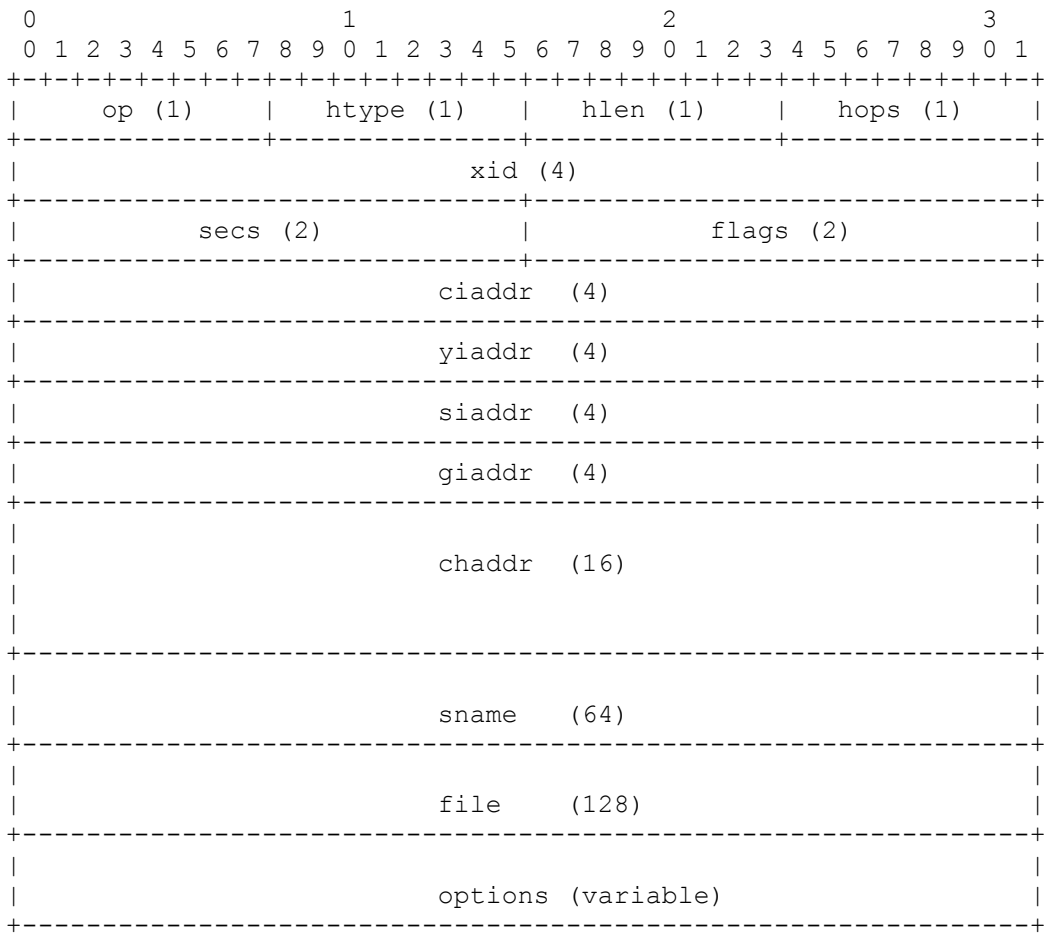


Figure 12 : Structure du message *DHCP*

- *Op* : Code du type de message (1 = *BOOTREQUEST* et 2 = *BOOTREPLY*)
- *Htype* : Type de l'adresse *Hardware* (1 = 10mb ethernet)
- *Hlen* : Longueur de l'adresse *Hardware* (6 pour 10mb ethernet)
- *Hops* : Mise à 0 par le client, incrémenté par le *DHCP relay agent*
- *Xid* : Numéro de transaction choisi par le client, permettant de créer une session entre le client et le serveur
- *Secs* : Temps écoulé depuis que le client a commencé la demande ou le renouvellement de l'adresse
- *Flags* : Actuellement setté à 0 pour de futures utilisations
- *Ciaddr* : Adresse du client. Champ rempli seulement pendant l'état de *Bound*, *Renew* ou *Rebinding*
- *Yiaddr* : Champ dans lequel l'adresse *IP* est proposée
- *Siaddr* : Adresse du prochain serveur à atteindre
- *Giaddr* : Adresse *IP* du *relay agent*
- *Chaddr* : Adresse *hardware* du client
- *Sname* : Nom du serveur *host* (en option)
- *File* : Nom du fichier de *Boot*
- *Option* : Champ de paramètres optionnels

4.4. Implémentation dans le *PMIPv4*

Le *DHCP* doit intimement discuter avec le *PMIP Client*. En effet, la relation entre les deux entités est présente à toutes les étapes de la mobilité. Lorsqu'un nouveau client se connecte, le *DHCP* avertit le *PMIP Client* de débuter une demande de mobilité. Après l'établissement du tunnel le *PMIP Client* doit redonner une adresse mobile au client. Lorsque ceci est terminé, le tunnel entre le *HA* et le *PMIP Client* doit être renouvelé périodiquement tant que le client est connecté. Pour ce faire, le client doit obtenir une adresse *IP* avec un temps de renouvellement plus petit que le temps du tunnel. Par cette méthode le client informe le *PMIP Client* de sa présence via le *DHCP*. Après vérification le *PMIP Client* peut renouveler le tunnel sachant que le client est toujours connecté.

Si le client obtient une adresse *IP* dans un *Foreign Network* (*FN*) puis se déplace dans un autre *FN*, le *DHCP* ne doit pas fonctionner de la même manière. Dans ce cas, le client possède déjà une *IP* mobile et ne demandera donc qu'un renouvellement de bail. Le *DHCP* acceptera seulement si la demande auprès du *HA* a été acceptée. Dans le cas où le *HA* rejette la demande de mobilité, le *DHCP* devra donner au client une adresse de sa propre *range* ou alors orienter la demande vers un autre serveur *DHCP*.

Le dernier cas prévoit que le *DHCP* au niveau du *FN* est un *DHCP Relay*. Dans ce cas, tout ce qui a été décrit plus haut reste identique à l'exception de la gestion de l'adressage qui se fait plus loin au niveau du *HN* où se trouve le serveur *DHCP*.

Récapitulation des points principaux à développer :

- Doit pouvoir gérer plusieurs clients *DHCP* en parallèle.
- Gestion des messages arrivant pour y répondre correctement.
- Informe le *PMIP Client* seulement si un *thread* de ce dernier est lancé. Différencier une première connection d'un client sans adresse avec un déplacement vers un autre réseau mobile.
- Reste bloqué en attendant l'adresse *IP* à fournir au client donné par le *PMIP Client*.
- Si l'attente est trop longue ou que la réponse du *PMIP Client* est négative, réhoriente vers un serveur *DHCP* pour obtenir une adresse locale.
- Le temp du renouvellement du bail doit être au minimum deux fois inférieur au temp de vie du tunnel

4.5. Développement

4.5.1. Gestion de connexions multiples simultanées

Comme vu précédemment dans la partie de génie logiciel, le *DHCP* utilise une structure particulière pour la gestion des clients à cause de son port d'écoute. La classe *DHCP_Dispatcher* ouvre un *socket* sur le port 67 puis rentre dans une boucle sans fin, *while* (*true*). A l'intérieur de celle-ci se trouve la méthode qui reçoit les messages *DHCP*. Ils sont analysés d'après leur adresse *MAC* pour savoir s'ils sont déjà répertoriés par l'index dans la table de hachage. Cette table doit contenir un *thread* par client connecté qui s'occupera suivant le message entrant de répondre correctement. Si c'est un nouvel arrivant, l'adresse *MAC* est inconnue. Un nouveau thread qui reçoit en paramètre le message entrant est alors créé dans la table de hachage. Le thread se termine immédiatement après avoir répondu au message. Si l'adresse *MAC* du message

entrant est connue, alors le message est passé à l'objet de la table de hachage au bon index. Ce dernier analyse le message pour y répondre puis termine le thread.

4.5.2. Gestion des messages entrants

La classe *Java* (*DHCP_Instance*) instanciée en objet lors de chaque nouveau client, n'est pas en permanence lancé en *thread*. Lorsque l'objet reçoit le message depuis le *Dispatcher*, il doit l'analyser par son champ *Code* qui indique la nature de la requête. Cette analyse est gérer par un *Switch* (*Code*), case *numéro_Code* :, qui lance en *thread* la bonne méthode pour répondre au message.

4.5.3. Informer le PMIP Client

Ce cas peut se présenter à deux occasions. D'une part lorsqu'un nouveau client sans adresse arrive sur le réseau et demande une adresse via un *DHCP Discover*. D'autre part lorsque le client possède déjà une adresse *IP* mobile obtenue avant son dernier déplacement. Il ne fait qu'un renouvelement par un *DHCP Request* après s'être rendu compte que la couche *L2* du réseau est revenue.

Dans le cas du *Discover* il faut vérifier si un *thread PMIP Client* est déjà lancé et est en attente du *DHCP*. Si c'est le cas le *PMIP Client* doit être informé de l'arrivée d'un nouveau client tout en lui communiquant l'adresse *MAC*. Il faut ensuite se placer dans un état bloquant en attente de sa part du renvoi de l'adresse *IP* à fournir au client. Si aucun thread n'est lancé, ou qu'une réponse négative revient du *PMIP Client*, il faut réorienter la demande vers un vrai serveur *DHCP*.

Dans le cas du *Request* il faut bien analyser la nature de ce message. En effet, il peut représenter le message *Request* normal de la première négociation lors d'un *DHCP Discover*. Dans ce cas il ne faut rien faire. Si le *Request* qui se présente contient déjà une adresse, il faut avertir le *PMIP Client* de lever le tunnel. En cas de réponse positive de ce dernier, le *DHCP* acceptera l'adresse *IP* demandée par le client. En cas de refus d'utilisation de cette adresse le *DHCP* renverra un *Nack* et le client devra recommencer toute la négociation depuis le début avec un *Discover*.

Pour ce faire, il faut utiliser un vérificateur *IF* (*condition*) afin de vérifier l'exactitude de la réponse a fournir. Dans certains cas, plusieurs vérificateurs peuvent se retrouver imbriqués.

4.5.4. Etat bloquant

La notion d'état bloquant fait appel à de la programation concurrente. En *Java* il est possible de créer par l'objet *CountDownLatch* un système de verrous. Il possède deux méthodes qui sont l'opposé l'une de l'autre. La première *Objet.await ()* permet de bloquer dans l'attente de la deuxième méthode *Objet.countDown ()*.

4.5.5. Réorientation vers un serveurs DHCP

Cette partie est à prendre en compte dans un second temps. De base, le protocole développé est capable de répondre aux requêtes de mobilité. Mais dans le cas où le protocole serait implémenté de manière fonctionnelle, il se peut que le client se voie refuser la mobilité. Pour cela, il faut que le *DHCP* oriente les paquets vers un autre serveur *DHCP*. Le draft explique que le *DHCP* peut se trouver sur le *PMA* ou au niveau du *HA*. Dans ce dernier cas, le *DHCP* au niveau du *PMA* sera un *relay*. Rien n'empêche dès ce moment qu'il fonctionne par les deux méthodes en même temps ! Si la connection mobile est impossible, il passe en mode *relay* et transmet la demande au serveur *DHCP* du réseau local. Toute la structure de gestion du *DHCP* est prête à accueillir cette solution, mais elle n'est pas encore implémentée.

4.5.6. Temps de renouvellement du bail DHCP

Le temps de renouvellement du bail doit être en relation avec le temps de renouvellement du tunnel. Après avoir été débloqué, le *DHCP* va récupérer l'adresse *IP* mobile du client au niveau du *PMIP Client*. Il en profitera aussi pour prendre le temps de vie du tunnel. Ce temps est coupé en deux avant d'être redonné au client comme temps de renouvellement.

4.6. Problèmes rencontrés

4.6.1. Choix de l'interface d'écoute.

Ce problème est lié à un problème *Java* ainsi que d'implémentation réseau. La machine sur lequel tourne le *PMA* peut n'avoir qu'une seule interface réseau. C'est par cette dernière que les paquets pour le *HA* sont envoyés mais aussi ceux pour le client. Le nouveau client envoit sa requete *DHCP* sur le réseau, mais c'est à ce moment que se pose le problème d'implémentation. Si il y a un serveur *DHCP* sur le réseau, il y a de fortes chances que le client reçoive une adresse bien avant le service *DHCP* du *PMA* fournissant l'adresse mobile. Il n'est pas possible non plus de bloquer mes requêtes sur le *DHCP* du réseau pour deux raisons. Premièrement, parce que le labo de test tourne sur le réseau et il n'est pas possible de me bloquer l'adresse. Deuxièmement, si le PMA ne m'autorise pas la mobilité c'est le serveur *DHCP* du réseau local qui doit fournir une adresse *IP*.

De ce fait, il a été décidé de placer deux intérfaces réseau sur la même machine. L'une configurée avec une adresse du réseau local et l'autre avec un réseau interne sur lequel sera branché la borne *Wireless* de mobilité. Maintenant que l'architecture physique est en place il faut créer le programme de service *DHCP* du *PMA*. Java offre pour la comunication *IP*/*UDP* deux classes qui héritent de la classe *Socket*. La première, *DatagramSocket* permet simplement la réception et l'envoi depuis un port fixe des paquets *UDP*. Elle ne dispose pas de méthode relative d'une association sur une interface spécifique. Les paquets sont émis et reçus sur les deux interfaces de la machine. De par ce problème, le service va aussi essayer de répondre aux demandes de clients sur le réseau bien qu'il ne diposent pas de la mobilité *PMIP*. La deuxième classe *MulticastSocket* permet de spécifier l'interface voulue. Par contre sa tâche est dévolue à l'envoi de paquets pour un groupe de discussion. Ces groupes sont compris dans une plage d'adresse de minimum 224.0.0.0 à maximum 239.255.255.255. Il n'est donc pas possible d'envoyer ni de recevoir des messages broadcast *DHCP* en 255.255.255.255.
Finalement, c'est la classe *DatagramSocket* qui est utilisée, bien qu'elle ne soit pas très optimisée pour ce travail. Ce sera un point à changer par la suite.

4.6.2. BOOTP -> DHCP

Les messages créés par le programme sont envoyés comme le protocole *BOOTP*. Pour corriger le problème, il suffit d'ajouter dans le champ "*Option*" en position 1, la nature du message (1 pour le *DHCPDiscover* , 2 : *DHCPOffer*, etc ..).

4.6.3. DHCP et Linux.

Le programme n'arrive pas à donner une adresse à *Linux*, contrairement à *Windows* qui l'accepte immédiatement. Le problème provient du champ option qui ne contient pas en position 53 de valeur *LeaseTime*. Par défaut *Windows* considère ce temps comme infini, mais *Linux* prend le protocole au mot et le met à 0. Ceci signifie pour lui que l'adresse est périmée et qu'il lui faut un nouveau bail. *Linux* et le serveur *DHCP* s'engagent alors dans une conversation de sourds entre ordinateurs. Pour corriger le problème, il faut ajouter le temps de *LeaseTime* à une valeur plus grande que celle du *renewal* et du *bind* en postion 53 du champ option.

C H A P I T R E 5

P M I P C L I E N T

5.1. Fonctionnement

Le *PMIP Client* est là pour simuler au niveau du réseau le client mobile *MN* du *MIP*. Sa fonction principale est de communiquer avec le *FA* comme le ferait le *MN*. Pour chaque nouveau client un *thread PMIP Client* doit être lancé avec en paramètre les informations du *MN* qui vient de se connecter auprès de *l'AAA*. En effet, dans le cas de la mobilité par *MIP*, le client conservait avec lui toutes ces données dans la configuration du *MN*. Avec le protocole *PMIP* le client est déchargé de tout programme additionnel ce qui complique légèrement la situation. Lors de la phase d'obtention de l'adresse *IP* mobile, le client communique son adresse *MAC* au *PMIP Client* par l'intermédiaire du *DHCP*. Le *PMIP Client* va débloquer le *thread* en attente associé à cette adresse *MAC*. Le *thread* doit établir le tunnel avec le *HA* puis retourner une adresse mobile pour le *DHCP*. Lorsque la mobilité est établie, le *PMIP Client* est en charge de renouveler le tunnel entre le *HA* et le *FA*. Pour cela il s'assure que le client est toujours présent avant de répondre au serveur.

5.2. Structure du message *PMIP Client* (*PRRQ* & *PRRP*)

Ces messages sont identiques à ceux utilisés dans le réseau *MIP*. La Figure 13 montre la structure des champs du message. La seule différence est le nom utilisé qui change du *MIP*.

5.2.1. Corps du message

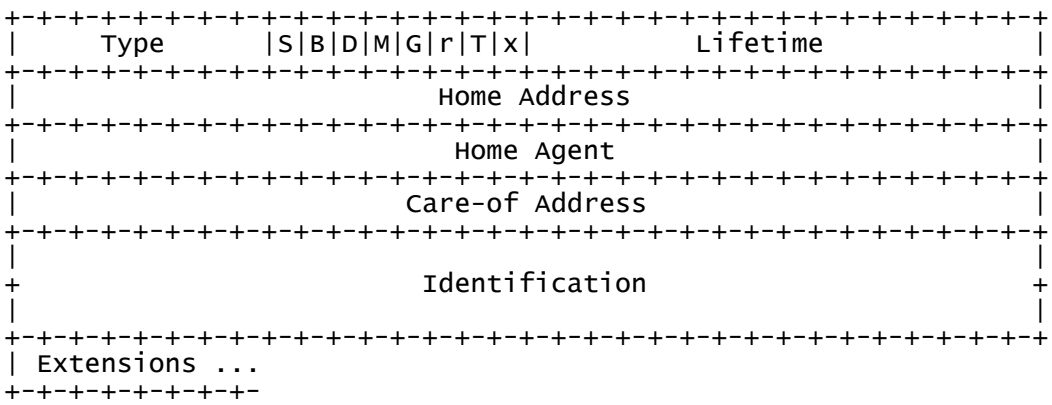


Figure 13 : Structure du message *PMIP Client* (*PRRQ* & *PRRP*)

- *Type* : Type de message (1 = *Registration Request* et 2 = *Registration Reply*)
- *S* : 1 = le *Mobile Node* demande au *HA* qu'il retienne les *binds* de mobilité
- *B* : 1 = le *MN* demande à HA de lui *forwarder* le *broadcast* via le tunnel
- *D* : 1 = le *MN* désencapsule le tunnel => *PMIP* toujours set à 0
- *M* : 1 = le *MN* demande une encapsulation minimale au *HA*
- *G* : 1 = Le *MN* demande que le *HA* utilise une encapsulation tunnel type *GRE*

- *R* : Envoi à 0 par le *MN*. Champ réservé pour le *HA*
- *T* : 1 = Demande de *Reverse Tunneling* auprès du *HA*.
- *X* : Envoi à 0 par le *MN.* Champ réservé pour le *HA*

- *LifeTime* : Nombre de secondes restantes avant que l'enregistrement soit considéré comme expiré. Une valeur de 0 indique une requête de désenregistrement et *0xffff* pour infinie.
- *Home Address* : Adresse *IP* du *Mobile Node*
- *Home Agent* : Adresse *IP* du *HA* associé au *MN*
- *Care of Address* : Adresse *IP* de la fin du tunnel (celle du *FA* pour le cas du *PMIP*)
- *Identification* : Construit par le *MN*. Il est utilisé pour faire correspondre les requêtes d'enregistrement et de réponse et pour protéger contre les *Replay Attacks*.
- *Extensions* : Partie englobant toute les extensions. Cette partie doit obligatoirement contenir une extension *Authentication Extension* dans toutes les requêtes d'enregistrement.

5.2.2. Extensions

Il existe deux familles d'extensions, les *NonSkippable* et *Skippable*. Elles sont différenciées respectivement par le champ type de 0 à 127 et de 128 à 255. S'il n'est pas reconnu, il doit respectivement ignorer le message ou ignorer l'extension.

Type Authentication Extension

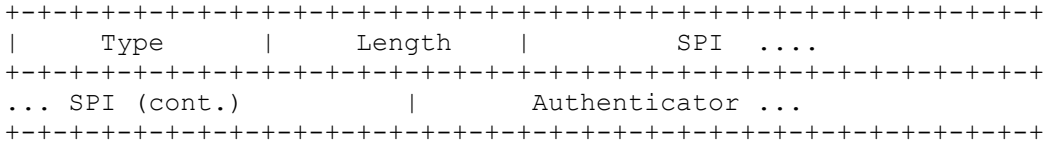


Figure 14 : Extension Authentication

- *Type* : 32
- *Length* : Longueur du paquet (4 plus le nombre de *byte* de l'authentificateur)
- *SPI* : *Security Parameter Index* (*SPI*). Identifiant unique au *MN*. Ce dernier et le *HA* l'utilisent comme identifiant permettant une sécurité accrue de la signature du message
- *Authenticator* : Champ contenant la signature par défaut en *HMAC-MD5* du corps du message plus les authentifications qui le précèdent plus ses propres données

Type-Length-Value Extension

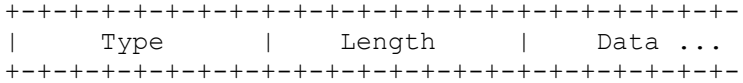


Figure 15 : Extension *Length-Value*

- *Type* : Indique le type de l'extension
- *Length* : Indique la longueur du champ *Data*
- *Data* : Champ contenant les données

Type Long Extension

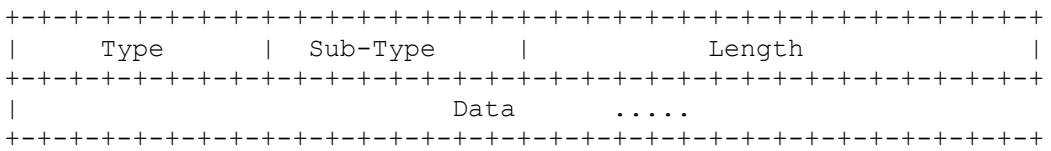


Figure 16 : Extension *Long*

- *Type* : Indique le type de l'extension
- *Sub-Type* : Numéro unique qui donne à chaque membre un type personnel
- *Length* : Indique la longueur du champ *Data*
- *Data* : Champ contenant les données

Type Short Extension

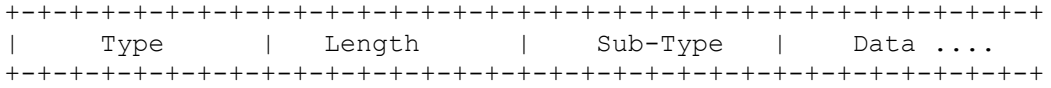


Figure 17 : Extension *Short*

- *Type* : Indique le type de l'extension
- *Length* : Indique la longueur du champ *Data*
- *Sub-Type* : Numéro unique qui donne à chaque membre un type personnel
- *Data* : Champ contenant les données

Type Proxy Mobile IPv4 Device ID Extension

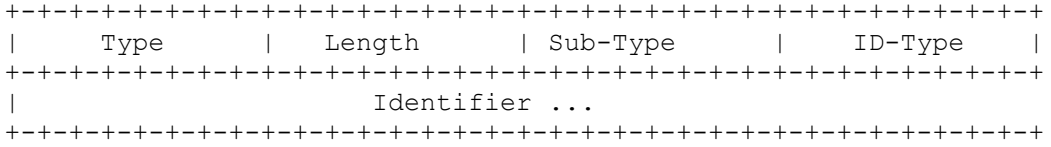


Figure 18 : Extension *PMIP ID*

- *Type* : 147
- *Length* : Longueur en *byte* de *Sub-type* plus *ID-Type* plus *Identifier*
- *Sub-Type* : Spécifie la nature du message
- *ID-Type* : Spécifie le contenu de l'identifiant
- *Identifier* : Identifiant du champ *ID-type*

5.3. Implémentation dans le *PMIPv4*.

Le *PMIP Client* simule le client *MIP* auprès du *FA*. Pour chaque client connecté, un *thread* d'instance *PMIP Client* est lancé. Lors de son lancement, le *thread* ouvre un nouveau *socket UDP* sur un port libre qu'il gardera jusqu'à la fin de la transaction. Les messages envoyés sont de même facture que ceux du *MN* du *MIP*. Le *Draft* spécifie néanmoins que la variable *D* du champ de désencapsulation est obligatoirement à 0. Ceci implique que l'adresse *CoA* de désencapsulation du tunnel n'est jamais celle du *Client* mais du *FA*.

Après avoir recu les informations du client par le serveur *AAA* via le *NAS*, le *thread* doit se lancer et se bloque en attendant l'arrivée du client par le *DHCP*. Aussitôt débloqué, il établit la connexion avec le *HA* et la maintient jusqu'à ce que le *Client* se déconnecte ou change de réseau. Dans ce dernier cas, le *PMIP Client* et le *DHCP* doivent effacer toutes les données, fermer *threads* et *sockets* relatifs au client. Lors du renouvellement du tunnel le *PMIP* doit envoyer des messages selon une période de temps définie lors de l'établissement du tunnel.

Lors de la réception des messages *PRRP* provenant du serveur, le *thread* doit impérativement vérifier la validité du champ d'authentification. Pour vérifier cette information il doit reprendre tout le contenu du message, sauf le dernier champ de l'extension *Authentication* et recréer une signature avec clé secrète (*SecretKey*) du client. Si les deux signatures correspondent, c'est que le message est intact et n'a pas été modifié. Dans le cas contraire, le *PMIP Client* doit silencieusement rejeter le message. Après cela, il doit encore vérifier le champ d'identification contenant l'heure du serveur au format *TimeStamp*. Si l'heure diffère trop par rapport à l'heure actuelle du client (entre 3 et 7 secondes), ce dernier doit réajuster son horloge. Après toutes ces vérifications il peut analyser le contenu du message d'après le champ type.

Le *Draft* spécifie au niveau du *HA* un temps obligatoire d'une seconde entre l'envoi et la réception d'un message. Le *PMIP Client* ne doit donc pas répondre trop vite et attend une seconde avant de répondre au message. Lorsque le tunnel est paramétré en *Reverse Tunneling*, la *RFC* [RFC2344] spécifie que le *MN* doit impérativement envoyer ses messages *IP* avec un *TTL* (*Time to Live*) à 255. Cette spécification sert à diminuer le risque d'attaque par déni de service par des hôtes malveillants envoyant des messages d'enregistrement au *HA*.

Récapitulation des points principaux à développer :

- Le *PMIP* doit pouvoir gérer plusieurs clients mobiles simultanément.
- Passer des paramètres entre *threads*.
- Doit connaître les informations du mobile en local.
- Les messages envoyés doivent contenir le champ *D* de désencapsulation à 0.
- Doit rester bloqué en attendant l'arrivé du client via le *DHCP*.
- Doit fermer le *thread* pour libérer de la mémoire si le client ne se présente pas.
- A la fin de la mobilité libérer toutes les ressources.
- Vérifier la validité des messages avec la signature du message.
- Vérifier le champ *Identification* contenant le *TimeStamp* du serveur et le modifier si besoin.
- Cryptage de la signature en *Hmac MD5*.
- Temps d'une seconde entre l'envoi et la réception d'un message.
- Setter le *TTL* à 255 dans le *header IP* du message.

5.4. Développement

5.4.1. Gestion de connexions multiples simultanées

Après avoir reçu les informations d'un nouveau client mobile, le *PMIP Client* crée un nouvel objet (*Worker*) contenant l'instance *thread* liée au nouveau client. L'objet *Worker* est indexé par l'adresse *MAC* du client dans une table de hachage. L'utilité de passer par cette méthode au lieu de placer directement le *thread* indexé dans la table, est que certains paramètres peuvent l'accompagner. Par exemple, si l'objet contient le *thread* ainsi qu'une variable, ils pourront être indexés les deux par la même adresse *MAC* dans le même tableau. Ceci permet au *thread* de pouvoir récupérer des informations ou d'en déposer pour d'autres. Il est important de dire que si plusieurs *threads* peuvent avoir accès à cette variable, il faut la synchroniser pour éviter des problèmes de concurrence.

Maintenant que le *thread* est lancé, il doit créer une connection *UDP* sur un port aléatoire qu'il gardera tout le temps de sa négociation avec le *HA*. Il n'y a donc pas besoin de se préoccuper des autres *threads PMIP Client* car les messages connaissent leur destinataire par le port source d'envoi.

5.4.2. Passage de paramètres entre threads

En *Java* un *thread* en cours d'éxecution (*run*) ne peut pas recevoir d'information. Il est possible par contre de lui transmettre des informations lors de son instanciation. Un *thread* en cours d'activité peut néanmoins accéder à la mémoire commune de l'ordinateur pour y récupérer des informations. Pour cela, il peut faire appel à une méthode d'une autre classe afin lui fournir les données dont il a besoin.

Un problème intervient quand le *thread* n'est pas le seul à accéder à ces données. Par exemple un autre programme peut aller placer des informations que le *thread* vient lire. Comme ces données sont partagées entre plusieurs *threads*, il faut introduire la notion de programmation concurrente pour que la ressource commune reste accessible sans créer de problème. Les méthodes qui permettent d'écrire ou de lire l'information commune entre plusieurs *threads* doivent être synchronisées (*synchronized*). Cette synchronisation donne la priorité au premier *thread* qui se présente puis la redonne en libèrant la ressource lorsque ce dernier a terminé.

Lorsqu'un *thread* doit attendre une information provenant d'un autre *thread*, il est possible de le mettre en attente jusqu'à ce que l'autre *thread* le débloque après avoir modifié la valeur commune. Ainsi, nous sommes sûrs que le premier *thread* peut lire l'information après sa modification. Ce système d'attente peut se faire en *Java* par l'utilisation de *CountDownLatch* qui offre une sorte de système de verrous.

La classe *PMIP_Worker* a été créée pour contenir les informations liées au client ainsi que le *thread PMIP Client* instancié. Le worker contient trois variables ainsi que des méthodes *synchronized* pour leur lecture et écriture :

- *MAC* : *String* utilisé dans tout le programme pour tracer un client
- *stateOfMN* : *Boolean* qui sert à connaître la présence du client
- *MNHomeIPAddress* : Tableau d'octet contenant l'adresse *IP mobile v4* à retourner au client

5.4.3. Obtention et gestion des données du client mobile.

Lors de l'arrivée d'un nouveau client sur le réseau, le *NAS* avertit le *PMIP Client* en lui fournissant les paramètres mobiles du client sous forme d'objet. Le *PMIP Client* crée un *Worker* en lui passant à son tour les informations recues puis l'indexe dans la table de hachage. Le *Worker* fraichement créé transmet l'objet contenant les informations au nouveau *thread*. Ce dernier attend l'arrivée du client par le *DHCP* pour continuer la négociation du tunnel dont il connaît maintenant toutes les informations.

5.4.4. Désencapsulation tunnel au niveau du FA

C'est le client qui lors de ces premiers messages d'authentification auprès du *HA* spécifie qui de lui ou du *FA* désencapsule le tunnel. Pour cela le *bit D* du champ code au début du message doit être à 0 pour spécifier la désencapsulation au niveau du *FA*. Le *Draft* spécifie par un Must l'importance de ce *bit*. Pour éviter tout problème lors de l'envoi du message, la méthode transformant l'objet message en tableau de *byte* force via un masque le *bit* à 0.

5.4.5. Attente du thread

Lorsque le *thread* est lancé par le *NAS* lors de l'arrivée d'un client sur le réseau, il doit dormir le temps que le client se présente par le *DHCP*. Si cela ne vient à jamais se produire il faut que le *thread* se termine pour libérer de la mémoire après un temps d'attente défini.

Pour résoudre ce problème, il existe plusieurs solutions différentes comme par exemple le *CountDownLatch*, l'utilisation native des méthodes *wait* et *notify* ou alors la combinaison des méthodes *sleep* et *interrupt*. Les deux premiers cas n'intègrent pas une notion de temps, il faut pour cela créer un *thread Timer* supplémentaire qui attend avant de débloquer le *thread* mis en attente. Cette programmation est bonne et robuste mais reste lourde au niveau de l'implémentation. Le problème peut être résolu plus simplement grâce à l'utilisation de la méthode *Sleep*(*Sec*) placée dans le *thread* qui doit être endormi. Cette méthode soulève une exception lorsque elle est interrompue par la méthode *interrupted*().

Bien que moins robuste que les deux solutions précédentes, elle permet néanmoins dans ce cas précis de se montrer utilisable et fiable. En effet, l'exception levée après l'interruption est incapable de savoir qui du système ou du programme vient de la provoquer. Mais, dans ce cas précis, cela n'a absolument aucune importance car après l'interruption le *thread* va lire une valeur *Boolean* contenue dans la mémoire afin de savoir si l'arrêt est volontaire ou non. Cette valeur *Boolean* est traitée dans un sélecteur *IF* qui continue le programme en cas de valeur positive ou bien termine le *thread*. Cette solution reste robuste et elle est surtout bien plus légère à implémenter. Elle lève cependant un problème mineur lors de la programmation qui doit être spécifié. Comme l'auteur de l'interruption de la méthode *Sleep* ne peut pas être authentifiée, le programme doit impérativement lire une variable *Boolean* pour savoir si l'arrêt est intentionnel ou non. Il faut pour cela modifier la valeur de cette variable *Boolean* avant de demander l'interruption du *thread* via la méthode *interrupted*(). Cela aura pour effet de prouver la volonté de l'utilisateur d'interrompre ce *sleep*() ;

5.4.6. Libération des ressources

Pour libérer ses ressources au niveau du *PMIP Client*, le *thread* peut faire appel à une méthode de la classe *PMIP Client* avant de se terminer. En lui donnant l'adresse *MAC* en paramètre, celle-ci permet de suprimmer suivant l'index le contenu dans la table de hachage. Après cela, le *thread* ne disposant de plus aucun point d'attache avec le reste du programme, peut se terminer normalement.

5.4.7. Vérification de la signature du message

Les messages entrants provenant du *HA* doivent être hachés en *md5* dans leur entier mais sans la partie du champ authentification contenant la signature. Cette nouvelle signature obtenue sera comparée à celle du message original. En cas de non concordance le message doit être ignoré.

5.4.8. Cryptage de la signature en Hmac MD5

Pour utiliser l'encryption *MD5* en *Java* il faut impérativement importer les librairies contenues dans `javax.crypto.*`. La clé secrète (*SecretKey*) doit être de 128 bits et peut être au format *HEX* ou *UTF8*. Le premier caractère (*UTF8*) du premier nombre (*HEX*) est stocké dans un tableau de 16 octets à partir de l'octet de poids le plus fort. Si la clé est plus petite que 128 *bits*, le reste du tableau sera comblé par des 0. Si la taille dépasse 128 bits le message sera tout simplement tronqué et seul le début sera gardé comme clé.

Il faut maintenant créer un objet de type *SecretKey* qui contient le tableau d'octets avec la clé ainsi que le type d'encryption souhaité.

```
SecretKey key = new SecretKeySpec(passwordInBytes, "HmacMD5");
```

Ensuite il faut créer le *MAC* avec cet objet qui contient la clé secrète hachée (*key*).

```
Mac mac = Mac.getInstance(key.getAlgorithm());
```

Il ne reste plus qu'à crypter les données contenues dans un tableau d'octet pour les sécuriser. Pour cela il suffit de passer ce tableau en paramètre à la méthode *doFinal* de l'objet *MAC* et de récupérer le résultat sous forme de tableau d'octets.

```
TableauDeByte = mac.doFinal(DonneeDansTableauDeBytes);
```

La variable *TableauDeByte* contient la signature de notre tableau de données.

5.4.9. Vérification du champ identification

Ce champ contient l'heure locale de la machine en format *TimeStamp*. Lors de l'envoi du premier message au *HA* l'heure locale du client (le *PMIP Client* dans notre cas) est placée dans le champ identification. Si l'heure du *HA* ne correspond pas avec celle du message reçu, ce dernier renvoie un code 133 contenant son heure locale. Le client n'a d'autre choix que de corriger son heure en interne ou d'ajouter une valeur *d'offset* lors de l'envoi des futurs messages.

5.4.10. Temps d'attente entre les messages

Cette partie peut être utilisée pour pour éviter que le *HA* ne reçoive de message moins d'une seconde après son propre envoi. Elle peut aussi être adaptée pour le temps d'attente avant de renouveler le tunnel.

Pour éviter tout problème d'interruption du *thread PMIP Client*, il faut créer une routine qui patiente le temps voulu. Après ce temps, elle va redonner la main en débloquant le *thread PMIP Client* (voir Figure 19). Pour ce faire, il faut créer un objet *CountdownLatch* dans le *thread PMIP Client*. Il est ensuite passé en paramètre à un objet préfabriqué contenant ce dernier plus un *thread* actif d'attente. Ce *thread* lancé va attendre le temps nécessaire via la méthode *native sleep* puis va redonner la main au *thread PMIP Client*. Ainsi, après le *sleep* il exécutera la méthode *countDown ()* de l'objet *CountdownLatch* passé en paramètre. Ceci débloquera l'instruction bloquante du *thread PMIP Client* placé juste après le démarrage du *thread* d'attente.

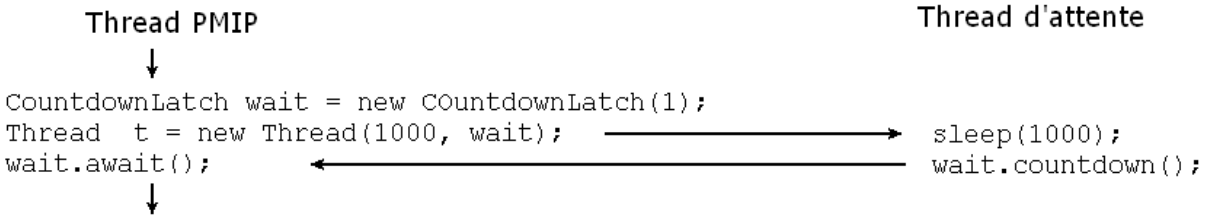


Figure 19 : *Thread* d'attente

5.4.11. Fixation du TTL à 255 dans le header IP

Comme les messages *PMIP* sont envoyés sur de *l'UDP*, seules les deux classes *Java* citées précédemment dans le chapitre *DHCP* peuvent convenir. La première, le *DatagramSocket* n'offre aucun possibilité pour changer ce paramètre. La seconde, en revanche permet sa modification, mais une fois de plus ce *socket* n'est pas fait pour de *l'unicast* de message. La vision de la plage de *broadcast* étant trop serrée il n'est pas possible d'envoyer ces messages. Il faut donc se résigner en attendant de trouver une autre solution que la modification du *TTL* par défaut du Système d'exploitation.

5.5. Problèmes rencontrés

5.5.1. Erreur Code 76

Lors du premier envoi de paquet fait par le programme développé, le *HA* a répondu par un code d'erreur 76 : *The Mobile Node is Too Far* [RFC3024]. Ce message provient du fait que le *header IP* ne contient pas le bon *TTL* qui devrait être de 255. En effet, par défaut *Linux* possèdent un *TTL* de 64 et *Windows* de 128. Pour changer ce *TTL*, aucune classe *Java* du *package 6.0* de base n'offre cette option sauf *MulticastSocket*. Malheureusement, ce dernier n'est pas fait pour et ne permet pas d'être utilisé dans notre cas. Pour résoudre ce problème temporairement avant de trouver une autre solution, il est possible de changer directement le *TTL* par défaut du système d'exploitation.

Dans le cas du programme actuel, un fixe est fait pour *Linux*. Une classe détect *l'OS* utilisé et exécute la ligne de commande correspondante depuis un fichier *bash*. Par exemple pour *Linux* : `echo '255' > /proc/sys/net/ipv4/ip_default_ttl`

Pour *Windows XP* il n'y a pas de moyen via *Java* de changer ce paramètre. Il faut le changer manuellement dans la base de registre en y ajoutant une règle *Dword* à la valeur *0xFF* (255 en décimal). Voici le lien de la base de registre auqel il faut ajouter la règle : HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters

5.5.2. Erreur Code 133

Ce champ de 8 octets contient le temps *TimeStamp UTC* [RFC1305] en fractions de secondes depuis le 1er Janvier 1970 à 00h00. Il est utilisé pour faire le lien entre les messages du *PMIP Client* et du *HA* par les 4 octets de poids faible. Il a aussi une fonction de sécurité, pour la protection contre les *replay attacks*. Les quatre premiers octets de poids fort possèdent les secondes. Les quatre derniers de poids faible servent aux fractions de secondes. Bien que *Linux* et *Windows* ne descendent pas en dessous de la microseconde, le reste des *Bits* de poids faible sont complétés par un chiffre généré par un *random* de bonne qualité.

Le problème est que *Java* commence son *TimeStamp* le premier janvier 1970 et le *HA* possède lui une référence à partir du premier janvier 1900. Il suffit de comparer les

deux *TimeStamp* et de créer une valeur de décalage qui sera ajoutée dans les futurs messages. Voici comment cela a été résolu :

Il faut en premier lieu récupérer la date actuelle du système d'exploitation.

```
Date date = new Date(new java.util.Date().getTime());
```

Puis la transformer au format *TimeStamp*

```
timeStampDate = new Timestamp(date.getTime());
```

Il faut séparer les secondes des fractions de secondes

```
Long seconds = timeStampDate.getTime();
seconds /= 1000;
int fractioOfSeconds = timeStampDate.getNanos();
```

Et ajouter le nombre aléatoire dans les bits de poids faible

```
fractioOfSeconds += (int)(Math.random()*1000000);
```

Il ne reste plus qu'à répartir les deux variables dans les 4 octets de poids fort et de poids faible du champ *Identification*.

5.6. MUST

Chaque *Must* est traduit en francais du *Draft PMIP* ou du *RFC 3344* et accompagné de la référence. Si certain *Must* sont manquants par rapport au *RFC* du *MIP* c'est qu'ils ne touchent pas directement le module *PMIP Client* à développer ou qu'il ne sont pas encore implémentés.

5.6.1. Sécurité

- ✓ Chaque PRRQ doit être protégé par une extension d'authentification (signature).
  Draft PMIP, Chap. 4.1.1 & 3.2 & 5.1

- ✓ Chaque extension d'authentification doit contenir un *SPI*.
  RFC 3344, Chap. 1.6

- Le valeur 0 et 255 pour le *SPI* ne doivent jamais être utilisés dans les messages d'association.
  RFC 3344, Chap. 1.6
  => Le programme ne vérifie pas cet aspect, car l'utilisateur doit être conscient de ce qu'il définit comme *SPI*.

5.6.2. Gestion des extensions du message

- ✓ Les messages avec extensions de type 0 à 127 non reconnus sont considérés comme "*skippable*" et doivent être silencieusement mis de côté. Pour un *Type* non reconnu de 127 à 255, le message est "*non skippable*" et doit être conservé en ignorant l'extension.
  RFC 3344, Chap 1.8

- ✓ Le champ *length* de l'extension *Short format* doit être évalué par la longueur en *byte* des informations dans le champ *data*, + 1.

- ✓ L'extension de sécurité doit être la dernière à être placée à la fin du message après les autres extensions, pour pouvoir englober tout le message dans sa signature.
  RFC 3344, Chap 3.4

- ✓ L'extension de sécurité doit englober dans sa signature le *header* du message *PMIP*, toute les extensions du message avant lui ainsi que ses propres données (Type, Length et le SPI).
  RFC 3344, Chap 3.5.1

- ✓ L'encryption par défaut de la signature doit être faite en *Hmac-MD5*.
  RFC 3344, Chap 3.5.1

5.6.3. Envoi et réception des messages

- ✓ Le *MN* (dans notre cas le *PMIP Client*) ne doit pas envoyer plus d'un message par seconde lors de la négociation.
  RFC 3344, Chap 2.4.2 & 3.6.3

- Le *MN* (dans notre cas le *PMIP Client*) doit vérifier le champ *checksum UDP* du message entrant et écarter silencieusement le message s'il n'est pas correct.
  RFC 3344, Chap 3.1 & 3.6.2.1
  => Impossible avec les *sockets* proposés par le *package* de base *Java*. Une librairie externe nommé *jPCAP* serait en mesure de résoudre ce problème.

- Si les 32 *bits* de poids faible du message de réponse ne correspondent pas au message envoyé, il doit être écarté silencieusement s'il n'est pas correct.
  RFC 3344, Chap 3.6.2.1
  => Pas encore implémenté ! Ce cas ne s'est pas encore posé, car ce test n'a pas eu lieu.

- ✓ Le *MN* (dans notre cas le *PMIP Client*) doit ajuster son *TimeStamp* mis dans les messages pour répondre au *HA*.
  RFC 3344, Chap 3.6.2

- ✓ Si le *MN* (dans notre cas le *PMIP Client*) ne possède pas de *home address*, *l'IP* source doit être 0.0.0.0, sinon l'adresse *IP* source doit être la *home address*.
  RFC 3344, Chap 3.6.1.1

- Le *MN* (dans notre cas le *PMIP Client*) ne doit pas changer les *bits M* et *G* du champ code si il ne désencapsule pas le tunnel (ce qui est le cas du *PMIP*).
  RFC 3344, Chap 3.6.1.2
  => Pas encore implémenté. Peut être fixé avec un masque comme pour le bits *D* mais actuellement ces *bits* sont par défaut à 0.

5.6.4. Mutliples accès

- ✓ Le *PMA* doit être à même de pouvoir *forwarder* tous les paquets du client dont il est en train de s'occuper.
  Draft PMIP, Chap. 4.2.1

5.6.5. Renouvellement du tunnel

- ✓ Le *PMA* doit s'assurer que le client est toujours présent avant de renouveler le tunnel.
  Draft PMIP, Chap. 4.4

- ✓ Le *lifetime* du tunnel établi par le client durant les requêtes doit être utilisé.
  RFC 3344, Chap 3.4

- Si le client reçoit dans les messages du *HA* lors de l'établissement du tunnel, un *lifetime* plus petit que celui proposé, il doit utiliser le temps donné par le *HA*.
  RFC 3344, Chap 3.4
  => Pas encore implémenté ! Ce cas ne s'est pas encore posé car ce test n'a pas eu lieu.

- Si le client reçoit dans les messages du *HA* lors de l'établissement du tunnel, un *lifetime* plus grand que celui proposé, il doit l'ignorer et prendre le sien.
  RFC 3344, Chap 3.4
  => Pas encore implémenté ! Ce cas ne s'est pas encore posé car ce test n'a pas eu lieu.

- Si le *lifetime* est de 0 dans les messages provenants du *HA* c'est que ce dernier désire résilier la connection. Si le *lifetime* est de *0xFFFF* le temps du tunnel est considéré comme infini.
  RFC 3344, Chap 2.1.1
  => Pas encore implémenté !

C H A P I T R E 6

N A S

6.1. Fonctionnement

Le *NAS* utilisé dans le cadre du protocole *PMIP* signifie *Network Acces Server* ce qui n'est pas à confondre avec le *NAS* ou *SAN* signifiant aussi *Network Attached Storage*. Il apparaît comme une sorte de première porte d'entrée pour le client sur le réseau, d'où le nom de : Serveur d'accès réseau. Lors de la connection du client, il est sensé, suivant les requêtes, rediriger les informations vers un élément interne du réseau. Parfois, ces données peuvent être encapsulées pour être transportées via un autre protocole. Il permet donc au client d'outrepasser provisoirement la frontière du réseau pour atteindre un élément interne, ce qui est le cas pour le service *AAA*. Dans ce dernier cas, le *NAS* transmet dans le réseau les trames du protocole *EAP* provenant du client en les encapsulant dans le protocole *AAA* utilisé.

6.2. Implémentation dans le *PMIPv4* d'après le *Draft*

La raison de l'implémentation d'un système *AAA* pour ce protocole n'est pas directement liée à la sécurité réseau. Dans le cas du protocole *MIP*, les données se trouvent dans le fichier de configuration du *MN*. Maintenant, le *PMIP* qui est l'élément responsable de la connection mobile du client ne connaît plus ses informations. En effet, comme il fonctionne au niveau du réseau, il doit se débrouiller pour trouver les informations mobiles du client. De par ses caractéristiques, le *NAS* est le premier élément à voir le nouveau client arriver sur le réseau.

En échangeant les données de connection du client avec le serveur *AAA* il est à même de prélever des informations. Après avoir authentifié le client, le serveur *AAA* peut sans autre lui retourner des informations depuis sa base de données. Ces dernières peuvent contenir l'adresse du *HA* à contacter, les paramètres de négociation du tunnel ainsi que d'autres paramètres. Après avoir récupéré toutes les données du client, le *NAS* va avertir le *PMA* d'un nouvel arrivant sur le réseau. Ainsi, il profite de lui donner toutes les informations nécessaires.

6.3. Implémentation du *NAS* dans le projet

Tous les routeurs *Wireless* proposant d'utiliser une protection par *AAA* possèdent un *NAS* interne. Le problème est que ce *NAS* est basique et n'est pas prévu pour la mobilité. Il n'est donc pas possible de le paramétrer pour lui demander de capter des informations afin de les donner au *PMIP Client*. Il faut créer un *NAS* qui offre cette option mais la solution n'est pas simple. Le *NAS* écoute les trames *EAP* provenant du réseau *Wireless*, les encapsule dans le protocole *AAA* choisi et lui-même empaqueté dans une trame *IP*.

Il existe trois solutions pour résoudre le système. A la base le *PMA*, ou du moins sa partie *PMIP Client*, est sensé fonctionner sur un routeur *Wireless*. La première solution serait de développer un *NAS* en language *C* et de l'installer sur une borne *Wireless* installée avec un système d'exploitation *Linux*. Bien que *Java* peu tourner sur cette plateforme, le language *C* de plus bas niveau se prête plus facilement au travail à accomplir. La deuxième solution, bien que lourde de conception, est de développer un serveur *proxy AAA*. Il fait croire au client qu'il est le serveur et fait croire au serveur

qu'il est le client. Mais cela requiert beaucoup de temps pour un résultat relativement moyen. Néamoins, il existe des libraires *Java open source* permettant de programmer ce système. La troisème solution est celle qui a été retenue par manque de temps. C'est l'utilisation d'un *Sniffer* de paquet réseau qui détecte les messages *AAA* d'autorisation d'accès au réseau, et qui va en avertir le *PMIP Client*. Ce paquet transmis par le serveur *AAA* possède un nom d'identifiant du client qui passe en clair. Cette méthode permet de respecter le *Draft* et rendre le protocole fonctionnel avant de pouvoir faire mieux.

6.4. Développement

Le protocole *AAA* conseillé pour faire de la *proxy Mobile IP* est *Diameter*. L'avantage qu'il apporte par rapport à ses prédécesseurs est qu'il possède une notion native pour la mobilité. Malheureusement, le temps mis à disposition pour installer et paramètrer un serveur utilisant *Diameter* a été outrepassé lors de problèmes d'installations. Le *Draft* du *PMIP* stipule (page 10, ligne 19, chap 4.1.1) clairement qu'il est possible d'utiliser une autre méthode de notre choix pour fournir les données. L'utilisation d'un *Sniffer* peut résoudre le problème en analysant des trames *Radius* au lieu de *Diameter* par exemple. Plusieurs étudiants font leur travail de diplôme sur un code *open source* nommé *FreeRADIUS*. Il est possible de raccorder un de leur serveur au réseau de test *PMIP* et d'analyser les trames passées.

Java ne fournit pas de base dans ces librairies de classe qui permettent de *sniffer* le réseau. Il faut pour cela télécharger le *package jPCAP* [JPCAP]. Pour fonctionner sur *Windows* ce dernier nécessite la libraire *WinPCAP* et sous *Linux*, *LibPCAP*. Ce *package* permet de combler des manques dans les librairies *JDK* de *Java*. Il permet de créer un objet qui écoute en mode *promiscuous* sur un interface et un port précis. Il faut lui passer en paramètre le temps de rafraichissement de ces captures.

Créer le capteur *jPCAP* sur l'interface choisi

```
JpcapCaptor c = JpcapCaptor.openDevice (interface, taille des
                paquets, promiscuous, temp de rafraichissement);
```

Filtre pour ne prendre que les paquets à destination du port du *AAA*

```
c.setFilter(Numéro du port, true);
```

Récupère les données et les place dans un objet *packet UDP*

```
UDPPacket packet = (UDPPacket) c.getPacket();
```

Après avoir *sniffé* la trame d'acceptation *Radius*, il faut en extraire le nom du client en clair. Le champ contenant le nom se trouve toujours en premier dans les options du message. Pour le récupérer il suffit de compter le nombre d'octets fixes avant l'information afin d'y accéder directement. Maintenant que le nom est récupéré, il faut authentifier la personne pour obtenir ses informations. Pour ce faire, il est possible d'inscrire toutes les données du client dans un fichier texte. Il contiendra sur chaque ligne les données mobiles propres à un client. Le premier mot de la ligne sera le nom passé en clair dans les messages afin de faire l'authentification et ainsi retrouver les informations. Ce système est réutilisé des classes *PMIPReadConfig* et *DHCPReadConfig*. Après avoir récupéré toutes les informations de la ligne du fichier de configuration relative au client, il faut informer le *PMIP Client* et lui donner ces paramètres. Lorsque la classe *MN_Data_Mobile* est instanciée, elle permet de serrer un objet qui peut contenir les informations du client. Après avoir *sniffé* le message et trouvé le nom du client, le *NAS* fournit les données trouvées depuis un objet *MN_Data_Mobile* à la méthode du *PMIP Client*.

6.5. Problèmes rencontrés

6.5.1. Premiers tests Radius

Après l'activation de la sécurité par *Radius* au niveau de la borne *WireLess*, il est possible depuis un analyseur de trame du genre *WireShark* de percevoir une activité sur le réseau. La première trame *Radius* est de type *Request*. La *RFC* peu clair sur le *timeline* du protocole explique qu'il n'y a que quatre types de messages : le *Request*, le *Challenge*, le *Accept* et le *Reject*. Le premier est le seul qui provient du *NAS* en destination du serveur *AAA*. Il sert à transmettre les informations du client. Les trois autres messages sont uniquement envoyés par le serveur *AAA* en direction du *NAS*. Le challenge permet de demander d'échanger des informations suplémentaires avec le client. *L'Accept* donne au client l'autorisation de se connecter sur le réseau. Le *Reject* refuse cette derniere affirmation.

Le *RFC* de *Radius* [RFC2865] explique aussi que le protocole transporte les trames *EAP* sur un système sécurisé de différents types (TLS, PEAP, LDAP, etc…). Le *timeline* est quant à lui absent du *RFC*. Il n'est donc pas clairement expliqué s'il est possible de répondre au premier *request* du client par un message *Accept*. Après avoir réussi à capter un message *radius* depuis *Java* avec la classe *Jpcap*, il a été tenté de répondre par un message *Accept* préformaté. Cette méthode ne fonctionne pas, car ce n'est pas de cette manière que fonctionne le protocole. Après avoir reçu la première requête, Le serveur *AAA* doit vérifier dans sa base de données les informations de connexion et sécurité du client. Suivant le type de sécurité qu'utilise le client (*TLS*, *PEAP*. *LDAP*, etc…) un échange de paramètres de sécurité plus ou moins long s'effectue. Ce n'est qu'à la fin de cet échange que le message *Accept* ou *Reject* est transmis. Il n'y a donc pas d'alternative à l'utilisation d'un serveur Radius ou à la programmation entièrement manuelle.

6.5.2. Test avec serveur Radius

Plutôt que de réinstaller et paramétrer un serveur *Radius* complet, il a été possible d'utiliser le travail d'un autre diplômant devant développer ce système. Comme le *NAS* est activé sur la borne *Wireless* et que le serveur *Radius* est en place, il faut trouver un moyen pour récupérer les informations qui transitent entre les deux. La méthode utilisée est de faire un "*forward*" de paquet en *Java*. Pour cela il suffit d'indiquer au *NAS* que le serveur est l'ordinateur sur lequel se trouve le programme *Java*. Le programme va recevoir les trames et les renvoyer sans les transformer vers le vrai serveur *Radius*. Le protocole commence à échanger des informations entre le *NAS* et le serveur mais la séquence s'arrête au milieu puis recommence dès le début. Le problème est que les premieres trames de *challenge* servent à négocier un tunnel *IP* sécurisé entre le client et le serveur *AAA*. Comme le *NAS* croit que le programme *Java* est le serveur il essaie d'établir le tunnel avec lui. Pour que cette méthode fonctionne il faut faire non pas un "*forward*" mais un *proxy Radius*, ce qui revient au final à refaire un serveur complet.

6.5.3. Sniffer de trames Radius

Avec le *package jPCAP* trouvé sur Internet, il est possible de récupérer des trames réseau qui ne nous sont pas destinées. Ainsi, la séquence et l'établissement du tunnel sécurisé entre le client et le serveur peut se passer dans les meilleures conditions. Le *sniffer* va analyser les trames et capturer seulement la trame *Accept*. Cette trame est cryptée, à l'exception du champ du nom du client. Il n'est donc pas possible de récupérer des données transmises par le serveur *Radius* autrement qu'en les décryptant. La solution est de mettre les informations mobiles du client dans un fichier texte et de les indexer par le nom passé en clair dans les messages *Accept*.

CHAPITRE 7

FICHIER DE CONFIGURATION

Le *PMIP Client* et le *DHCP* doivent pouvoir être configurables comme l'est le *FA* et le *HA d'Open Dynamics*. Pour cela, il existe plusieurs solutions dont deux qui ont été retenues. La première est un fichier *XML* contenant toutes les informations dans une arborescence. Ce type est très apprécié actuellement car très facilement modulable et reste relativement simple à modifier de par sa hiérarchisation des données. *Java* a par contre besoin d'un parseur *XML* qui est relativement difficile à utiliser. Il existe plusieurs classes téléchargeables directement sur Internet qui facilite l'implémentation de ce type de fichier de configuration. Il existe aussi la même et ancienne méthode qu'utilise *Open Dynamics* qui est de passer par un fichier texte. Ce dernier est beaucoup plus lisible que le *XML* et permet de mettre plusieurs commentaires qui peuvent être mis en rapport avec la variable. Par contre, la structure est nettement moins pratique à utiliser que le *XML*. Pour rester en relation avec la configuration de *Dynamics*, cette solution va être gardée.

Le fichier de configuration ressemble à ceux utilisés par *Dynamics*. Les lignes de commentaires commencent par un # et sont ignorées lorsque le programme les lit. Les variables sont notées par leur nom et séparées par un espace de leur valeur. Si elles sont précédées d'un #, les variables ne sont pas lues par le programme et ce dernier applique la valeur par défaut contenue dans sa table. Dès son instanciation, la classe de configuration, doit ouvrir un fichier texte qui se situe à la racine de l'installation du programme *PMIP Client* dans un dossier intitulé *config*. Ce fichier doit être parcouru ligne après ligne. Un test est fait sur le premier caractère rencontré sur la ligne. Si c'est un #, il doit l'ignorer et passer à la ligne suivante, sinon il lit le premier mot. Celui-ci est le nom de la variable qui doit être comparée avec une entrée dans une table de hachage. Cette dernière contient les valeurs par défaut indexées par leur nom de variable. Il est ainsi possible de ne pas respecter d'ordre précis dans la disposition des variables dans le fichier texte. Si un nom trouve sa correspondance dans la table de hachage, la valeur sera modifiée en conséquence. Dans le cas contraire, l'utilisateur a fait une erreur et a modifié le nom de la variable, ce qui aura comme effet de ne rien changer et de laisser la valeur par défaut. Cette classe possède différentes méthodes pour retourner les valeurs demandées par les programmes. Plusieurs exemples de ces fichiers de configuration utilisés, sont disponibles en annexe de ce document.

CHAPITRE 8

ENVIRONNEMENT DE DEVELOPPEMENT

8.1. Architecture du réseau de test

8.1.1. Architecture de base MIP

Le premier réseau installé pendant les premières semaines du travail de diplôme est adapté pour une architecture *MIP*, voir Figure 20. Il se situe dans les locaux de l'HEIG-VD d'Yverdon sur le site de Chesaux à l'étage B. Ce réseau possède un *HA* (Réseau Rouge), deux *FA* équipés de deux routeurs *Wireless* (Réseau Vert et Bleu) paramétrés en *AP* et un *Mobile Node*.

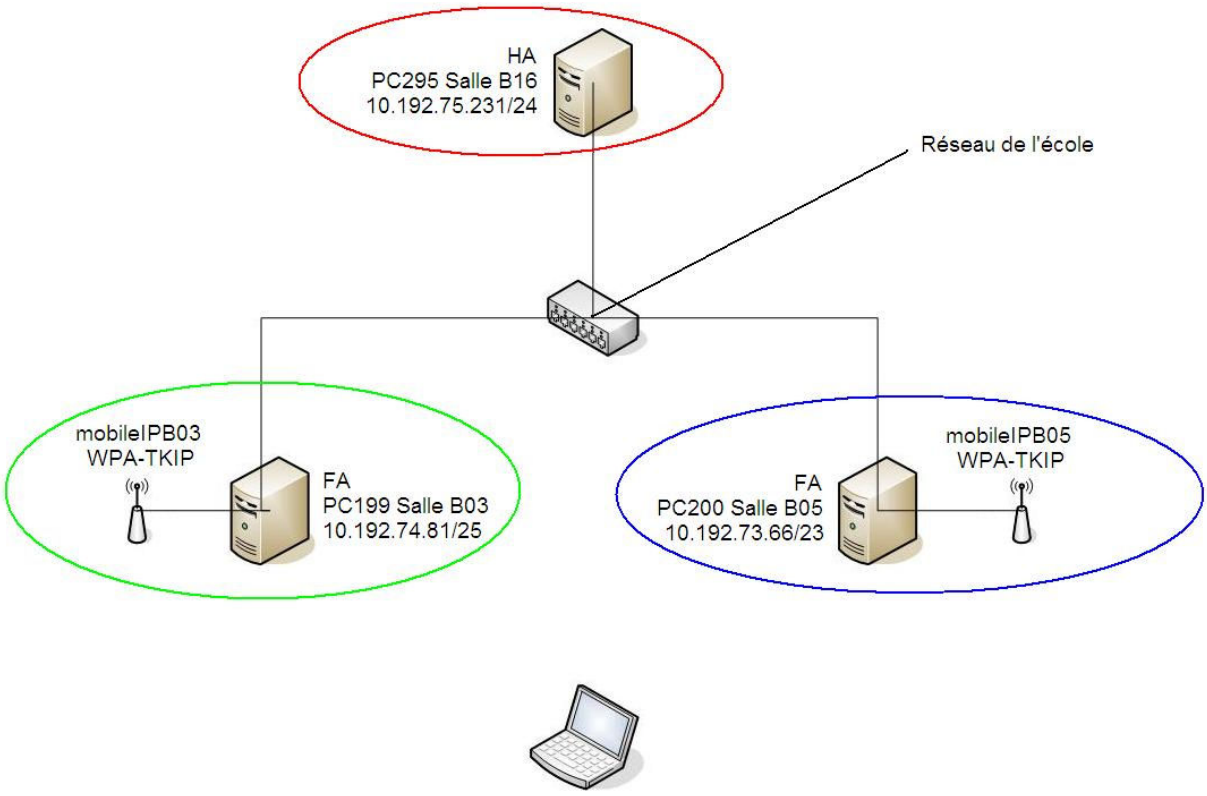


Figure 20 : Structure du réseau de test.

Ce premier réseau est installé et testé avec la configuration (décrite en annexe) de la solution *mobile IP* (*MIP*) *Open Source*, *OpenDynamics*. Il permet de se familiariser avec ce protocole et de préparer le *HA* et le *FA* qui seront réutilisés plus tard dans le réseau *PMIP*. Il dispose actuellement d'un ordinateur par réseau et de deux bornes *Wireless* dans les réseaux éloignés (vert et bleu). L'ordinateur possède deux cartes réseau, dont la première est connectée au réseau local et la deuxième possède une adresse fixe privée sur laquelle est branchée la borne *Wireless*

8.1.2. Spécification des ordinateurs de test

- Le *MN* est installé sur l'ordinateur portable (*Asus F3JP*) du diplômant. *L'OS* est une version *Linux Ubuntu 8.04 – Hardy Heron* (sortie avril 2008).
  IP mobile : 10.192.75.232/24

- Les *FA* sont installés sur des *PC* (*Dell Optiplex gx270*) avec un *Ubuntu 8.04*. Ces deux machines, *PC199* (Réseau Vert) et *PC200* (Réseau Bleu) sont respectivement en salle B03 et B05. Elles sont toutes les deux reliées à un *router Wireless* configuré en *Bridge* pour assurer le rôle d'un *AP*. Ces *routers* sont des *Linksys* (WRT54GS v5.1).

	PC199 / B03	PC200 / B05
IP réseau école	10.192.74.81/25	10.192.73.66/23
IP réseau interne	192.168.0.1/24	192.168.1.201/24

- Le *HA* est installé sur le PC295, avec un *Ubuntu 7.10*.
  IP : 10.192.75.231/24

8.2. Installation du réseau de test

8.2.1. Installation de Dynamics sur Ubuntu

En premier lieu, il faut installer deux librairies, *libgmpX* et *libreadlineX* (X est le numéro de version). Les versions utilisées sont : *libgmp3-dev* et *libreadline5-dev*.

```
- Sudo apt-get install libgmp3-dev
- Sudo apt-get install libreadline5-dev
```

Ensuite il faut éditer le fichier *Wireless.h* et lui ajouter la commande : *#define INFAMSIZ 16* juste en dessous de la ligne *#define _LINUX_WIRELESS_H*.

```
- Sudo gedit /usr/include/linux/wireless.h
```

Pour terminer, il faut extraire le répertoire *dynamics-0.8.1.zip* puis exécuter la commande suivante depuis le répertoire *dynamics-0.8.1* :

```
- Sudo make install
```

8.2.2. Configuration de Dynamics

Pour configurer chaque élément de *Dynamics*, il existe des fichiers texte de configuration (*.conf) qui se trouvent dans le répertoire /usr/local/etc/. Les trois fichiers concernés sont dynmnd.conf (pour le *MN*), dynfnd.conf (*FA*) et le dynhad.conf (*HA*). Pour accéder en local à ces fichiers, il suffit de taper la commande :

```
- Sudo gedit /usr/local/etc/*.conf.
```

Pour l'accès à distance il faut passer via le terminal *SSH* avec la commande :

```
- Sudo ssh <UserName>@<IP_Machine_Distant>.
```

Puis ouvrir le fichier via *NANO* :

```
- Sudo nano /usr/local/etc/*.conf
```

Le protocole est configuré de manière minimale sans inclure toutes les options de sécurité. Les fichiers de configuration se trouvent en annexe de ce document.

8.2.3. Modification du client Linux

Le client *Ubuntu* sur lequel est installé le *MN* doit être légèrement modifié car le programme de gestion du réseau de l'interface graphique est limité. En effet, avec le nombre de réseaux *Wireless* de l'école, il ne faut pas que le *MN* essaie de se connecter sur différentes bornes. Pour limiter les connexions, il faut aller modifier deux fichiers, *Interfaces* et *WPA_Supplicants* via le terminal. Le premier possède la configuration *IP* de l'interface selon un label qui représente la borne à laquel il faut se connecter. Ce label se retrouve dans le *WPA_Supplicants* et possède les paramètres pour se connecter sur une borne *Wireless* précise. Ces deux fichiers se trouvent en annexe de ce document sous deux configurations différentes selon le *MIP* ou le *PMIP*. En effet, lors des tests avec le *MN* sous une architecture *MIP*, le client possède une adresse *IP* fixe (dans le fichier *Interface*) et une connection via *WPA* sur la borne (*WPA_Supplicant*). Mais pour le *PMIP* le client doit posséder une adresse dynamique par *DHCP* (*Interface*) et une connection par *Radius* (*WPA_Supplicant*).

8.3. Test du réseau *MIP* avec *MN*

8.3.1. Introduction

Ce premier test avec un réseau *MIP*, s'est effectué avec un *MN* configuré le plus simplement possible pour pouvoir paramétrer le *HA*. La désencapsulation se fait au niveau du *MN* pour l'instant afin d'avoir une configuration simple sur le *FA*. Les tests se sont bien passés bien que la déconnexion et reconnexion lors du passage entre deux réseaux soit un peu lente. En effet, le temps d'attente est d'environ 3 à 5 secondes ce qui ne dérange pas pour un téléchargement mais devient tout de suite plus ennuyeux avec de la téléphonie sur *IP*. Une fois le fichier *WPA_Supplicant* ajouté, le client tente de se connecter uniquement aux deux bornes *Wireless* de test. Le fichier *Interface* est configuré de manière à recevoir une adresse *IP* via *DHCP*.

8.3.2. Modification requise pour adaption en PMIP

Le *Draft* du *PMIP* explique que la désencapsulation se fait soit au niveau du *PMA*, soit du *FA* dans le cas d'un *PMA* séparé. C'est cette dernière solution qui est utilisée puisque le réseau dispose déjà d'un *FA* fonctionnel. La prochaine étape du réseau de test est de laisser le *FA* désencapsuler le tunnel. Il faut modifier la configuration au niveau du *MN* et du *FA* pour leur signaler à chacun que c'est le *FA* qui porte la *Care of Address* (*CoA*). A cette étape il est possible de remarquer que le programme *Dynamics* ne respecte pas tout à fait la *RFC 3344* du *MIP*. Ce dernier note clairement qu'un *FA* doit savoir utiliser les deux méthodes en même temps. C'est-à-dire que le *FA* doit être à même de gérer des clients qui demandent la désencapsulation et d'autre qui le le font pas. Dans le cadre du projet ce n'est pas un probleme mais il est important de le spécifier.

Le test de cette nouvelle configuration a posé quelques problèmes. En tout premier la négociation du tunnel se passe sans problème. Le client obtient une adresse *DHCP* du réseau éloigné dans lequel se trouve le *FA*. C'est là où le problème se pose car le *FA* demande à ce que le client possède une adresse du même réseau que le *HA*. Pour contourner le problème le client hérite d'une adresse fixe de la plage d'adresse du *HN* dans lequel se trouve le *HA*. Pour cela il faut modifier le fichier *Interface* et paramétrer pour chaque borne *Wireless*, l'adresse fixe à utiliser ainsi que le masque de sous-réseau et la passerelle par défaut. C'est cette dernière configuration du *MIP* qui se trouve en annexe.

Le second test avec cette adresse fixe modifiée n'est toujours pas concluant. Lorsque le *MN* tente de discuter avec le *HA*, les trames passent bien par le *FA* et sont transmises au *HA*. Mais lors du retour, les trames ne sont pas remises au client et le *FA* commence à faire des requêtes *ARP* via la route par défaut. Le problème survient parce

que le *FA* possède deux interfaces réseau. La route par défaut de *Linux* est l'interface réseau où se trouve la *Care of Adresse* (*eth0*). Cette interface est celle qui discute avec le *HA* et dont l'adresse *IP* est dans la plage *IP* du réseau distant. Le *FA* reçoit donc bien les trames que le *HA* envoie à l'adresse du client (dans notre cas 10.192.75.232). Mais il ne sait pas où la renvoyer. C'est pour cela qu'il s'adresse à sa passerelle par défaut. Le *FA* va commencer à émettre des requêtes *ARP* pour trouver à qui appartient l'adresse mais sans avoir de réponse. C'est clairement un problème de routage au niveau de *Linux*. Pour résoudre temporairement le problème il suffit d'ajouter une route spécifiant que tout ce qui est pour l'adresse du client (10.192.75.232) doit être dirigé vers l'interface sur laquelle est branchée la borne *Wireless*. Depuis le terminal il suffit d'entrer la commande :

- `IP route add 10.192.75.232 dev eth1.`

Ce problème de routage provient du *FA* qui n'ajoute pas la route lorsque deux interfaces sont connectées sur l'ordinateur. Après quelques recherches *Open Dynamics* précise dans sa documentation qu'il subsiste des bugs de routage sans préciser clairement de quel type. Il est à supposer que celui-ci en est un ! Pour corriger ce problème, il faut reprendre les sources du programme et corriger cet oubli de routage lorsque le *FA* dispose de plus d'une interface. Après la résolution de ce premier gros problème qui a fait perdre beaucoup de temps par rapport au planning Gantt, la mobilité fonctionne parfaitement. Par cette méthode le temps de déconnexion puis de reconnexion pendant le passage entre deux réseaux n'est pas moins long que précédemment. Après renseignement, il existe des *Drafts* en cours de développement qui permettent d'accélérer le processus. Ce réseau de test permet maintenant d'ajouter un *PMIP Client* sur le même ordinateur que le *FA* pour pouvoir utiliser le protocole *Proxy Mobile IP* (*PMIP*).

8.4. Problèmes rencontrés

Les petits problèmes de programmation ne trouvent pas leur place dans cette section. Ils ont d'ailleurs été décrits, pour les plus importants, plus haut dans les chapitres relatant du développement. Seuls les problèmes liés à l'architecture, à l'implémentation et au protocole sont présents dans cette partie. Ils seront décrits dans l'ordre dans lesquels ils ont été rencontrés.

8.4.1. Fixe du TTL des paquets IP

Le problème du *TTL* doit absolument être résolu pour continuer le projet. La classe *MulticastSocket* propose l'option de modification du *TTL*. L'objet instancié de cette classe possède une méthode *SetTimeToLive*(255). Le problème est que *Java* ignore ce changement. Certaines sources sur Internet préconisent l'installation de la version 7 de *Java* pour corriger le problème mais sans succès. Sur un autre forum ils précisent que c'est un bug de *Java* sur *Linux* et qu'il faut intégrer trois lignes de code en début du projet qui vont modifier les paramètres de la machine virtuelle lors du lancement :

```
Properties props = System.getProperties();
props.setProperty("java.net.preferIPv4Stack","true");
System.setProperties(props);
```

Mais une fois de plus sans résultat. Finalement, après avoir perdu trop de temps à essayer de faire fonctionner cette classe en se renseignant sur des forums, il s'est avéré qu'elle n'est pas utilisable pour résoudre ce problème.

Il existe une solution qui sera plutôt un fixe temporaire du problème, qui consiste à changer le *TTL* par défaut de *l'OS* juste avant l'envoi du message et de le rechanger après. La commande *Linux* pour modifier le *TTL* par défaut est :

```
echo '255' > /proc/sys/net/ipv4/ip_default_ttl
```

Java n'arrive pas à exécuter directement cette ligne de commande mais peut par contre lancer un *bash* (*NewTTL.sh*) contenant cette ligne de code. Voici le code :

```
Runtime r = Runtime.getRuntime();
Process p = r.exec(«"sh ./NewTTL.sh");
p.waitFor();
```

Le problème de cette méthode est qu'elle est dépendante de *l'OS* utilisé et donc qu'elle devra s'adapter suivant les *OS* rencontrés. Pour cela *Java* propose de retourner le nom de *l'OS* sur lequel il tourne :

```
System.getProperties().getProperty("os.name");
```

Il suffit donc de récupérer cette valeur pour savoir quel fichier *bash* exécuter. Un problème de cette méthode est qu'il faut revenir au *TTL* par défaut après avoir envoyé le paquet. Un deuxième fichier *bash* peut contenir la vraie valeur par défaut de *l'OS* mais *Java* ne dispose pas de moyen pour récupérer le *TTL* existant, il faut donc le paramétrer manuellement dans le fichier. Cette solution fonctionne bien pour *Linux* mais pour *Windows* l'accès à la base de registre est compliquée depuis *Java*. Le *TTL* par défaut a été modifié manuellement dans la base de registre. Ceci est un fixe provisoire mais il devra être changé pour pouvoir proposer ce protocole de manière *open source*.

8.4.2. Installation d'OpenDiameter

Sur *Linux*, *OpenDiameter* nécessite l'installation de plusieurs libraires comme par exemple : *ACE*, *OpenSSL* et *BOOST*. Le premier site Internet trouvé [OpenDiameter] proposant *OpenDiameter* met à disposition une marche à suivre pour l'installation du programme. Celle-ci n'est que peu explicite et ne décrit pas l'installation de toutes les libraires. En effet, lors de l'installation plusieurs problèmes surviennent. Pour *ACE* l'installation ne fonctionne pas comme le décrit la marche à suivre. La dernière version en date de *Linux* (*8.10*) possède la plupart de ces libraires par défaut lors de l'installation. Après un *UpGrade* du système, l'installation ne fonctionne toujours pas. Des recherches sur Internet pour résoudre le problème présentent un site suisse [OpenDiameter2] proposant un *package* contenant *OpenDiameter* et toutes ses libraires mais pour une version antérieure *d'Ubuntu* (*Gutsy 7.10*). Ce dernier s'installe correctement et dispose d'une bonne documentation. Mais le temps vient à manquer et cette partie est remplacée par l'utilisation d'un *sniffer* de paquet et l'utilisation d'un serveur *Radius* d'un autre diplômant.

8.4.3. Installation du PMIP Client avec le FA

Le *Draft PMIP* spécifie que lorsque le *PMA* est partagé (*split*), les deux entités PMIP *Client* et *FA* peuvent communiquer depuis le même ordinateur ou séparément. Le premier test du programme se fait en local sur la même machine que le *FA*. Pour discuter entre eux, les deux éléments disposent de deux interfaces à choix, le *loopback* et l'interface sur lequel se trouvent les clients *MIP*. Lors des tests effectués, la connection ne s'établit pas correctement. Dans le premier cas, quand l'interface () vers lequel sont dirigés les clients *MIP* est utilisé pour discuter avec le *FA*, le programme *Java* n'arrive pas à envoyer d'informations. Il est impossible d'apercevoir les paquets transmis par un programme de capture puisque les paquets n'atteignent pas la couche la plus basse pour être envoyés sur le réseau. Le *FA* ne reçoit pas non plus ces paquets, car il ne tente pas de communiquer avec le *HA*. Il reste encore l'option de transmission par l'interface de *loopback*. Le *PMIP Client* envoi les données vers le *FA* qui les reçoit et lève le tunnel avec le *HA*. Le problème est lors du retour de l'information, lorsque le *FA* essaie d'informer le *PMIP Client*, ce dernier n'arrive pas à récupérer les trames depuis le *loopback*. Cette situation étrange peut laisser penser que *Java* n'est pas capable d'écouter sur cette interface. Un petit programme de test

Java, mettant en scène deux *threads* dont le premier écoute et l'autre envoie sur l'adresse de *loopback*, prouve que *Java* en est capable. Un deuxième test est fait et met en collaboration le petit programme et le *PMIP Client*. Après avoir négocié le tunnel, le *FA* renvoie l'information au *PMIP Client* qui ne la reçoit pas mais par contre il reçoit l'information du petit programme test.

Après ces problèmes rencontrés, il est décidé de tester le programme depuis deux ordinateurs séparés mais sur le même réseau interne. Un deuxième ordinateur est installé avec un *Linux 8.04* sur lequel tourne le *PMIP Client*. Avec cette solution, tout se passe comme prévu et le protocole est négocié correctement. Voici ci-dessous le schéma d'architecture matériel utilisé actuellement avec les deux machines séparées (voir Figure 21) ainsi que le *timeline* du protocole dans son état actuel (voir Figure 22).

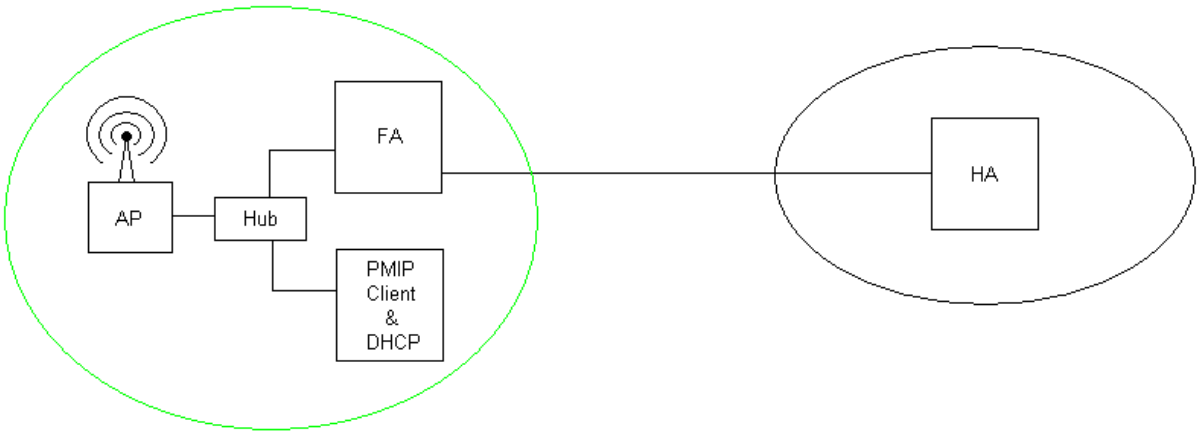


Figure 21 : Modification de la structure physique du réseau

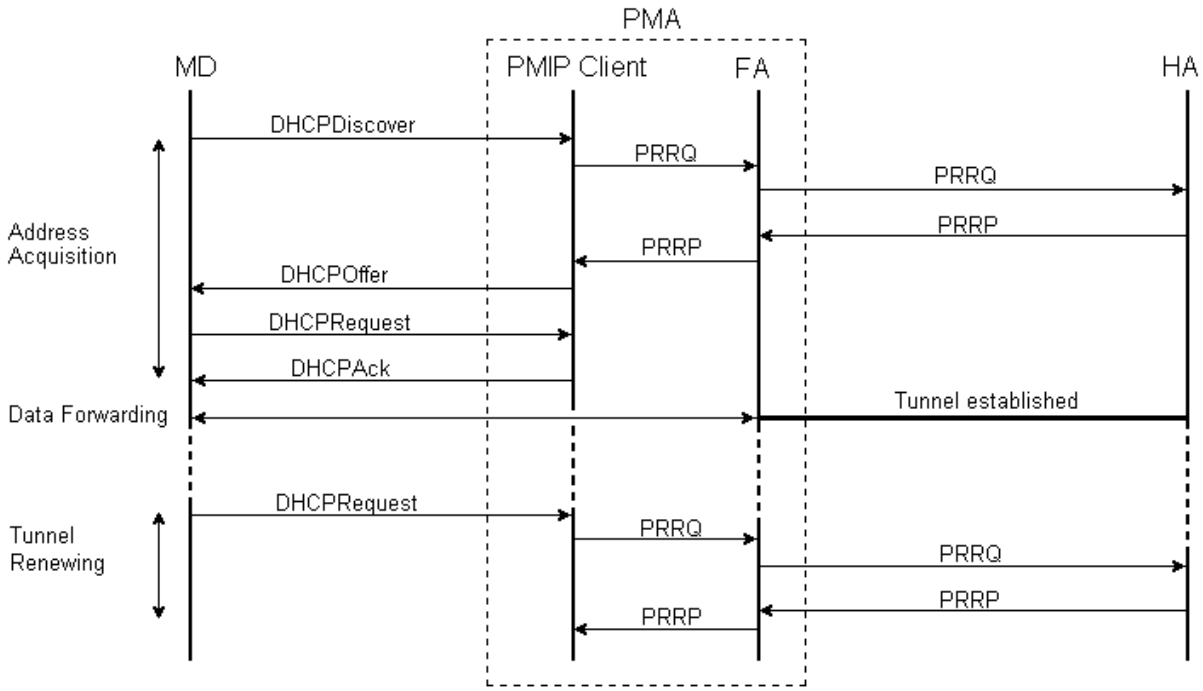


Figure 22 : *TimeLine* avec *PMIP Client* et *FA* séparé sur deux *PC*

Le *package jPCAP* permet de contrôler plus bas dans les couches les trames des paquets envoyés. Deux tests ont été effectués vers la fin du projet, dont celui du *sniffer* de messages *Radius*. L'autre est un test d'envoi de paquet *UDP* avec un *TTL* de 255 et une interface réseau bien définie. Ce *package* pourrait résoudre la plupart des problèmes rencontrés jusqu'à présent. Le seul ennui est l'écoute des paquets qui se fait avec un rafraîchissement régulier. Il existe apparamment un mode bloquant mais il n'a pas été essayé.

8.4.4. Service DHCP sur Linux

Le développement du service *DHCP* s'est fait en début de projet et n'a pas été retouché depuis. Après son insertion dans la structure *PMIP Client*, il est testé sur une machine *Linux* fraîchement installée. Lors du lancement du programme, le programme plante et lève une exception lorsque le *DHCP* tente de répondre au client. L'exception indique que réseau est inatteignable (*Network is Unreachable*). Lorsque le programme a été développé sur une machine de test, l'interface de cette dernière avait déjà obtenu une adresse *IP* dynamique depuis le réseau de l'école. Cela a eu pour effet de mettre une route par défaut sur l'interface courant. Lorsque le *DHCP* répond au client il envoie ces paquet au 255.255.255.255 et s'en remet donc à sa route par défaut pour connaître l'interface sur lequel il est envoyé. Le problème vient de là, car le nouveau *Linux* sur lequel est installé le *DHCP* n'a jamais possédé d'adresse *DHCP*, mais seulement une adresse fixe. Il ne possède pas de route par défaut et lorsque le service *DHCP* tente d'envoyer ce message, il dit qu'il ne connaît pas le réseau à atteindre. Pour résoudre le problème temporairement il suffit d'ajouter la route depuis la console :

- `sudo route add default dev eth0`

La solution est d'ajouter l'exécution de cette commande dans le service *DHCP* lors de son démarrage. Il doit pour cela récupérer son adresse *IP* et ensuite ajouter cette route.

8.4.5. Installation du PMIP Client contenant jPCAP sur un Linux

Sous *Windows* le programme ne pose aucun problème et s'exécute correctement. Sur *Linux* la compilation du programme s'effectue sans problème mais lors du lancement le programme plante en levant une exception système. Le programme à toujours été testé sur *Linux* durant tout son développement sauf avec l'insertion de la libraire *jPCAP*. Cette dernière nécessite l'installation de la librairie *Linux Libpcap*. Le programme à été testé avec toutes les versions de cette libraire jusqu'à sa dernière version en date (*libpcap0.8-dev*) mais le *PMIP Client* plante toujours de la même manière. C'est un problème à régler obligatoirement si l'on veut le faire tourner sur *Linux* avec cette libraire qui remplacera peut être même les *sockets* dans une version future. Essayer de changer de version de *Linux* (*8.04*) n'a pas été tenté mais de toute manière ce n'est pas une solution. Il n'y a actuellement aucune proposition pour résoudre ce problème à part passer du temps sur des forums et chercher plus d'informations sur la communauté *Linux*.

8.4.6. Test de mobilité avec client Windows

Lorsque le client se présente sur la borne *Wireless* tout le protocole se déroule correctement jusqu'à l'obtention de l'adresse *IP*. Le client *Windows* dispose de la mobilité jusqu'au renouvellement du tunnel. A ce moment le tunnel tombe car il n'a aucune indication provenant du *DHCP* que le client est toujours là. Après analyse avec *Wireshark*, les clients envoient au minimum deux *renewal* avant que le tunnel tente de se rafraîchir. Ces messages de *renewal* ne sont pas captés par le *Socket Java*. L'adresse *IP* est correct le *MAC* de destination du message est celui de la passerelle par défaut du

réseau, c'est-à-dire du deuxième *PC* ou se trouve le *FA*. Dans un même réseau physique, c'est l'adresse *MAC* qui prime. Maintenant, pourquoi *Windows* envoie-t-il ce paquet à destination d'une autre adresse *MAC* que celle des paquets dont ils proviennent ? Cela reste une énigme. Il est à supposer que le Système d'exploitation au niveau de la couche *IP* répond au message qui lui a été envoyé. Mais pour la couche ethernet là où se situe l'adresse *MAC*, il se rend compte que l'adresse *IP* où il faut envoyer le *renewal* (soit 192.168.0.2) diffère de son adresse *IP* (10.192.75.232). Ce doit être pour cela qu'il indique l'adresse *MAC* du *default Gateway*.

Pour fixer ce problème avant de savoir exactement pourquoi, il suffit que le service *DHCP* donne une adresse au client avec un *rebound time* egal au *renewal time*. Ainsi par cette méthode le client renvoie deux messages de *renewal*. Le premier avec une adresse de destination de 192.168.0.2 et l'adresse *MAC* de la passerelle par défaut. Le deuxième qui est celui de *rebound* envoyé en broadcast à 255.255.255.255 et avec une adresse *MAC* de 00:00:00:00:00:00. Ce dernier est capté par le *socket*, le tunnel peut être rafraîchi.

8.4.7. Test de mobilité avec un client iPhone (Mac OS)

Dès le premier test, l'iPhone n'accepte pas l'adresse *IP* fournie par le service *DHCP*. Après la réponse *Ack* du *DHCP* le téléphone renvoie imédiatement un *DHCP Discover*. Ce problème est déjà apparu lors des premiers tests avec le client *Linux* quand les offres *DHCP* ne possèdent pas de temps de bail (*Release Time*). Or ce n'est pas le cas ici car ce temps envoyé sur quatre octets à 255.255.255.255, donc au maximum. Après réduction de se temps par la moitié, l'iPhone accepte l'adresse *IP*. En fait, le téléphone est bugé car les programmeurs ont oublié de non-signer la variable. Le test est très simple à vérifier car si le temps donné est de 2147483647 c'est-à-dire tous les *bits* sur quatre octets à 1 sauf celui de poids le plus fort, l'iPhone accepte l'adressage. Avec 2147483648 c'est-à-dire tous les bits à 0 sauf celui de poids le plus fort l'iPhone redemande une adresse après chaque message *Ack*. Pour l'iPhone ce numéro est négatif et représente réellement une valeur négative. Ce phénomène est appelé le complément à deux pour permettre à des nombres binaires d'être signés. Lorsque le bit de poids le plus fort est à 1 le chiffre est négatif. Pour ce problème il n'existe pas de solution à part fournir un temps de bail plus petit ou attendre une mise à jour de l'iPhone.

8.4.8. Test de mobilité avec un client Linux

Ce cas est le pire de tous, car le client *Linux* reçoit une adresse *IP* mobile mais n'arrive pas à communiquer avec le *HA*. La raison est simple car *Linux* ne prend pas en compte la passerelle par défaut donnée via *DHCP*. En fait *Linux* ne paramètre volontairement pas cette adresse (192.168.0.1) car elle ne correspond pas à la plage de l'adresse *IP* fournie (10.192.75.232). Après l'obtention de son adresse *IP*, *Linux* se met à émettre des requêtes *ARP* pour découvrir des gens de sa plage de réseau (10.192.75.0). Pour corriger ce problème qui est entièrement normal, l'ordinateur qui possède le *FA* doit être en mesure de faire du *Proxy ARP*. Ceci permet de faire croire au client *Linux* en lui donnant son adresse *MAC* comme destination des paquets, qu'il fait partie du même réseau et que le client peut lui envoyer ces messages. Depuis *Linux* il est très facile de faire du *proxy ARP* via la console. Le *FA* devrait donc en théorie analyser l'adresse *IP* des messages lors de l'établissement du tunnel et faire du *proxy ARP*. Rappelons qu'*OpenDynamics* affirme avoir des bugs de routage. Ils n'ont peut être jamais pensé à ces problèmes surtout qu'il n'est plus mis-à-jour depuis plusieurs années. Ce problème rejoint celui du *routing* de ce chapitre, section 8.3.2, paragraphe 4. Pour corriger ce problème il faut modifier au même endroit le fichier source *d'OpenDynamics* que le problème de routage.

CHAPITRE 9

TRAVAIL A POURSUIVRE & AMELIORATIONS

Le protocole fonctionne mais de manière très basique et dans des conditions bien définies. Les améliorations pour rendre ce programme totalement operationnel sont décrites dans différentes sections ci-dessous.

9.1.1. Changer les sockets

L'utilisation des *sockets* pose d'énormes problèmes à cause de leur trop grande simplicité. En effet il suffit juste de leur donner un tableau d'octet et d'appeler la méthode `Socket.send(trame)` pour envoyer le message. Le *package jPCAP* [JPCAP] pourrait quand à lui résoudre ces problèmes. Il est possible effectivement de construire ses propres trames couche après couche. Le seul problème à résoudre est de réussir à écouter les paquets réseau de manière bloquante. C'est-à-dire que l'objet d'écoute ne doit pas se trouver dans une boucle *While* infinie pour prendre toutes les informations sinon il va consommer de la ressource processeur inutilement.

9.1.2. Librairie jPCAP et Linux

Ce problème nécessite juste du temps et de la patience pour aller chercher l'information sur des forums. Il est possible de faire des tests avec d'autres versions d'*Ubuntu* voir même une autre distribution de *Linux*. Actuellement il n'y a pas de solution proposée pour résoudre ce problème.

9.1.3. Proxy ARP & routage

Ces problèmes proviennent clairement *d'OpenDynamics*. Pour les résoudre le plus compliqué est de trouver où il faut placer la modification car ce code est extrêmement mal documenté et commenté. Depuis *Linux* il est très simple en quelques commandes via le terminal de résoudre ce problème. Le language *C* avec lequel *OpenDynamics* est fait permet lui aussi de résoudre le problème en quelques lignes.

9.1.4. Complèter les Must manquants

Lorsque tous les points précédents seront résolus, il sera possible de continuer à résoudre les bugs et conventions du protocole. En effet, pour tester toutes les facettes et tous les cas il faut que l'environnement fonctionne parfaitement avant de continuer. Les *Must* manquants sont indiqués dans la section MUST 5.6

9.1.5. Relay DHCP

Lorsque le *PMIP Client* fonctionnera totalement et respectera en un mot la *RFC* et ce dans tout les cas, il sera possible de se pencher sur le sercive *DHCP*. Actuellement ce service est très basique et ne respecte de loin pas toute les conventions de sa *RFC*. Cela reste tout de même une partie importante du développement et qui doit être complétée. Le *relay DHCP* permet dans le cas où le client n'obtient pas la mobilité d'obtenir une adresse *IP* directement du serveur *DHCP* du réseau dans lequel il se trouve. Le service doit donc relayer l'information en modifiant les trames pour y ajouter des informations. Ces dernières seront retirées lors de la retransmission des trames au client. Pour le développement il faut se référrer à la *RFC* [RFC3046]

9.1.6. Test de mobilité avec retour du client dans le réseau home

Après que le protocole fonctionne entre les deux réseaux éloignés, il faut aussi tester le cas du client qui revient dans le réseau *home* et celui du client qui le quitte pour un réseau éloigné. En théorie il n'y a rien à ajouter dans ce cas puisque le client obtiendra une adresse directement du serveur *DHCP* du réseau *Home*.

9.1.7. Amélioration de la JavaDoc

Dès le moment où le programme *PMIP Client* développé est mis sur un site de téléchargement, il faut que *Java* doc qui l'accompagne soit fait le plus clairement possible. Pour cela il faut qu'une personne externe essaie de reprendre le code et s'appuie sur la documentation *Java* fournie pour expliquer ce qu'il ne comprend pas afin de l'améliorer. En dernier lieu il peut être possible de le traduire dans plusieurs langues

9.1.8. Intégrer un système AAA Diameter

Le *Draft* préconise l'utilisation d'un service *AAA* se reposant sur le protocole *Diameter*. Ce dernier est relativement récent et l'on ne trouve que peu d'information à son sujet. Il existe néanmoins un site en Suisse qui fourni une version *Open Source* [OpenDiameter2] fonctionnant sur *Ubuntu*. Un autre site propose lui aussi un *package Java* [Java Diameter] relativement bien conçu permettant d'utiliser ce protocole. Il est néanmoins soumis à certaines règles d'utilisation qui sont affichées sur le site. C'est ce *package* qui doit être utilisé pour remplacer la classe temporaire *NAS*. Ce point est l'aboutissement final du protocole *PMIP* et permet une utilisation complète et viable. En contrepartie l'installation, le paramétrage et le développement du *NAS* pour communiquer avec le *PMIP Client*, représente à lui seul un travail de diplôme à temps complet.

CHAPITRE 10

CONCLUSION

Ce projet proposé par l'institut IICT de l'HEIG-VD paraît à première vue relativement simple. Effectivement, le *Draft* ne contient que peu de pages et le protocole en lui-même ne paraît pas difficile. Mais en analysant le *Draft* plus en détails, il est possible de se rendre compte que ce projet touche à pratiquement toute l'arborescence des technologies informatiques. Il demande une bonne notion d'une dizaine de protocoles différents. Toute une partie est liée à la sécurité et à l'authentificaion *Wireless* par une technologie *AAA*. Les messages du protocole *PMIP* sont cryptés avec des algorythmes de chiffrage. Une bonne maîtrise des systèmes d'exploitation courants (*Windows* et *Linux*) est obligatoire. Il faut aussi posséder un bon sens d'analyse pour éviter des problèmes de programmation lors du développement.

Durant ces trois mois, le projet a pris des tournures très différentes. Au tout début, le protocole est découvert en faisant des tests de mobilité avec le réseau *MIP*. Puis est arrivée la partie de réflexion concernant le développement du protocole. Cette partie est importante pour éviter de partir dans une mauvaise direction et de se retrouver coincé avec du code inutilisable. Ensuite est venue la partie contenant les problèmes que nous éspérons éviter. Il y a ceux relatifs au code qu'il est possible de résoudre relativement facilement. D'autres sont des problèmes d'implémentation ou d'incompatiblité qui nécessitent plus de temps pour leur résolution. Dans son état actuel le code est fonctionnel mais dans un cadre bien précis. En effet, à cause du retard pris sur le planning Gantt, le développement en est à l'itération de fonctionnement basique de la sixième semaine. Ce retard ne provient pas des problèmes liés à la programmation, car ils ont été comptés dans l'estimation Gantt. Il provient notamment de *Java* qui est un language de haut niveau et qui est relativement limité par son *package* de base. Il est possible de trouver d'autres *packages*, voire même de les créer nous même, pour permettre de pallier à ces problèmes. Mais dans les deux cas c'est une charge de travail en plus et donc du temps en plus.

J'ai trouvé le développement de ce *Draft* très enrichissant, car il m'a permis d'utiliser une grande partie des connaissances acquises lors de ma formation dans cet établissement. Une implémentation totale du protocole n'est pas possible à une personne dans le temps imparti. Toute la partie *AAA* par *Diameter* représente à elle-seule un travail de diplôme. Je regrette toutefois que le protocole n'ait pas pu être plus abouti. Il m'aurait permis de réaliser jusqu'au bout le travail d'un ingénieur en finalisant par des tests, des évaluations et des propositions d'améliorations. Pour terminer, *Java* qui est le language de programmation utilisé pour ce projet, ne me paraît pas être un bon choix. En effet, les *packages* fournis par *Java* possèdent un vaste choix d'utilisation pour des applications de haut niveau. Mais lorsqu'il est décidé de travailler dans des couches basses, il faut trouver d'autres *packages*. Une autre solution est de les développer soi-même ce qui augmente relativement le temps et la charge de travail. Je pense que le language *C* ou *C++* serait un meilleur choix de par son affinité avec *Linux* et ses possibilités de gestion de bas niveau.

REMERCIEMENTS

- Le travail accompli jusqu'à présent, ne se serait pas aussi bien déroulé sans l'aide du collaborateur scientifique de l'IICT, Monsieur Doswald Alistair.

- Je tiens aussi tous particulièrement à remercier les professeurs Markus Jaton et Liechti Olivier, pour leur aide lors de problèmes de codes Java.

- Pour l'encadrement et le suivi de ce travail de diplôme, je remercie mon professeur responsable des branches de télécommunications, Dr. Robert Stephan.

- Mon frère Philippe pour la correction des fautes d'orthographe de ce document.

- Monsieur Nicolas Oberli pour avoir lu ce document dans son entier et participé à sa clarification.

- L'étudiant Mohammed Ahmed pour m'avoir mis à disposition le serveur *AAA Radius* de son réseau de test et pour avoir configuré mes machines clients *Windows* et *Linux* avec leur certificats.

RÉFÉRENCES

RFC

[RFC1305] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation", RFC 1305, March 1992.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC2104] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, March 1997

[RFC2784] D.Farinacci, T.li, S.Hanks, D.Meyer, Cisco Systems "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000

[RFC2865] C.Rigney, S.Willens, Livingston, A. Rubens, Merit, W.Simpson "Remote Authentification Dial In User Service (RADIUS)" RFC 2865, June 2000

[RFC2905] Vollbrecht J, Calhoun P, Farrell S, Gommans L, Gross G "AAA Authorization Applications Exemples" RFC2905, August 2000.

[RFC3024] Montenegro, G., "Reverse Tunneling for Mobile IP (revised)", RFC 3024, January 2001.

[RFC3046] M. Patrick, Motorola BCS "DHCP Relay Agent Information Option" RFC 3046, January 2001

[RFC3344] Perkins, C., "IP Mobility Support for IPv4" RFC 3344, August 2002.

[RFC3748] B. Aboba, Microsoft, L. Bunk, J. Vollbrecht, Sun, "Extensible Authentication Protocol" RFC 3748, June 2004

DRAFT

[DRAFT PMIP] Leung, K, "WiMAX Forum/3GPP2 Proxy Mobile IPv4 Draft-leung-mip4-proxy-mode-09", July 31, 2008

Sites Internet

[Dynamics] dynamics.sourceforge.net
Code Open Source offrant le service MIPv4
Auteur: Helsinki University of Technologie (HUT)
Dernière mise à jour, Octobre 2001

[JPCAP] http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html
Packetage Java contenant des fonctions d'accès réseau (sniffing)
Keita Fujii
Dernière mise à jour, Juin 2007

[Java DHCP] www.dhcp.org/javadhcp/
Classes Java contenant la structure de bas du DHCP.
Auteur: Jason Goldschmidt et Nick Stone
Dernière mise à jour, septembre 1999

[Java Diameter] http://i1.dk/
Package Java contenant les libraires pour crée le protocole Diameter
Auteur : Ivan Skytte Jørgensen
Dernière mise à jour: Novembre 2008

[OpenDiameter] http://www.opendiameter.org/
Auteur: Students of University of Murcia
Dernière mise à jour, décembre 2007

[OpenDiameter2] http://www.csg.uzh.ch/staff/morariu/opendiameter/
Auteur : CSG, Comunication System Group

[IETF] http://www.ietf.org/
The Internet Engineering Task Force

ABREVIATIONS

- A -

- *AAA* — *AAA* : *Autorization Authentication and Accounting*. Nom générique donné au protocole implémentant cette technologie. Il permet à des clients *Wireless* de s'authentifier par certificat ou avec un acompte et un mot de passe sur le réseau.
- *AAAF* — Partie du serveur pouvant être à distance, nécessaire à l'architecture *AAA*. Il est souvent utilisé comme *NAI* (*Network Access Server*) visible par l'utilisateur, le *F* signifie éloigné (*Foreign*). Il communique ses données sur un serveur centralisé *l'AAAH*
- *AAAH* — Serveur centralisé gérant toutes les demandes provenant des *AAAF*. Le *H* signifie maison (*Home*), il est donc l'entité suprême de décision de l'autorisation, de l'authentification et de l'acompte.
- *AP* — Point d'accès (*Access Point* : *AP*) est communément appelé borne *Wireless*. C'est l'élément réseau qui permet la connexion sans fil au reste du réseau. Il peut souvent se retrouver associé avec un routeur.
- *AR* — Le routeur d'accès (*Access Router* : *AR*) est l'élément réseau qui se trouve souvent chez le client ou le plus éloigné au niveau de la topologie réseau. Il permet la connexion de nouveaux clients.

- C -

- *CHAP* — Protocole d'authentification basé sur la résolution d'un « défi » (*Challenge Handshake Authentication Protocol*)
- *CoA* — La *Care of Address* est l'adresse *IP* ou le tunnel est désencapsulé dans le réseau éloigné (*FN*). Dans le cas du protocole *MIP* la *CoA* est toujours sur le *FA* puisque il est obligatoire que se soit lui qui désencapsule.

- D -

- *DHCP* — Protocole de configuration dynamique d'hôte (*Dynamic Host Configuration Protocol*)

- E -

- *EAP* — Protocole d'identification universel, fréquemment utilisé dans les réseaux sans fil (*Extensible Authentication Protocol*)

- F -

- *FA* *Foreign Agent* : Agent éloigné. Elément de l'architecture *MIP* discutant avec le client dans le réseau éloigné.
- *FN* *Foreign Network* : Réseau éloigné. Terme utilisé en mobilité pour spécifier le réseau éloigné visité par le client.
- *Foo* Mot essentiellement utilisé dans le jargon informatique signifiant tout et n'importe quoi.

- G -

- *GRE* Protocole utilisé pour créer des tunnels de communication (*Generic Routine Encapsulation*)

- H -

- *HA* *Home Agent* : Agent maître. Entité principale de l'architecture *MIP* auxquels se connectent tous les éléments mobiles.
- *HN* *Home Network* : Réseau maître. Réseau dans lequel se trouve le *Home Agent* et là où le client n'a pas besoin de gestion de mobilité.

- I -

- *IETF* Regroupement de personnes participant au développement d'Internet (*Internet Engineering Task Force*)
- *IP* Protocole de communication de réseau informatique (*Internet Protocol*)
- *IPCP* Protocole responsable de la configuration, de l'activation et de la désactivation des modules du protocole *IP* aux deux extrémités de la liaison Point à Point (*PPP*).

- M -

- *MD* Elément mobile représentant le client *IPv4* qui se voit hériter de la mobilité par le réseau.
- *MIP* *Mobile IP*. Protocole de mobilité décrit dans le *RFC 3344*.
- *MN* *Mobile Node* : Nœud Mobile. Elément du protocole *MIP* installé sur le client mobile.

- N -

- *NAS* *Network Access Server* (*NAS*), fait office d'intermédiaire entre l'utilisateur final et le serveur.

- O -

- *OS* Système d'exploitation (*Operation System* : *OS*) est chargé d'assurer la liaison entre les ressources matérielles, l'utilisateur et les applications
- *OSI* Modèle *OSI* (*Open Systems Interconnexions*), « Interconnexion de systèmes ouverts » est un modèle de communications entre ordinateurs

- P -

- *PDA* Petit ordinateur portable de la taille d'un téléphone portable (*personal digital assistant* : *PDA*)
- *PAP* Protocole d'authentification de mot de passe (*Password Authentication Protocol* : *PAP*), souvent utilisé pour authentifier un utilisateur à un serveur d'accès réseau (*NAS*)
- *PMA* *Proxy Mobile IP*. Elément du *Protocol Proxy Mobile IP* regroupant le *PMIP Client* et le *FA* et se situant dans le réseau éloigné (*FN*).
- *PMIP* *Proxy Mobil IP*. protocole homologue au *MIP*
- *PMIP Client* *Proxy Mobile IP Client*. Elément du protocole PMIP qui englobe la partie Client (*MN*) du protocole *MIP* à l'intérieur du *PMA*.
- *PPP* Le protocole point à point (*Point-to-Point Protocol* : *PPP*) est un protocole de transmission sur Internet qui permet d'établir une connexion de type liaison point à point entre deux hôtes.
- *PRRQ* *Proxy Registration Request*. Message de commande envoyé par le *PMA* au *HA*
- *PRRP* *Proxy Registration Reply*. Message de commande envoyé par le *HA* et reçu par le *PMA*.

- R -

- *RFC* Documents et normes concernant Internet gérés par *l'IETF* au format texte le plus simple (*Request For Comment* : *RFC*)

- T -

- *TTL* Le *TTL* (*Time To Live*) est le nombre de sauts (routeurs) qu'un paquet *IP* peut faire avant d'être détruit par le réseau.
- *TimeStamp* Temps en secondes depuis le 1 janvier 1900. Attention *Java* commence lui à partir de 1970.

- W -

- *Wireshark* Programme gratuit de capture de trame (*Sniffer*) développé pour *Windows* et *Linux*

A N N E X E S

Annexe A : Configuration *Open Dynamics* du *MN* (réseau *MIP*).

Annexe B : Configuration *Open Dynamics* du *FA* (réseau *MIP* & *PMIP*).

Annexe C : Configuration *Open Dynamics* du *HA* (réseau *MIP* & *PMIP*).

Annexe D : *WPA_Supplicant* et Interface (*MIP*)

Annexe E : *WPA_Supplicant* et Interface (*PMIP*)

Annexe F : Installation et lancement du *PMIP Client*.

Annexe G : Fichier config du *PMIP Client*

Annexe H : Fichier config du service *DHCP*

Annexe I : *Package jPMIPClient*
- I1 : *PMIPClientSocket*
- I2 : *PMIPClientMessage*
- I3 : *PMIPClientExtension*
  - I3.1 ExtensionTypeLengthValue
  - I3.2 ExtensionTypeLongFormat
  - I3.3 ExtensionTypeMobileID
  - I3.4 ExtensionTypeSecurityAuthentication
  - I3.5 ExtensionTypeShortFormat
- I4 : *PMIPReadConfig*
- I5 : FixeSetTTL

Annexe J : *Package jDHCP*
- J1 : DHCPSocket
- J2 : DHCPOption
- J3 : DHCPMessage p
- J4 : PMIPReadConfig

Annexe K : *Default Package*
- K1 : PMIP_Client
- K2 : PMIP_Worker
- K3 : PMIP_Instance
- K4 : DHCP_Dispatcher
- K5 : DHCP_Instance
- K6 : NAS
- K7 : MN_Data_Mobile

Annexe L : Rapport de travail hebdomadaire

Annexe M : Diagramme de Gantt

```
# $Id: dynmnd.conf,v 1.56 2001/10/20 13:36:07 jm Exp $
# Mobile Node configuration file
#
# Dynamic hierarchial IP tunnel
# Copyright (C) 1998-2001, Dynamics group
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License version 2 as
# published by the Free Software Foundation. See README and COPYING for
# more details.
#
##################################################################
##################################################################
#
# NOTE!
#       This is an example configuration file designed to give
#       perspective to the system configuration AND to provide
#       a basis for a working simple test environment.
#       The values of some of the parameters may not be the
#       same as the daemon's defaults, so don't get confused.
#
#       To get a minimal test working, you will need to check the
#       following items:
#         * MNHomeIPAddress
#         * HAIPAddress
#         * EnableFADecapsulation
#         * HomeNetPrefix (if using FA decapsulation or
#           dynamics HA address resolution)
#         * SPI and SharedSecret
#       The rest of the items should work with their preset values in
#       most cases and they can be used to fine tune the operations
#       after the basic operation have been tested successfully.
#
##################################################################
##################################################################
#
# The Mobile Nodes's IP address in the Home Network.
# If using AAA (see UseAAA below), home address can be set to 0.0.0.0 in order
# to request a home address from the AAA infrastructure. This requires that
# also MN NAI is configured.
MNHomeIPAddress 10.192.75.232

# The Mobile Node's Network Access Identifier (NAI) [RFC2794]
# If configured, this NAI is used in registration requests to identify the
# mobile user for AAA services.
#
# MNNetworkAccessIdentifier "user@example.com"

# UseAAA < TRUE | FALSE >. TRUE enables AAA extensions (key requests using
# material from AAA, HA and home address discovery using AAA, etc.). This
# requires that MN NAI and AAA related items below are configured.

# FALSE disables these extensions.
UseAAA FALSE

# The IP address of Mobile Node's Home Agent. In case of a private HA address
# this is the address of the surrogate HA. If the HA address is unknown, set
```

```
# this to 0.0.0.0 and make sure that HomeNetPrefix is correct for dynamic
# HA address resolution or use AAA to discover HA address. If the HA has
# multiple interfaces, this should be the address of the "public" interface,
# i.e., the one toward default gateway (it has to be reachable from the foreign
# networks).
HAIPAddress 10.192.75.231
# If the HA has more than one interfaces, HAIPAddress should be configured to
# be the one reachable from the Internet (i.e., from the foreign networks the
# MN may visit). To allows MN to detect other HA's interfaces, their IP
# addresses may be configured here. MN will use this list in addition to
# HAIPAddress when determining whether an agent advertisement is from its own
# HA (i.e., when MN is at home). Multiple lines containing different addresses
# may be used to configure more than one alternative HA address.
# AlternativeHAIPAddress 10.1.2.3
# AlternativeHAIPAddress 10.2.3.4

# AllowHomeAddrFromForeignNet < TRUE | FALSE >. TRUE allows AAA to assign
# a home agent and home address from the foreign network (assuming they are
# set to 0.0.0.0 above). FALSE means that both the home agent and the home
# address must be from the home domain.
AllowHomeAddrFromForeignNet FALSE

# The following configuration options PrivateHAIPAddress, PrivateHAIdentifier,
# and HANetworkAccessIdentifier are only used with home networks that use
# private IP addresses and a surrogate HA. In other cases they should be left
# commented.

# The private IP address of Mobile Node's Home Agent.
# Needed only, if surrogate HA is used.
# PrivateHAIPAddress 192.168.200.200

# The identifier for the private HA in SHA (unique 32-bit number)
# PrivateHAIdentifier 1

# Home Agent Network Access Identifier (NAI)
# If configured, this NAI is used to match the HA agent advertisements when
# a MN is determining whether it is at home or not. This is mainly used with
# private HA address that may not be globally unique.
#
# HANetworkAccessIdentifier "ha@example.com"

# EnableFADecapsulation < TRUE | FALSE >. TRUE enables a mode where
# the FA decapsulates the IP-within-IP encapsulated IP packets.
# FALSE disables this mode and sets the default mode where the
# MN decapsulates the IP-within-IP encapsulated IP packets.
# With FA decapsulation the MN uses its home address in the interface even in
# the foreign network and with MN decapsulation MN needs to acquire a
# co-located care-of address from the visited network (this needs an external
# program; see man pages for more information).
# The two modes cannot be used simultaneously.
EnableFADecapsulation TRUE

# Network address of home network (CIDR format: a.b.c.d/prefix_length)
# This is used with FA decapsulation and dynamics HA address resolution. If
# commented, the routing entry is not removed nor added. The home net entry
# may optionally be used with MN decapsulation - see MNDecapsRouteHandling
# option below.
```

```
#
# Example: 192.168.242.0/24
HomeNetPrefix 10.192.75.0/24
# Home net default gateway
# This entry can be used to force a gateway that the MN uses when it is
# at home. If this is left commented, the MN tries to use the default route
# that was in use when the program was started.
#
# HomeNetGateway 192.168.242.254

##########################################################
##########################################################
# a SPI (Security Parameter Index) must be defined for every MN.
# It is used for indexing the security association at the Home Agent.
SPI 1001
#
# The SharedSecret is provided as a HEX number string. The shared secret can
# also be given as a character string
# (e.g. character string "ABCDE" corresponds to HEX number string 4142434445).
# Note: RFC 2002 specifies that the default key size is 128 bits (i.e.
# 16 bytes or 32 hex 'characters'). Dynamics supports also other key lengths.
# This shared secret is used with the HA. This must be commented out when using
# AAA infrastructure for key generation. In this case, the AAA related items
# below must be configured.
# SharedSecret < shared secret >
# SharedSecret 016A352B2F235E
SharedSecret "********"
#
# Authentication algorithm
#    1: MD5/prefix+suffix (a.k.a. keyed-MD5) [RFC 2002]
#    4: HMAC-MD5 [RFC 2104]
#    5: SHA-1 [FIPS 180-1]
#    6: HMAC-SHA1 [RFC 2104]
# Note! MD5/prefix+suffix has known weaknesses and use of HMAC-MD5 is
# recommented. MD5/prefix+suffix algorithm is for backwards compatability with
# older versions that do not support more secure HMAC-MD5.
AuthenticationAlgorithm 4
#
# Replay prevention method:
#   0: none
#   1: time stamps
#   2: nonces
ReplayMethod 1
#
# Mobile Node may have optional security associations with Foreign
# Agents. If the security association exists an additional Mobile Node -
# Foreign Agent Authentication Extension is added to the registration requests.
#
# The following list contains the shared secrets indexed by SPI (and
# Foreign Agent IP address). The algorithm field specifies the method
# used for key distribution (see the list above). The format of the share
# secret field is identical to the one used with the MN-HA security
# association list above.
#
FA_SECURITY_BEGIN
# SPI         FA IP         Alg.   Shared Secret
#2001         192.168.0.1   4      0123456789ABCDEF
```

```
#2002        192.168.0.2  4       "eslkfj89jr3hduh3R!as"
FA_SECURITY_END


# MN-AAA Authentication and Challenge/Response [RFC3012]

# If the MN does not have a security association with an FA, it may use AAA
# infrastructure for authentication. If this is used, also MN NAI
# ('MNNetworkAccessIdentifier' above) should be configured.

# SPI to be used in MN-AAA authentication.
# Reserved SPI values:
#   2 = CHAP_SPI, CHAP style authentication using MD5 [RFC 3012]
#   3 = MD5/prefix+suffix [draft-ietf-mobileip-aaa-key-03.txt]
#   4 = HMAC MD5 [draft-ietf-mobileip-aaa-key-03.txt]
# MN-AAA-SPI 12345

# Shared secret for MN-AAA authentication (see 'SharedSecret' above for format
# instructions)
# MN-AAA-SharedSecret "test"

# Algorithms to be used for MN-AAA authentication and key generation
#   1 = MD5/prefix+suffix (RFC 2002)
#   2 = RADIUS authentication (Sec. 8 of RFC 3012)
#   3 = MD5/prefix+suffix (RFC 2002)  (alias for 1 above)
#   4 = HMAC-MD5 (Sec. 6 of RFC 3012; RFC 2104)
#   5 = SHA-1 (FIPS 180-1)
#   6 = HMAC-SHA1 (RFC 2104)
# Note: with algorithm 2, 'MN-AAA-SPI' should be set to reserved number
# CHAP_SPI (default: 2).
# MN-AAA-AuthenticationAlgorithm 4
# MN-AAA-KeyGenerationAlgorithm 4


##################################################################
##################################################################
# TunnelingMode < 1 | 2 | 3 | 4 >
# The packets between the MN and a Correspondent Node (CN) can be routed using
# different routes. This option can be used to select, which mode will be
# selected.
# Possible values:
# 1 = automatic, prefer reverse tunnel (i.e. bi-directional tunnel)
# 2 = automatic, prefer triangle tunnel (i.e. tunnel only in CN->MN direction)
# 3 = accept only reverse tunnel
# 4 = accept only triangle tunnel
TunnelingMode 1

# When MN can get its own co-located care-of address and use reverse tunneling,
# the normal method is to set the default route to the tunnel. This means that
# all the packets destined to other networks than the current subnet in the
# visited network are send via the HA. If the co-located COA is public, it can
# be used for sessions that do not need constant IP address(e.g.most of the
# web browsing). The following configuration option specifies the routing
# operation that is used with the co-located COA.
# Possible values:
#   0 = set default route to the tunnel
#   1 = set only the home net route to the tunnel (the above HomeNetPrefix
```

```
#       options must be set)
#   2 = do not change the routing entries (i.e. some external means must be
#       used to direct traffic to the tunnel,e.g.manually adding host route
#       to a specific host)
MNDecapsRouteHandling 0

# DefaultTunnelLifetime is the lifetime suggested in registration
# The lifetime is defined in seconds, default value is 300.
# The request timer will be set according to this value. If the FA's agent
# advertisment has a smaller time, it is used instead.
# Special case: 65535 (or more) seconds means unlimited time (the binding will
# not expire)
# MNDefaultTunnelLifetime [ seconds ]
MNDefaultTunnelLifetime 300

# UDP port to be used for sending registration requests
# Port 434 is allocated for Mobile IP signaling and this should not be changed
# unless the network is known to use some other port (i.e. all the FAs and HAs
# must have the same port configured).
UDPPort 434

# Socket priority for signaling sockets (UDP) can be set with SO_PRIORITY to
# allow easier QoS configuration. If this argument is set, the given value is
# used as a priority for the signaling socket. E.g. CBQ class can be used to
# make sure that signaling is not disturbed by other traffic on a congested
# link.
# This feature is still undocumented and can be left commented.
#
# SocketPriority 1

# The log messages are written through syslog service. The facility to be
# used defaults to LOG_LOCAL0, but it can be set with this parameter
# to any of the possible facilities (LOG_AUTHPRIV, LOG_DAEMON, and so on).
# The processing of log messages is defined in /etc/syslog.conf file.
SyslogFacility LOG_DAEMON

# Ignore these interfaces. No agent advertisements are received nor
# agent solicitations sent for these interfaces.
IGNORE_INTERFACES_BEGIN
lo
dummy0
tunl0
gre0
IGNORE_INTERFACES_END

# Other programs may set routing entries so that the data connexion may
# fail. The MN can try to enforce the routes that it believes should be used.
# This operation should currently be used only with FA decapsulation.If the
# route enforcement is activated the MN prevents certain route changes.
EnforceRoutes FALSE

# MN can be instructed to poll for current AP address when using a wireless
# LAN driver that supports wireless extensions. This can be used to speed up
# handoffs when using managed mode (BSS).
# Polling interval is configured in micro seconds
# (i.e., 1000000 equals to 1 second)
# -1 = AP polling disabled
```

```
APPollingInterval -1

# MN can be instructed to send periodic agent solicitations to find new FAs.
# Normally, MN uses agent solicitations when it does not have a valid agent
# advertisement. Periodic solicitation occurs even if the connexion seems to
# be up. This will cause more broadcast messages and is thus disabled in
# default configuration, but it can speed up handoffs in some environments.
# Solicitation interval is configured in micro seconds (usec)
# (i.e., 1000000 usec equals to 1 second). A rnadom time between 0 and 0.5
# second will be added to solicitation intervals to prevent unwanted
# synchronization of broadcast messages. In addition, solicitations will not be
# send more often than once per second, so this interval should not be
# configured to be less than 1000000 usec.
# -1 = Periodic agent solicitation disabled
SolicitationInterval -1

##################################################################
##################################################################
# Mobile Nodes use unix domain sockets to communicate through their API
# interfaces.
# The group and owner must be names as strings, no groupIDs or userIDs are
# allowed. The file permissions are set in octal values like in chmod(1).
# The configuration parameters of the two API sockets are as follows:
MNAPIReadSocketPath "/var/run/dynamics_mn_read"
MNAPIReadSocketGroup "root"
MNAPIReadSocketOwner "root"
MNAPIReadSocketPermissions 0666
#
MNAPIAdminSocketPath "/var/run/dynamics_mn_admin"
MNAPIAdminSocketGroup "root"
MNAPIAdminSocketOwner "root"
MNAPIAdminSocketPermissions 0700
#
# Every configuration file must end to the keyword 'END'.
END
```

```
# $Id: dynfad.conf,v 1.64 2001/10/20 13:36:07 jm Exp $
# Foreign Agent configuration file
#
# Dynamic hierarchial IP tunnel
# Copyright (C) 1998-2001, Dynamics group
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License version 2 as
# published by the Free Software Foundation. See README and COPYING for
# more details.
#
##########################################################
##########################################################
#
# NOTE!
#       This is an example configuration file designed to give
#       perspective to the system configuration AND to provide
#       a basis for a working simple test environment.
#       The values of some of the parameters may not be the
#       same as the daemon's defaults, so do not get confused.
#
##########################################################
##########################################################
#
# Interfaces to be used for Mobile IP services
# interface: name of the interface, e.g. eth0
# type:
#    1 = both upper and lower direction
#    2 = only upper direction (to upper FA / HA)
#    3 = only lower direction (to lower FA / MN)
#    Note: Only one interface can be used for upper direction, but
#    multiple interfaces can be used for lower direction.
# agentadv:
#    0 = do not send agent advertisements without agent solicitation
#    1 = send agent advertisements regularly
#   -1 = do not send any (even solicited) agent advertisements
# interval: number of seconds to wait between two agentadvs
#         (if allowed for this interface)
# force_IP_addr: local address to be forced for this interface
#               (can be used to select one of the multiple virtual
#               addresses); if not entered, the primary address of the
#               interface is used
#
# In the example below, interface "eth0" can be used for both upper and lower
# directions, Agent Advertisements are sent regularly with the interval
# of 30 s, and the primary address of the eth0 interface is used.
# Correspondingly, interface "eth1" would allow only lower direction
# connexions (connexions with MNs or lower FAs) with periodical
# Agent Advertisements with an interval of 20 s, and a specific IP address
# would be forced to the interface "eth1".
#
# The last entry that is of type upper (or both upper and lower), is
# used to send messages in the upper direction.
#
INTERFACES_BEGIN
# interface  type  agentadv  interval  force_IP_addr
eth0        2      1        30
```

```
eth1        3       1        30
#eth1       3     1        20       192.168.240.2
INTERFACES_END
##################################################################
##################################################################

# Network Access Identifier (NAI) of this FA
# Unique identifier for this FA. A macro [interface] can be used to get
# the hardware address of an interface in dot-separated format.
NetworkAccessIdentifier "[eth0]@example.com"

# Address of the highest FA
# This address is used in the communication with the HA and it is advertised
# in agent advertisement messages. This should be from the "public side"
# interface of the FA (i.e., the interface that is toward HA or the default
# gateway).
# If this FA is the highest FA for some organization, use its address here.
# In this case, the address would most probably be from the interface that
# is configured for upper direction (type 1 or 2 in the interface list above).
HighestFAIPAddress 10.192.74.81

# Highest FA public key extension hash
# This hash code is used to protect the public key from active
# man-in-the-middle style attacks. Its use is optional, but recommended. If
# this hash is configured, the FA broadcasts it in the agent advertisements
# and the MNs send it in their registration requests to the HA (protected with
# MN-HA authentication extension).
# The hash code is printed by the rsakeygen utility and if used, it must be
# the hash of this organization's highest FA.
# HighestFAPubKeyHash 78439F9EA1FE32EDD8CE2028062DC96A

# Address of the upper FA
# This is the address of the FA to which the requests are forwarded
# on they way to the Home Agent.
# If this is the same as the FA's own IP address,
# then this FA is really the highest FA and the requests are forwarded
# directly to the Home Agent.
UpperFAIPAddress 10.192.74.81

# HighestFA < TRUE | FALSE >
# Whether this FA is the highest FA (i.e. it does not have any upper FAs and
# it communicates directly with the Home Agents).
HighestFA TRUE

# UDP Port that this FA listens to for signaling messages
UDPPort 434
# The port to be used in signaling with the upper FA
UpperFAUDPPort 434
# The port to be used in signaling with the HA
HAUDPPort 434

# RFC 2344 style tunnel hijacking protection requires that the MN uses TTL
# value 255 on all registration request messages and the FAs check this.
# Since RFC 2002 compliant MN implementations do not necessarily set the TTL
# to 255, this may limit the access of those MNs. This option can be used to
# change the checking of the TTL field in the IP header.
# 0 = no TTL checking (i.e. accept any value)
```

```
# 1 = check the TTL only in registration requests that ask for reverse
#     tunneling (i.e. the MN should be RFC 2344 compliant and use the TTL 255)
# 2 = check the TTL on every registration requests (this might deny the access
#     of some MN implementations)
RegistrationTTLCheck 1

# Socket priority for signaling sockets (UDP) can be set with SO_PRIORITY to
# allow easier QoS configuration. If this argument is set, the given value is
# used as a priority for the signaling socket. E.g. CBQ class can be used to
# make sure that signaling is not disturbed by other traffic on a congested
# link.
# This feature is still undocumented and can be left commented.
#
# SocketPriority 1

# Tunnel interface name (FA will use names line TUNL0, TUNL1, ...)
TunnelDevice "TUNL"

# The start of the range of routing tables that this FA can use.
# Linux kernel 2.2.X has 256 routing tables (0 .. 255), but 0, 253, 254, and
# 255
# are reserved.
RoutingTableStart 1
# The end of the range of routing tables that this FA can use. Defaults to 252.
RoutingTableEnd 252
# Available routing table range defaults to 1-252.

##################################################################
##################################################################
# FA must keep track of the authorized network addresses.
# This list can be used to limit the allowed IP addresses from which the
# registration requests can be sent (lower FAs or MNs).
#
# To allow classless subnetting, also the netmask is included in the list.
# AUTHORIZEDNETWORKS is a list that has a pair
# [ networkaddress ]/[ netmask ]
# on each row separated by the line breaks.
# Here is an example:
#
#AUTHORIZEDNETWORKS_BEGIN
# [ networkaddress ]/[ netmask ]
#192.168.240.0/255.255.255.0
#192.168.240.0/24
#AUTHORIZEDNETWORKS_END
#
# This does not limit the connexions by IP address
AUTHORIZEDNETWORKS_BEGIN
# [ networkaddress ]/[ netmask ]
0.0.0.0/0.0.0.0
AUTHORIZEDNETWORKS_END
#
# Whether or not this FA allows MNs to be connected directly (i.e. whether it
# can be the lowest FA)
AllowMobileNodes TRUE
#
##################################################################
##################################################################
```

```
# A Foreign Agent may have optional security associations with other nodes
# (FA, HA, MN).
#
# If the security association exists the session key can be
# encrypted with the help of shared secret and thus man-in-the-middle
# style attacks can be prevented. If no security association is set
# for a certain Foreign Agent - Foreign Agent pair, public key encryption
# (RSA) is used.
#
# The following list contains the shared secrets indexed by SPI (and
# IP address of the other node).
#
# The node field specifies the type of the node. It used to select the
# approriate authentication extension type.
#    1 = FA
#    2 = HA
#    3 = MN
# The algorithm field specifies the method used for authentication and
# key distribution:
#    1: MD5/prefix+suffix (a.k.a. keyed-MD5) [RFC 2002]
#    4: HMAC-MD5 [RFC 2104]
#    5: SHA-1 [FIPS 180-1]
#    6: HMAC-SHA1 [RFC 2104]
# Note! MD5/prefix+suffix has known weaknesses and use of HMAC-MD5 is
# recommented. MD5/prefix+suffix algorithm is for backwards compatability with
# older versions that do not support more secure HMAC-MD5.
#
# Shared secret can be given as a HEX code string, i.e. two characters (0-F)
# correspond to one octet. The shared secret can also be given as a character
# string (e.g. "ABCDE" corresponds to 4142434445).
# Note: RFC 2002 specifies that the default key size is 128 bits (i.e.
# 16 bytes or 32 hex 'characters'). Dynamics supports also other key lengths.
#
FA_SECURITY_BEGIN
# SPI         Node IP             Node  Alg.    Shared Secret
#2001         192.168.0.1  1      4     0123456789ABCDEF
#2002         192.168.0.2  2      4     "eslkfj89jr3hduh3R!as"
FA_SECURITY_END
#
# RSA key file
FAKeyFile "/etc/dynfad.key"


# Mobile IPv4 Challenge/Response (RFC 3012)

# Dynamics supports Mobile IPv4 Challenge/Response protocol as an optional
# addition to the Mobile IP registration. This is not used in a default setup
# because the extensions used for challenges will prevent MNs that do not
# support this addition from using the FA at all.
EnableChallengeResponse FALSE

# Number of last advertised challenges that will be accepted (default value
# given in RFC 3012: 2).
ChallengeWindow 2

# Length of the challenge to be used (in bytes; 0 .. 255)
ChallengeLength 4
```

```
# Whether the FA requires the challenge to be present in every registration
# request (if not present, request will be denied with MISSING_CHALLENGE
# error). If 'EnableChallengeResponse' is TRUE, the challenge is required from
# MNs which do not have a security association with the FA. With
# 'RequireChallenge' TRUE, the challenge is required also from the MNs that
# have the security association.
RequireChallenge FALSE

# Whether the FA adds new challenge to all the registration replies.
ChallengeInRegReply TRUE


# AAA Keys for Mobile IP (draft-ietf-mobile-aaa-key-07.txt)

# The FA can be configured to deny registration replies that do not have
# an Unsolicited MN-FA Key Material From AAA extension for an MN that does
# not have a security association with the FA.
RequireMNFASecAssoc FALSE

##################################################################
##################################################################

# The maximum number of tunnels (confirmed bindings) going through this FA
# The default value for MaxBindings is 20.
# If more than MaxBindings Mobile Nodes try to register, the new registrations
# are refused.
MaxBindings 100

# The maximum number of pending registration requests (unconfirmed bindings)
# the FA is willing to maintain (typically 5 according to rfc2002-bis draft).
# Additional registrations will be rejected until at least one of the pending
# registrations has been completed or has timed out.
# Zero means no limit on pending registration requests.
MaxPending 5

# The number of seconds after which pending registration requests MAY be
# deleted. Zero means do not force pending registration request deletion
# before their requested lifetime has expired.
DeletePendingAfter 7

# EnableFADecapsulation < TRUE | FALSE >
EnableFADecapsulation TRUE

# Triangle tunnel means that the packages to MNs are sent via the HA, but
# packages from MN are routed directly (i.e. FA use normal IP routing).
# EnableTriangleTunneling < TRUE | FALSE >
EnableTriangleTunneling FALSE

# Reverse tunnel means bi-directional tunneling in which both the packages
# from and to MN are send via HA
# EnableReverseTunneling < TRUE | FALSE >
EnableReverseTunneling TRUE

# Force FA to use reverse tunneling even if triangle tunneling is requested.
ForceReverseTunneling FALSE
```

```
# FA may require registration even from those MNs which have acquired a
# co-located care-of address. This option selects whether the agent
# advertisements messages have 'Registration required' flag or not
RegistrationRequired TRUE

# DefaultTunnelLifetime is the maximum lifetime advertised for this FA.
# This should not be greater than any of the maximum lifetimes configured
# for upper FAs (i.e. best to use the same maximum for whole FA organization).
# The lifetime is defined in seconds, default value is 400.
# The request timer will be limited with this value.
# Special case: 65535 (or more) seconds mean unlimited time (the binding will
# not expire)
FADefaultTunnelLifetime 600

# FA uses a packet socket for raw L2 header access. When sending registration
# messages, this is only used between the lowest FA and the MN. Current code
# does not implement fragmentation and packets larget than the used MTU are
# thus probably dropped. FA can be configured to not use packet socket when
# sending frames, but this may require broadcast ARP for MN's home address
# in the foreign network. This is against RFC 2002, so it should be used only
# if the packet socket does not work.
# Possible values for PacketSocketMode:
#    0 = use packet socket when sending registration replies to MN (default)
#    1 = do not use packet socket at all for sending registration messages
PacketSocketMode 0

# The log messages are written through syslog service. The facility to be
# used defaults to LOG_LOCAL0, but it can be set with this parameter
# to any of the possible facilities (LOG_AUTHPRIV, LOG_DAEMON, and so on).
# The processing of log messages is defined in /etc/syslog.conf file.
SyslogFacility LOG_DAEMON

# Foreign Agents use unix domain sockets to communicate through their API
# interfaces.
# The group and owner must be names as strings, no groupIDs or userIDs are
# allowed. The file permissions are set in octal values like in chmod(1).
# The configuration parameters of the two API sockets are as follows:
FAAPIReadSocketPath "/var/run/dynamics_fa_read"
FAAPIReadSocketGroup "root"
FAAPIReadSocketOwner "root"
FAAPIReadSocketPermissions 0766
#
FAAPIAdminSocketPath "/var/run/dynamics_fa_admin"
FAAPIAdminSocketGroup "root"
FAAPIAdminSocketOwner "root"
FAAPIAdminSocketPermissions 0700
#
# Every configuration file must end with the keyword 'END'.
END
```

```
# $Id: dynhad.conf,v 1.39 2001/10/20 13:36:07 jm Exp $
# Home Agent configuration file
#
# Dynamic hierarchial IP tunnel
# Copyright (C) 1998-2001, Dynamics group
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License version 2 as
# published by the Free Software Foundation. See README and COPYING for
# more details.
#
##########################################################
##########################################################
#
# NOTE!
#       This is an example configuration file designed to give
#       perspective to the system configuration AND to provide
#       a basis for a working simple test environment.
#       The values of some of the parameters may not be the
#       same as the daemon's defaults, so don't get confused.
#
#
##########################################################
##########################################################
#

# Interfaces to be used for Mobile IP services. Note that you have to configure
# each interface that may receive or send registration messages.
# interface: name of the interface, e.g. eth0
# ha_disc:
#    0 = do not allow dynamic HA discovery
#    1 = allow dynamic HA discovery with broadcast messages
# agentadv:
#    0 = do not send agent advertisements without agent solicitation
#    1 = send agent advertisements regularly
#   -1 = do not send any (even solicited) agent advertisements
# interval: number of seconds to wait between two agentadvs
#         (if allowed for this interface)
# force_IP_addr: local address to be forced for this interface
#               (can be used to select one of the multiple virtual
#               addresses); if not entered, the primary address of the
#               interface is used
INTERFACES_BEGIN
# interface  ha_disc  agentadv  interval  force_IP_addr
eth0        1       1        10
#eth1       1       1        20       192.168.240.2
INTERFACES_END

# Network Access Identifier (NAI) of this HA
# Unique identifier for this HA. A macro [interface] can be used to get
# the hardware address of an interface in dot-separated format.
# This is needed, if private address space is used in the home network.
# NetworkAccessIdentifier "[eth0]@example.com"

# Surrogate HA IP Address
# This is only needed, if private address space and a surrogate HA are used in
# the home network.
```

```
# SHAIPAddress 10.10.10.10

# Private HA Identifier at SHA
# Unique identifier (32-bit number) at SHA for this private HA.
# This is only needed, if private address space and a surrogate HA are used in
# the home network.
# PrivateHAIdentifier 1

# UDP port to listen for registration requests
# The default is 434
UDPPort 434

# Socket priority for signaling sockets (UDP) can be set with SO_PRIORITY to
# allow easier QoS configuration. If this argument is set, the given value is
# used as a priority for the signaling socket. E.g. CBQ class can be used to
# make sure that signaling is not disturbed by other traffic on a congested
# link.
# This feature is still undocumented and can be left commented.
#
# SocketPriority 1

# MaxBindings can be used to restrict the maximum number of Mobile Nodes
# that are concurrently attached to this Home Agent.
# The default is 20.
MaxBindings 20

# The default tunnel lifetime is suggested also by the HA.
# The default lifetime is 500.
HADefaultTunnelLifetime 600

# The Registration error reply interval should be restricted to
# avoid system overloading situations when receiving too much
# incorrect Registration Reply messages.
# The default value for RegErrorReplyInterval is 1 second.
RegErrorReplyInterval 1

# Triangle tunnel means that the packages to MNs are send via the HA, but
# packages from MN are routed directly (i.e. FA use normal IP routing).
# EnableTriangleTunneling < TRUE | FALSE >
EnableTriangleTunneling FALSE

# Reverse tunnel means bi-directional tunneling in which both the packages
# from and to MN are send via HA
# EnableReverseTunneling < TRUE | FALSE >
EnableReverseTunneling TRUE

##################################################################
##################################################################
# The Home Agent needs to know what kind of security parameters each
# authorized Mobile Node uses. that is why there is a tbale that maps
# (in many-to-many relationship) SPI numbers, or SPI-number ranges to
# IP adresses - or IP-address ranges defined by network adresses and
# netmasks. The netmask may be defined in two ways: either in
# "bit offset notation" (the third row in the example) or in the
# "dotted decimal notation" (the fifth row in the example below).
# The list of Mobile Node information is separated between two
# keywords: AUTHORIZEDLIST_BEGIN and AUTHORIZEDLIST_END.
```

```
#
# < SPI | SPI-range      IP | network/netmask  >
# Example:

AUTHORIZEDLIST_BEGIN
# SPI         IP
#1000          192.168.240.2
#1001          192.168.240.3
#1002          0.0.0.0/0
#11000-11999     192.168.241.4
#12000         192.168.250.0/255.255.255.0
#13000-14000     192.168.251.0/28
1001          10.192.75.232
AUTHORIZEDLIST_END

# The Home Agents needs a security association for each authorized Mobile
# Node. The association includes following information.
#
# SPI (Security Parameter Index): a key for the other fields.
#
# Authentication Algorithm:
#    1: MD5/prefix+suffix (a.k.a. keyed-MD5) [RFC 2002]
#    4: HMAC-MD5 [RFC 2104]
#    5: SHA-1 [FIPS 180-1]
#    6: HMAC-SHA1 [RFC 2104]
# Note! MD5/prefix+suffix has known weaknesses and use of HMAC-MD5 is
# recommented. MD5/prefix+suffix algorithm is for backwards compatability with
# older versions that do not support more secure HMAC-MD5.
#
# Replay Protection Method:
#    0: none
#    1: timestamps
#    2: nonces
#
# Timestamp tolerance indicates how many seconds the MN's timestamp can differ
# from the HA's clock. 7 seconds is the recommended default value. This
# tolerance is checked only when timestamps are used for replay protection.
#
# The maximum lifetime for the binding is given in seconds.
# Special case: 65535 (or more) seconds means unlimited time (the binding will
# not expire)
#
# Shared Secret: a secret data known by MN and HA. It can be given as
# a HEX code string, i.e. two characters (0-F) correspond to one octet.
# The shared secret can also be given as a character string (e.g.
# "ABCDE" corresponds to 4142434445).
# Note: RFC 2002 specifies that the default key size is 128 bits (i.e.
# 16 bytes or 32 hex 'characters'). Dynamics supports also other key lengths.
#
# The SPI is the key identificator for the rest of the security parameters
# on the same line. SPI number ranges may be assigned the same security
# parameters.
#
# The list of Mobile Node information is separated between two
# keywords: SECURITY_BEGIN and SECURITY_END.
#
SECURITY_BEGIN
```

```
#       auth.   replay  timestamp       max             shared
# SPI   alg.    meth.   tolerance       lifetime        secret
1001   4        1       120             600             "********"
#1002    4       2       60              120             01020304050607
#10000   4       1       60              300             016A352B2F235E
#10001   4       1       120             180             0EF42BD234ECCAA2
SECURITY_END
#
##################################################################
##################################################################
# Home Agent may have optional security associations with Foreign
# Agents. If the security association exists the session key can be
# encrypted with the help of shared secret and thus man-in-the-middle
# style attacks can be prevented. If no security association is set
# for a certain Foreign Agent - Home Agent pair, public key encryption
# (RSA) is used.
#
# When private address space is used, this list must have a security
# association with the surrogate HA instead of the FAs. Possible security
# associations with the FAs are then configured to the SHA.
#
# The following list contains the shared secrets indexed by SPI (and
# Foreign Agent IP address). The algorithm field specifies the method
# used for authentication and key distribution:
#    1: MD5/prefix+suffix (a.k.a. keyed-MD5) [RFC 2002]
#    4: HMAC-MD5 [RFC 2104]
#    5: SHA-1 [FIPS 180-1]
#    6: HMAC-SHA1 [RFC 2104]
# The format of the share secret field is identical to the one used with the
# MN-HA security association list above.
#
FA_SECURITY_BEGIN
# SPI           FA IP           Alg.    Shared Secret
#2001           192.168.0.1     4       0123456789ABCDEF
#2002           192.168.0.2     4       "eslkfj89jr3hduh3R!as"
FA_SECURITY_END
#
# The Highest FA public key can be protected from man-in-the-middle style
# attacks between the HFA and the HA with hash code. The use of this hash
# is optional, but recommented. The HA can have different ways of checking
# the hash code.
# Methods:
#    0: skip the hash code completely (not recommented)
#    1: if the hash code is received, check the public key with it
#    2: require the correct hash code for every registration message
#       with a public key (this may prevent the use of some organizations
#       which do not advertise the hash code)
PublicKeyHashMethod 1
#
##################################################################
##################################################################

# The log messages are written through syslog service. The facility to be
# used defaults to LOG_LOCAL0, but it can be set with this parameter
# to any of the possible facilities (LOG_AUTHPRIV, LOG_DAEMON, and so on).
# The processing of log messages is defined in /etc/syslog.conf file.
SyslogFacility LOG_DAEMON
```

```
# Home Agents (and Foreign Agents) use unix domain sockets
# to communicate through their API interfaces.
# The group and owner must be names as strings, no groupIDs or userIDs are
# allowed. The file permissions are set in octal values like in chmod(1).
# The configuration parameters of the two API sockets are as follows:
HAAPIReadSocketPath "/var/run/dynamics_ha_read"
HAAPIReadSocketGroup "root"
HAAPIReadSocketOwner "root"
HAAPIReadSocketPermissions 0766
#
HAAPIAdminSocketPath "/var/run/dynamics_ha_admin"
HAAPIAdminSocketGroup "root"
HAAPIAdminSocketOwner "root"
HAAPIAdminSocketPermissions 0700
#
# Every configuration file must end to the keyword 'END'.
END
```

WPA_SUPPLICANT (Config MIP)

```
network={
        ssid="mobileIPB03"
        # this id_str will notify /sbin/wpa_action to 'ifup FA_B03'
        id_str="FA_B03"
        psk="**********"
}

network={
        ssid="mobileIPB05"
        # this id_str will notify /sbin/wpa_action to 'ifup FA_B05'

        id_str="FA_B05"
        psk="**********"
}
```

INTERFACE (Config MIP)

```
auto lo
iface lo inet loopback

iface wlan0 inet manual
        wpa-driver wext
        wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf

# id_str="FA_B03"
iface FA_B03 inet static
      address 10.192.75.232
      netmask 255.255.255.0
      network 10.192.75.0
      broadcast 10.192.75.255

# id_str="FA_B05"
iface FA_B05 inet static
      address 10.192.75.232
      netmask 255.255.255.0
      network 10.192.75.0
      broadcast 10.192.75.255
```

WPA_SUPPLICANT (Config PMIP)

```
network={
        ssid="mobileIPB03"
        # this id_str will notify /sbin/wpa_action to 'ifup FA_B03'
        id_str="FA_B03"
        key_mgmt=WPA-EAP
        eap=LEAP
        identity="yann"
        password="ldap"
}

network={
        ssid="mobileIPB05"
        # this id_str will notify /sbin/wpa_action to 'ifup FA_B05'
        id_str="FA_B05"
        key_mgmt=WPA-EAP
        eap=LEAP
        identity="yann"
        password="ldap"
}
```

INTERFACE (Config PMIP)

```
auto lo
iface lo inet loopback

iface wlan0 inet manual
        wpa-driver wext
        wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf

# id_str="FA_B03"
iface FA_B03 inet dhcp

# id_str="FA_B05"
iface FA_B05 inet dhcp
```

Installation

Actuellement le code est fourni uniquement sur le CD, en fin de ce document. Le programme compilé en .jar exéctuable se trouve dans le fichier compressé PMIPClient.zip dans le dossier /code/PMIPClient/.

Le zip décompressé (en utilisant Winzip ou équivalent) contient un fichier PMIPClient.jar exéctuable et un dossier config contenant deux fichiers. Ce dossier doit impérativement se trouver dans le même dossier que le PMIPClient.jar pour que ce dernier puisse fonctionner.

Lancement du programme

Cette version du programme ne tourne actuellement que sur Windows. Le prérequis avant le démarrage est de changer le TTL par défaut dans la base de registre (se réferer au paragraphe 5.5.1).

Il est important de paramétrer correctement les programmes via les fichiers de configuration situés dans le dossier config situé au même niveau que le jar.

Lorsque ceci est fait, démarrer une session *DOS* depuis l'invitation de commande depuis : Démarrer/Executer/cmd. Puis depuis le dossier dans lequel se trouve le fichier jar, taper la commande suivante : java –jar PMIPConfig.jar

```
# PMIPCondif.txt
# Version 1
#
# PMIP Client configuration file
#
################################################################################
##
#
# DefaultTunnelLifetime is the lifetime suggested in registration
# The lifetime is defined in seconds, default value is 300.
# The request timer will be set according to this value. If the FA's agent
# advertisment has a smaller time, it is used instead.
# Special case: 65535 (or more) seconds means unlimited time (the binding
will
# not expire)
# MNDefaultTunnelLifetime [ seconds ]
MNDefaultTunnelLifetime 40

# UDP port to be used for sending registration requests
# Port 434 is allocated for Mobile IP signaling and this should not be
changed
# unless the network is known to use some other port (i.e. all the FAs and
HAs
# must have the same port configured).
#UDPPort 434

# TTL of the IP package. Set to 255 by default due to a security problem.
#TTL 255

# InetAdress of the FA interface that comunicate with the Mobile Node (MN).
# Set to 127.0.0.1 on the loopback adress by default.
FA_InetAddress 192.168.0.1

# Hardware Address of the FA interface that comunicate with the Mobile Node
(MN).
# Set to 00:00:00:00:00:00 on the loopback adress by default.
FA_MACAddress 00:0b:db:76:82:6f

# InetAdress of the interface send packet to the FA.
# Set to 127.0.0.1 on the loopback adress by default.
InetAdress 192.168.0.2

# COAddress is the Care Of Adress of the end of the tunneling. In the case
of the PMIP
# The Draft specify that whe MUST Desencapsulate the tunnel on the FA and
not on the MN
# Thos addresse is the same as the InetAdress of the FA interface that
communicate with the HA
# Default is 0.0.0.0
COAddress 10.192.74.81

# TimeBeforeKillThread is the time that the Thread PMIP wait the incoming
request DHCP
# of the Client. If no DHCP request arrived, the thread stop after this
Time
# The default Time is 300 sec (5 min)
TimeBeforeKillThread 300
```

```
# DHCPCondif.txt
# Version 1
#
# PMIP Client configuration file
#
################################################################################
##
#
# Its the IP address of the interface that communicate with the DHCP
client.
# This value will be returned in the Option DHCP field 54 Serveur
Identifier
# Set by default to "0.0.0.0"
InterfaceIP_DHCP 192.168.0.2

# Its the time of lease the bail IP address of the DHCP .
# This value will be returned in the Option DHCP field 51 Lease Time
# Set by default to "4294967295" that the max value on 4 byte
LeaseTime 2147483647

# IP address of the default gateway to return to the client
# Bye default the value is set to "0.0.0.0"
DefaultGateWay 192.168.0.1
```

```
package jPMIPClient;

import java.io.IOException;
import java.net.*;

import jpcap.JpcapCaptor;
import jpcap.JpcapSender;
import jpcap.NetworkInterface;
import jpcap.NetworkInterfaceAddress;
import jpcap.packet.EthernetPacket;
import jpcap.packet.UDPPacket;

/**
 * This class represents a Socket for sending PMIP Client Messages
 * @author Hercher Yann
 * @version 1  29/10/2008
 * @see java.net.MulticastSocket
 */
public class PMIPClientSocket extends DatagramSocket {

    static protected int PACKET_SIZE = 1500; // default MTU for ethernet
    private int freePort;

      /** Constructor who createCreate a Socket on a random free port and keep them during all the
       *  transaction
       *  Actualy the random free port is not implemented
       *  @throws IOException
    */
    public PMIPClientSocket() throws IOException {
      super();
    }
```

```
 /**
 * Sends a PMIPClientMessage object to a predifined host.
 * @param inMessage well-formed PMIPClientMessage to be sent to a FA or HA
 */
public synchronized void send(PMIPClientMessage inMessage) {

 byte data[] = new byte[PACKET_SIZE];
 data = inMessage.externalize();
 InetAddress dest;
 int port;
      try {
           //dest = InetAddress.getByName(PMIPReadConfig.getInetAdress());
           //port = PMIPReadConfig.getUDPPort();

           dest = InetAddress.getByName("192.168.0.1");
           port = 434;

           // Create the packet
           DatagramPacket outgoing = new DatagramPacket(data, data.length, dest,port);

           //FIX !!! change the default OS TTL
           //FixeSetTTL.setNewTTL();
           send(outgoing);
           //FixeSetTTL.setDefaultTTL();
      }
      catch (UnknownHostException e) {e.printStackTrace();}
      catch (IOException e) {e.printStackTrace();}
}
```

```
 /**
 * Receives a datagram packet containing the response into
 * a PMIPClientMessage object.
 * @return true if message is received, false if timeout occurs.
 * @param outMessage PMIPClientMessage object to receive new message into
 */

public synchronized boolean receive(PMIPClientMessage outMessage)  {
 try {
     DatagramPacket incoming = new DatagramPacket(new byte[PACKET_SIZE],
                                                                  PACKET_SIZE);
     receive(incoming);
     outMessage.internalize(incoming.getData());
 } catch (java.io.IOException e) {return false;}
 return true;
}

public synchronized boolean receivei(PMIPClientMessage outMessage)  {
 try {
     DatagramPacket incoming = new DatagramPacket(new byte[PACKET_SIZE],
                                                                  PACKET_SIZE);
     receive(incoming);
     outMessage.internalize(incoming.getData());
 } catch (java.io.IOException e) {return false;}
 return true;
}
```

```
public synchronized void sendi(PMIPClientMessage inMessage) {

 int index = 0; // find index of the interface that need to be open.
      //Obtain the list of network interfaces
      NetworkInterface[] devices = JpcapCaptor.getDeviceList();
      //for each network interface
      for (int i = 0; i < devices.length; i++) {
           // take their address
           for (NetworkInterfaceAddress a : devices[i].addresses)
                if (a.address.toString().compareTo("/"+PMIPReadConfig.getInetAdress()) == 0)
                     index = i;
      }

      try {
           JpcapSender sender = JpcapSender.openDevice(devices[index]);

           // create a UDP packet with specified port numbers
           UDPPacket packetUDP = new UDPPacket(freePort, PMIPReadConfig.getUDPPort());

           //spécify IPV4 header parameters
           InetAddress dst = InetAddress.getByName(PMIPReadConfig.getFA_InetAdress());
           InetAddress src = InetAddress.getByName(PMIPReadConfig.getInetAdress());
           packetUDP.setIPv4Parameter(0, false, true, false, 0, false, true, false, 0, 0,
                                                          PMIPReadConfig.getTTL(), 17, src, dst);

           //set the data field of the packet
           packetUDP.data = inMessage.externalize();

           //create an Ethernet packet (frame)
           EthernetPacket ether = new EthernetPacket();

           //set frame type as IP
           ether.frametype = EthernetPacket.ETHERTYPE_IP;

           //set source and destination MAC addresses
           ether.src_mac = devices[index].mac_address;
```

```
                ether.dst_mac = PMIPReadConfig.getFA_MACAdress_In_Byte_Array();

                //set the datalink frame of the packet packetUDP as ether
                packetUDP.datalink = ether;

                //send the packet IP
                sender.sendPacket(packetUDP);

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
    }
}
```

```
package jPMIPClient;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * This class represents all the PMIPClient Message.
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class PMIPClientMessage {
    private byte Type;             // type of registration message
    private int  S;                // Simultaneous bindings
    private int  B;                // Broadcast datagrams
    private int  D;                // Decapsulation by mobile node set to 0 -> Must of Draft PMIP
    private int  M;                // Minimal encapsulation
    private int  G;                // GRE encapsulation
    private int  r;                // Sent as zero
    private int  T;                // Reverse Tunneling requested
    private int  x;                // Sent as zero;
    private byte Code = 0;         // Represente the byte containing all the 8 before bits
    private byte[] Lifetime = new byte[2];     // Time before registration is considered expired
    private byte[] MNHomeIPAddress = new byte[4];  // IP address of the mobile node.
    private byte[] HAIPAddress = new byte[4];     // IP address of the mobile node's home agent
    private byte[] COAddress = new byte[4];       // IP address for the end of the tunnel
    private byte[] Identification  = new byte[8];   //  64-bit number field used against reply attack
    private List<PMIPClientExtension> extensionList = new ArrayList<PMIPClientExtension>();

    /** Method who's calling immediately itslef and give the params 0
     */
    public synchronized byte[] externalize(){
      return externalize(0);
    }
```

```
/** Method Externalize who return a Byte array who's contain
 * all the formated information of the PMIP Client Object Message
 * @param spi  The SPI number
 */
public synchronized byte[] externalize(int spi) {
  ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
  DataOutputStream outStream = new DataOutputStream (outBStream);
  try {
      outStream.writeByte(Type);
      outStream.writeByte(Code & ((byte)223)); // Mask is to set the D value to 0
      outStream.write(Lifetime, 0, 2);
      outStream.write(MNHomeIPAddress, 0, 4);
      outStream.write(HAIPAddress, 0, 4);
      //Check if the CoAddress is set to 0 for externalize the received message wich
      // not contain this adresse when de desencapsulation is on the FA
      if (COAddress[0] != 0 & COAddress[1] != 0 & COAddress[2] != 0 & COAddress[3] != 0){
      outStream.write(COAddress, 0, 4);}
      outStream.write(Identification, 0, 8);

      //Dump the list who's contain all the extansion
      for (int i = 0 ; i < extensionList.size() ; i++){
        //Conditon is up to creat the extension message.
        //if we give a SPI to the method that mean we want to creat a
        //Security extension
        if (spi != 0 & extensionList.get(i).type == 32){
              outStream.writeByte(32);
              outStream.writeByte(20);
              outStream.writeByte((spi>>24) & 0xff);
              outStream.writeByte((spi>>16) & 0xff);
              outStream.writeByte((spi>>8) & 0xff);
              outStream.writeByte((spi) & 0xff);
        }else{
              outStream.write(extensionList.get(i).getBytes(), 0,
                            extensionList.get(i).getBytes().length);
        }
      }
  } catch (IOException e) {System.err.println(e); }
```

```
    // extract the byte array from the Stream
    byte data[] = outBStream.toByteArray ();
    return data;
  }

  /** Convert a specified byte array containing a PMIP Client message
   * into a PMIPClientMessage object.
   * @return a PMIPClientMessage object with information from byte array.
   * @param  ibuff  byte array to convert to a PMIPClientMessage object
   */
  public synchronized PMIPClientMessage internalize(byte[] ibuff) {
    ByteArrayInputStream inBStream = new ByteArrayInputStream(ibuff, 0, ibuff.length );
    DataInputStream inStream = new DataInputStream (inBStream);

    try {
        Type = inStream.readByte();
        Code = inStream.readByte();
        inStream.readFully(Lifetime, 0, 2);
        inStream.readFully(MNHomeIPAddress, 0, 4);
        inStream.readFully(HAIPAddress, 0, 4);
        inStream.readFully(Identification, 0, 8);
        //Split bits from Byte Code with an int mask
        S = (Code & -128) >>7;
          B = (Code & 64) >>6;
          D = (Code & 32) >>5;
          M = (Code & 16) >>4;
          G = (Code & 8) >>3;
          r = (Code & 4) >>2;
          T = (Code & 2) >>1;
          x = (Code & 1);

          // Tourn until the end of the buffer to check all extension
          while (inStream.available() > 0) {
                //Read the first byte that represente the extension type
                int extensionType = inStream.readByte();
                // Check if the type is between 0 - 127 or 128 - 255
                // to know if he dont know the type respectively discard(0-127)
                // or ignor it (128-255)
                if (extensionType <=127){
```

```
                    switch(extensionType){
                        case 0:
                              // Do nothing because the type 0 exinsting in the RFC
                          break;

                        case 32:
                              extensionList.add(new ExtensionTypeSecurityAuthentication(extensionType,inStream));
                        break;

                        default:

                        break;
                      } // end Switch 0 to 127
                    }else{
                          switch(extensionType)
                      {
                        default:
                              new ExtensionTypeLengthValue(extensionType,inStream);
                              System.out.println("message ignored");
                        break;
                      } // end Switch 128 to 255
                    } //end Condition
            }
      } catch (IOException e) {
            System.err.println(e);
        }  // end catch
      return this;
    }

    /** Return a boolean after compared the Identificator in parameter
       *  with his own Identificator
       *  @param IIdentificationToCompare
       */
      public boolean IDcompareTo(byte[] IIdentificationToCompare){
            boolean condition = true;
            for(int i = 0 ; i < Identification.length-1 ; i++){
                  if (Identification[i] != IIdentificationToCompare[i]) condition=false;
            }
            return condition;
      }
```

```
/*****************************************************************/
/* add* methods for changing PMIPClientMessage datamembers. */
/*****************************************************************/

/** add a new extension in the extension list
 * @param extension  of PMIPClient message
 */
public void addExtensions(PMIPClientExtension extension) {
  extensionList.add(extension);
}


/*****************************************************************/
/* set* methods for changing PMIPClientMessage datamembers. */
/*****************************************************************/

/** type of registration message
 * @param Type  of registration message
 */
public void setType(byte Type) {
  this.Type = Type;
}

/** Set Simultaneous bindings
 * @param S  Simultaneous bindings
 */
public void setS(byte S) {
  this.S = S;
  Code = (byte)(Code | S<<7);
}

/** Set Broadcast datagrams
 * @param B  Broadcast datagrams
 */
public void setB(byte B) {
  this.B = B;
  Code = (byte)(Code | B<<6);
}
```

```
/** Set Minimal encapsulation
 * @param M  Minimal encapsulation
 */
public void setM(byte M) {
  this.M = M;
  Code = (byte)(Code | M<<4);
}

/** Set GRE encapsulation
 * @param G  GRE encapsulation
 */
public void setG(byte G) {
  this.G = G;
  Code = (byte)(Code | G<<3);
}

/** Set Sent as zero
 * @param r  Sent as zero
 */
public void setr(byte r) {
  this.r = r;
  Code = (byte)(Code | r<<2);
}

/** Set Reverse Tunneling requested
 * @param T  Reverse Tunneling requested
 */
public void setT(byte T) {
        this.T = T;
        Code = (byte)(Code | T<<1);
}

/** Set Sent as zero
 * @param x  Sent as zero
 */
public void setx(byte x) {
  this.x = x;
  Code = (byte)(Code | x);
}
```

```
/** Set Time before registration is considered expired
 * @param Lifetime  Time before registration is considered expired
 */
public void setLifetime(byte[] Lifetime) {
  this.Lifetime = Lifetime;
}

/** Set IP address of the mobile node.
 * @param MNHomeIPAddress  IP address of the mobile node.
 */
public void setHomeAddress(byte[] MNHomeIPAddress) {
  this.MNHomeIPAddress= MNHomeIPAddress;
}

/** Set IP address of the mobile node's home agent
 * @param HAIPAddress  IP address of the mobile node's home agent
 */
public void setHomeAgent(byte[] HAIPAddress) {
        this.HAIPAddress = HAIPAddress;
}

/** Set IP address for the end of the tunnel
 * @param COAddress  IP address for the end of the tunnel
 */
public void setCOAddress(byte[] COAddress) {
        this.COAddress = COAddress;
}

/** Set 64-bit number field used against reply attack
 * @param Identification  64-bit number field used against reply attack
 */
public void setIdentification(byte[] Identification) {
  this.Identification = Identification;
}

/** Set the High 4 byte of the Identification field
 * @param High4Byte  long number to convert in 4 byte
 */
public void setHigh4ByteIdentification(long High4Byte) {
  this.Identification [0]= (byte)((High4Byte>>24) & 0xff);
```

```
    this.Identification [1]= (byte)((High4Byte>>16) & 0xff);
    this.Identification [2]= (byte)((High4Byte>>8) & 0xff);
    this.Identification [3]= (byte)((High4Byte) & 0xff);
  }

  /** Set the High 4 byte of the Identification field
   * @param Low4Byte  long number to convert in 4 byte
   */
  public void setLow4byteIdentification(long Low4Byte) {
    this.Identification [4]= (byte)((Low4Byte>>24) & 0xff);
    this.Identification [5]= (byte)((Low4Byte>>16) & 0xff);
    this.Identification [6]= (byte)((Low4Byte>>8) & 0xff);
    this.Identification [7]= (byte)((Low4Byte) & 0xff);
  }

  /** Set Extension message
   * @param extension   of the message
   * @param pos   position in the list
   */
  public void  setExtension (int pos, PMIPClientExtension extension) {
    extensionList.set(pos, extension);
  }


  /*************************************************************
   * get* accesser fuctions return value of private data members*
   *************************************************************/

  /** Get Type of registration message */
  public byte getType() {
    return Type;
  }
  /** Get S Simultaneous bindings */
  public int getS() {
    return S;
  }
  /** Get B Broadcast datagrams */
  public int getB() {
    return B;
  }
```

```
/** Get D Broadcast datagrams */
public int getD() {
  return D;
}
/** Get M Minimal encapsulation */
public int getM() {
  return M;
}
/** Get G GRE encapsulation */
public int getG() {
  return G;
}
/** Get r Sent as zero */
public int getr() {
  return r;
}
/** Get T Reverse Tunneling requested */
public int getT() {
  return T;
}
/** Get x Sent as zero */
public int getx() {
  return x;
}

/** Get x Sent as zero */
public byte getCode() {
  return Code;
}

/** Get Lifetime Time before registration is considered expired */
public int getLifeTime() {
        return ((int) ((Lifetime[0]<<8 & 0xFFFFL)+(Lifetime[1]& 0xFFL)));

}

/** Get HomeAddress IP address of the mobile node. */
public byte[] getHomeAddress() {
  return MNHomeIPAddress;
}
```

```
/** Get HomeAgent IP address of the mobile node's home agent. */
public byte[] getHomeAgent() {
  return HAIPAddress;
}
/** Get COAddress IP address for the end of the tunnel */
public byte[] getCOAddress() {
  return COAddress;
}
/** Get Identification 64-bit number field used against reply attack */
public byte[] getIdentification() {
  return Identification;
}

/** Get High32bits in long unsigned format for the field identification*/
public long getHigh32BitsIdentification() {
  return ((long) ((Identification[0]<<24 & 0xFFFFFFFFL) +
                  (Identification[1]<<16 & 0xFFFFFFL)   +
                  (Identification[2]<<8 & 0xFFFFL)      +
                  (Identification[3]& 0xFFL)));
}

/** Get Low32bits in long unsigned format for the field identification*/
public long getLow32BitsIdentification() {
  return ((long) ((Identification[4]<<24 & 0xFFFFFFFFL) +
                (Identification[5]<<16 & 0xFFFFFFL)           +
                (Identification[6]<<8 & 0xFFFFL)              +
                (Identification[7]& 0xFFL)));
}

/** Get extension object from the list
 * @return a PMIPClientMessageExtension one extension of the message
 * selected in the extension list
 */
public PMIPClientExtension getExtension(int pos) {
  return extensionList.get(pos);
}

/** Get the extension length
 * @return a int that represent the length of the extension list
 */
```

```
    public int getExtensionLength() {
      return extensionList.size();
    }


    public int getExtensionPos(int Type) {
      int pos;
        for ( pos = 0; pos < extensionList.size() ; pos++){
            if(extensionList.get(pos).type == Type){
                  break;
            }
        }
      return pos;
    }
}
```

```
package jPMIPClient;

/**
 * This is the main class for the Extension PMIP Client Message
 * He only contain the commun variable of all his children
 * @author Hercher Yann
 * @version 1  29/10/2008
 * @see java.net.MulticastSocket
 */
public abstract class PMIPClientExtension {
     protected byte type;
     protected byte[] length = new byte[2];
     public abstract byte[] getBytes();

     public byte getType(){
          return type;
     }
}
```

```
package jPMIPClient;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;


/**
 * This class represents the extension of type Length Value
 * extend PMIPClientMessageExtension who's the main class ExtensionFormat
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class ExtensionTypeLengthValue extends PMIPClientExtension {

     byte[] data;

     /** Constructor that receive a buffer and dispatch it between the different
      *  extension variable
      *  @param type  of extension
      *   @param inStream that contain the extension Length Value
     */
     public ExtensionTypeLengthValue(int type, DataInputStream inStream) throws IOException{
          super.type = (byte) (type & 0xff);
          super.length[0] = inStream.readByte();
          data = new byte[length[0]];
          inStream.readFully(data, 0, length[0]);
     }

     /** Constructor used to create the extension Long Format
      * @param type
      * @param subType
      * @param data
      */
```

```
public ExtensionTypeLengthValue(byte type, byte subType, byte[] data){
      super.type = type;
      super.length[0] = (byte)(data.length & 0xff);
      this.data = data;
}

/** Return a byte array that contain the formated Length Value extension
 */
public byte[] getBytes(){
      ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
      DataOutputStream outStream = new DataOutputStream (outBStream);
      try {
          outStream.writeByte(type);
          outStream.write(length, 0, 1);
          outStream.write(data, 0, data.length);
      } catch (IOException e) {System.err.println(e);}
      return outBStream.toByteArray ();
}

/*************************************************************/
/* set* methods for changing Length Value Extension datamembers.*/
/*************************************************************/

 /** Set data of Length Value extension
 * @param data
 */
 public void setData(byte[] data){
      this.data = data;
      super.length[0] = (byte)(data.length & 0xff);
 }

 /*************************************************************
 * get* accesser fuctions return value of private data members*
 *************************************************************/
```

```
	/** Get Type of Length Value extension.*/
	public byte getType(){
		return super.type;
	}

	/** Get length of Length Value extension.*/
	public int getLength(){
		return (int)super.length[0];
	}

	/** Get data of Length Value extension.*/
	public byte[] getData(){
		return data;
	}
}
```

```
package jPMIPClient;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/**
 * This class represents the extension of type Long Format
 * extend PMIPClientMessageExtension which is the main class ExtensionFormat
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class ExtensionTypeLongFormat extends PMIPClientExtension {
     byte subType;
     byte[] data;

     /** Constructor that receive a buffer and dispatch it between the different
      *  extension variable
      *  @param type  of extension
      *    @param inStream contain the extension Long Format
      *  @throws IOException
      */
     public ExtensionTypeLongFormat(int type, DataInputStream inStream) throws IOException{
          super.type = (byte) (type & 0xff);
          subType = inStream.readByte();
          super.length[0] = inStream.readByte();
          inStream.readFully(data, 0, length[0]);
     }
```

```
 /** Constructor used to create the extension Long Format
  * @param type
  * @param subType
  * @param data
  */
public ExtensionTypeLongFormat(byte type, byte subType, byte[] data){
      super.type = type;
      this.subType = subType;
      super.length[0] = (byte)(data.length & 0xff);
      super.length[1] = (byte)((data.length>>8) & 0xff);
      this.data = data;
}

/** Return a byte array that contain the formated Long Format Extension
 */
public byte[] getBytes(){
      ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
      DataOutputStream outStream = new DataOutputStream (outBStream);
      try {
          outStream.writeByte(type);
          outStream.writeByte(subType);
          outStream.write(length, 0, 2);
          outStream.write(data, 0, data.length);
      } catch (IOException e) {System.err.println(e);}
      return outBStream.toByteArray ();
}
```

```
/*************************************************************/
/* set* methods for changing Long Format Extension datamembers.*/
/*************************************************************/

 /** Set subType of Long Format extension
 * @param subType
 */
 public void setSubType(byte subType){
      this.subType = subType;
 }

 /** Set data of Long Format extension
 * @param data
 */
 public void setData(byte[] data){
      this.data = data;
      super.length[0] = (byte)(data.length & 0xff);
      super.length[1] = (byte)((data.length>>8) & 0xff);
 }

 /*************************************************************
 * get* accesser fuctions return value of private data members*
 *************************************************************/

 /** Get subType of Long Format extension.*/
 public byte getSubType(){
      return subType;
 }

 /** Get Type of Long Format extension.*/
 public byte getType(){
      return super.type;
 }
```

```
	/** Get Length of Long Format extension.*/
	public int getLength(){
		return ( (int)super.length[0] + (int)(super.length[1]<<8) );
	}

	/** Get Data of Long Format extension.*/
	public byte[] getData(){
		return data;
	}
}
```

```
package jPMIPClient;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/**
 * This class represents the extension of type Mobile ID
 * extend PMIPClientMessageExtension who's the main class ExtensionFormat
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class ExtensionTypeMobileID extends PMIPClientExtension {
     byte subType;
     byte idType;
     byte[] identifier;

     /** Constructor that receive a buffer and dispatch it between the different
      *  extension variable
      *  @param type  of extension
      *   @param inStream that contain the extension Mobile ID
      *  @throws IOException
     */
     public ExtensionTypeMobileID(int type, DataInputStream inStream) throws IOException{
          super.type = (byte) (type & 0xff);
          super.length[0] = inStream.readByte();
          subType = inStream.readByte();
          idType = inStream.readByte();
          inStream.readFully(identifier, 0, length[0]);
     }
```

```
/** Constructor used to create the extension Mobile ID
 * @param type
 * @param subType
 * @param idType
 * @param identifier
 */
public ExtensionTypeMobileID(byte type, byte subType, byte idType, byte[] identifier){
      super.type = type;
      this.subType = subType;
      this.idType = idType;
      super.length[0] = (byte)(identifier.length & 0xff);
      this.identifier = identifier;
}

/** Return a byte array that contain the formated Mobile ID Extension
 */
public byte[] getBytes(){
      ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
      DataOutputStream outStream = new DataOutputStream (outBStream);
      try {
          outStream.writeByte(type);
          outStream.write(length, 0, 1);
          outStream.writeByte(subType);
          outStream.writeByte(idType);
          outStream.write(identifier, 0, identifier.length);
      } catch (IOException e) {System.err.println(e);}
      return outBStream.toByteArray ();
}
```

```
/*************************************************************/
/* set* methods for changing Mobile ID Extension datamembers.*/
/*************************************************************/

 /** Set subType of Mobile ID extension
 * @param subType
 */
 public void setSubType(byte subType){
      this.subType = subType;
 }

 /** Set idType of Mobile ID extension
 * @param idType
 */
 public void setIDType(byte idType){
      this.idType = idType;
 }

 /** Set identifier of Mobile ID extension
 * @param identifier
 */
 public void setIdentifier(byte[] identifier){
      this.identifier = identifier;
      super.length[0] = (byte)((identifier.length & 0xff) + 2);
 }

 /*************************************************************
 * get* accesser fuctions return value of private data members*
 *************************************************************/

 /** Get idType of Mobile ID extension.*/
 public byte getIDType(){
      return idType;
 }
```

```
	/** Get subType of Mobile ID extension.*/
	public byte getSubType(){
		return subType;
	}

	/** Get Type of Mobile ID extension.*/
	public byte getType(){
		return super.type;
	}

	/** Get Length of Mobile ID extension.*/
	public int getLength(){
		return (int)super.length[0];
	}

	/** Get identifier of Mobile ID extension.*/
	public byte[] getIdentifier(){
		return identifier;
	}
}
```

```
package jPMIPClient;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;


/**
 * This class represents the extension of type Security Authentication
 * extend PMIPClientMessageExtension who's the main class ExtensionFormat
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class ExtensionTypeSecurityAuthentication extends PMIPClientExtension {
     byte[] spi = new byte[4];
     byte[] authenticator = null;
     byte[] codeHash;
     String SecurityKey;

     /** Empty Constructor used for checing the security binding between the incoming message
     * field authenticator and the the authenticator buld with the own key et SPI
     */
     public ExtensionTypeSecurityAuthentication(){}
```

```
/** Constructor that receive a buffer and dispatch it between the different
 *  extension variable
 *  @param type  of extension
 *   @param inStream that contain the extension Security Authentication
 *  @throws IOException
*/
public ExtensionTypeSecurityAuthentication(int type, DataInputStream inStream) throws IOException{
     super.type = (byte)(type & 0xff);
     super.length[0] = inStream.readByte();
     inStream.readFully(spi, 0, 4);
     authenticator = new byte[length[0]-spi.length];
     inStream.readFully(authenticator, 0, length[0]-spi.length);
}

/** Constructor used to create the extension Security Authentication
 * @param SecurityKey
 * @param type
 * @param spi
 */
public ExtensionTypeSecurityAuthentication(String SecurityKey , int type,int spi){
     this.SecurityKey = SecurityKey;
     super.type = (byte)(type & 0xff);
     this.spi[3]  = (byte)(spi&0xff);
     this.spi[2]  = (byte)(spi>>8&0xff);
     this.spi[1]  = (byte)(spi>>16&0xff);
     this.spi[0]  = (byte)(spi>>24&0xff);
     super.length[0] = (byte)(20);  //4byte SPI + 16Byte authenticator 128bit md5
     this.authenticator = null;
}
```

```
/** Return a boolean after compared the authenticator in parameter
 *  with his own authenticator
 *  @param authenticatorToCompare
 */
public boolean compareTo(byte[] authenticatorToCompare){
     boolean condition = true;
     for(int i = 0 ; i < authenticator.length-1 ; i++){
          if (authenticator[i] != authenticatorToCompare[i]) condition=false;
     }
     return condition;
}

/** Return a byte array that contain the formated Security Authentication Extension
 */
public byte[] getBytes(){
     ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
     DataOutputStream outStream = new DataOutputStream (outBStream);
     try {
         outStream.writeByte(type);
         outStream.write(length, 0, 1);
         outStream.write(spi, 0, 4);
         if (authenticator != null){
          outStream.write(authenticator, 0, authenticator.length);
         }
     } catch (IOException e) {System.err.println(e);}
     return outBStream.toByteArray ();
}
```

```
/** When called this method take the variable CodeHash that contain the byte array to
 *  Hash in HMAC-MD5 and place it to Authentication field
 */
private void hash(){

     try {
          byte[] passwordInBytes = new byte[16];

          int passwordlength = SecurityKey.length();
          //trunk the length to max 16 byte = 128 bits
          if (SecurityKey.length() > 16) passwordlength = 16;

          // Place each character of the key in a byte array
          // the first char of the key is placed in the lowest case in the array
          for(int i = 0; i < passwordlength; i++)
          {
               passwordInBytes[i] = (byte)SecurityKey.charAt(i);
          }

          SecretKey key = new SecretKeySpec(passwordInBytes, "HmacMD5") ;

        // Create a MAC object using HMAC-MD5 and initialize with key
          Mac mac = Mac.getInstance(key.getAlgorithm());

        mac.init(key);

      this.authenticator = mac.doFinal(codeHash);

    } catch (InvalidKeyException e) {
    } catch (NoSuchAlgorithmException e) {
    }
}
```

```
/*************************************************************************/
/* set* methods for changing Security Authentication Extension datamembers.*/
/*************************************************************************/

 /** Set SPI of Security Authentication extension
 * @param spi
 */
 public void setSPI(int spi){
      this.spi[3]  = (byte)(spi & 0xff);
      this.spi[2]  = (byte)(spi >> 8 & 0xff);
      this.spi[1]  = (byte)(spi >> 16 & 0xff);
      this.spi[0]  = (byte)(spi >> 24 & 0xff);
 }

 /** Set Authenticator of Security Authentication extension
 * @param securityKey Security key partaged between the PMIP Client and the HA
 * @param codeHash  byte array that need to be hash
 */
 public void setAuthenticator(String securityKey , byte[] codeHash){
      this.codeHash = codeHash;
      this.SecurityKey = securityKey;
      hash();
 }

 /*************************************************************
 * get* accesser fuctions return value of private data members*
 *************************************************************/

 /** Get SPI of Security Authentication extension.*/
 public int getSPI(){
      return ( (int)spi[0] + (int)(spi[1]<<8) + (int)(spi[2]<<16) + (int)(spi[3]<<24) );
 }
```

```
	/** Get type of Security Authentication extension.*/
	public byte getType(){
		return super.type;
	}

	/** Get length of Security Authentication extension.*/
	public int getLength(){
		return ((int)super.length[0]);
	}

	/** Get authenticator of Security Authentication extension.*/
	public byte[] getAuthenticator(){
		return authenticator;
	}
}
```

```
package jPMIPClient;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;


/**
 * This class represents the extension of type Short Format
 * extend PMIPClientMessageExtension who's the main class ExtensionFormat
 * @author Yann Hercher
 * @version 1  29/10/2008
 */
public class ExtensionTypeShortFormat extends PMIPClientExtension {
     byte subType;
     byte[] data;

     /** Constructor that receive a buffer and dispatch it between the different
      *  extension variable
      *  @param type  of extension
      *   @param inStream that contain the extension Short Format
      *  @throws IOException
     */
     public ExtensionTypeShortFormat(int type, DataInputStream inStream) throws IOException{
          super.type = (byte) (type & 0xff);
          super.length[0] = inStream.readByte();
          subType = inStream.readByte();
          inStream.readFully(data, 0, length[0]);
     }
```

```
/** Constructor used to create the extension Short Format
 * @param type
 * @param subType
 * @param data
 */
public ExtensionTypeShortFormat(byte type, byte subType, byte[] data){
      super.type = type;
      this.subType = subType;
      super.length[0] = (byte)((data.length & 0xff) + 1);
      this.data = data;
}

/** Return a byte array that contain the formated Short Format Extension
 */
public byte[] getBytes(){
      ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
      DataOutputStream outStream = new DataOutputStream (outBStream);
      try {
          outStream.writeByte(type);
          outStream.write(length, 0, 1);
          outStream.writeByte(subType);
          outStream.write(data, 0, data.length);
      } catch (IOException e) {System.err.println(e);}
      return outBStream.toByteArray ();
}
```

```
/*************************************************************/
/* set* methods for changing Short Format Extension datamembers.*/
/*************************************************************/

 /** Set subType of Short Format extension
 * @param subType
 */
 public void setSubType(byte subType){
      this.subType = subType;
 }

 /** Set Data of Short Format extension
 * @param data
 */
 public void setData(byte[] data){
      this.data = data;
      super.length[0] = (byte)((data.length & 0xff) + 1);
 }

 /*************************************************************
* get* accesser fuctions return value of private data members*
*************************************************************/

 /** Get subType of Short Format extension.*/
 public byte getSubType(){
      return subType;
 }

 /** Get Type of Short Format extension.*/
 public byte getType(){
      return super.type;
 }

 /** Get length of Short Format extension.*/
 public int getLength(){
      return (int)super.length[0];
```

```
	}

	/** Get date of Short Format extension.*/
	public byte[] getData(){
		return data;
	}
}
```

```
package jPMIPClient;

import java.io.*;
import java.util.Hashtable;
import java.util.StringTokenizer;


/**
 * This class extract information variable from a .txt file for the configuration of
 * the PMIP Client. The other class can get the information by the get methode
 *
 * To add a new information, you need to add it into the configfile.txt and add it into the
 * the Hashtable valueOfPMIPClient with his default value.
 * Don't forget to create a methode get to return the value
 *
 * @author Hercher Yann
 * @version 1  14/11/2008
 */
public class PMIPReadConfig {
     static Hashtable<String, String> valueOfPMIPClient = new Hashtable<String, String>();

     /**
     * method that open the configuration file of the PMIP Client
     * and palce the information in a Hashtable
     */
     public void ReadPMIPConfig(){
          String variableName, value;
          String file ="PMIPConfig.txt";

          try{
               InputStream ips =
                    new FileInputStream(System.getProperties().get("user.dir")+"/config/"+file);
               InputStreamReader ipsr=new InputStreamReader(ips);
               BufferedReader br=new BufferedReader(ipsr);
```

```
        // Vairable that take every line of the config file
        String line;

      // Set the default option of the variable in the Hashtable
      valueOfPMIPClient.put("MNDefaultTunnelLifetime", new String("300"));
      valueOfPMIPClient.put("UDPPort", new String("434"));
      valueOfPMIPClient.put("TTL", new String("255"));
      valueOfPMIPClient.put("InetAdress", new String("127.0.0.1"));
      valueOfPMIPClient.put("FA_InetAddress", new String("127.0.0.1"));
      valueOfPMIPClient.put("FA_MACAddress", new String("00:00:00:00:00:00"));
      valueOfPMIPClient.put("COAddress", new String("0.0.0.0"));
      valueOfPMIPClient.put("TimeBeforeKillThread", new String("300"));


      StringTokenizer st;


// Turn still the end of the documents (the buffer in facts)
        while ((line=br.readLine())!=null){

              // Detect if a line is empty to jump an exception
              if (line.isEmpty() != true){

                    // Discard the # comentary ligne
                    if (line.charAt(0) != '#'){

                          // Separate the name and the variable
                          st = new StringTokenizer(line," ");
                          variableName = st.nextToken();

                          // Keep the variable only if name exist in the HashTable
                          if (valueOfPMIPClient.get(variableName) != null){
                                    value = st.nextToken();
                                    // Delete to replace the variable in the hashTable
                                    valueOfPMIPClient.remove(variableName);
                                    valueOfPMIPClient.put(variableName, value);
```

```
                        }
                    }
                }
            }
            br.close();
        }
        catch (Exception e){
            System.out.println(e.toString());
        }
    }


    /************************************************************
    * get* accesser fuctions return value of private data members*
    ************************************************************/

    /** Get the MNDEfaultTunnelLifeTime in int*/
    public static int getMNDefaultTunnelLifetime(){
        return Integer.parseInt((String)valueOfPMIPClient.get("MNDefaultTunnelLifetime"));
    }

    /** Get the UDPPort in int*/
    public static int getUDPPort(){
        return Integer.parseInt((String)valueOfPMIPClient.get("UDPPort"));
    }

    /** Get the TTL in int*/
    public static int getTTL(){
        return Integer.parseInt((String)valueOfPMIPClient.get("TTL"));
    }

    /** Get the InetAdress in String*/
    public static String getInetAdress(){
        return (String)valueOfPMIPClient.get("InetAdress");
    }
```

```
	/** Get the FA_InetAddress in String*/
	public static String getFA_InetAdress(){
		return (String)valueOfPMIPClient.get("FA_InetAddress");
	}

	/** Get the FA_MACAddress in String*/
	public static String getFA_MACAdress(){
		return (String)valueOfPMIPClient.get("FA_MACAddress");
	}

	/** Get the getFA_MACAdress_In_Byte_Array in byte array*/
	public static byte[] getFA_MACAdress_In_Byte_Array(){
		StringTokenizer st = new StringTokenizer(
				(String)valueOfPMIPClient.get("FA_MACAddress"),":");
		return new byte[]{(byte)(Integer.parseInt(st.nextToken(),16)),
					(byte)(Integer.parseInt(st.nextToken(),16)),
					(byte)(Integer.parseInt(st.nextToken(),16)),
					(byte)(Integer.parseInt(st.nextToken(),16)),
					(byte)(Integer.parseInt(st.nextToken(),16)),
					(byte)(Integer.parseInt(st.nextToken(),16))};
	}

	/** Get the COAddress in String*/
	public static String getCOAddress(){
		return (String)valueOfPMIPClient.get("COAddress");
	}

	/** Get the COAddress in String*/
	public static String getTimeBeforeKillThread(){
		return (String)valueOfPMIPClient.get("TimeBeforeKillThread");
	}
}
```

```
package jPMIPClient;

import java.io.*;


/**
 * This class has been created to fixe a bug of the Java methode MulticastSocket
 * In fact if we set some information in the socket they are not told in consideration.
 * So the setTimeToLive is'not changed in the message sent.
 *
 * to fix the bug i load some bash file that contain the commande to change te default TTL
 * of the current OS.
 *
 * @author Hercher Yann
 * @version 1  14/11/2008
 */
public class FixeSetTTL {
     private static String cmdNewTTL, cmdDefaultTTL;

     /**
     * Method that check the OS and set the default TTL
     */
     public void FixeSetTTL_Start() {
          String osName;

          //Determinate wich OS is running
          osName = System.getProperties().getProperty("os.name");

          //Choose the correct OS config Files
          if (osName.compareTo("Linux") == 0){

               cmdNewTTL = "/bugFixeTTL/setNewTTL.sh";
               cmdDefaultTTL = "/bugFixeTTL/setDefaultTTL.sh";
```

```
            // Create the new ttl file with the ttl's PMIPReadConfig
            File newFile = new File("."+cmdNewTTL);
            FileWriter writer = null;

            String texte = "echo '"+PMIPReadConfig.getTTL()+
                              "' > /proc/sys/net/ipv4/ip_default_ttl";

            try {
                  newFile.createNewFile();
                  writer = new FileWriter("."+cmdNewTTL, false);
                  writer.write(texte,0,texte.length());
                  writer.close();
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

      }
      // OS unknown
      else {
            // Do nothing
            cmdNewTTL = "";
      }
}
```

```
    /*************************************************************/
    /* set* methods for changing Long Format Extension datamembers.*/
    /*************************************************************/

     /** Set the new OS TTL by default
     */
     public static void setNewTTL(){
          try {
               Runtime r = Runtime.getRuntime();
               Process p = r.exec("sh "+System.getProperties().get("user.dir")+cmdNewTTL);
               p.waitFor();//si l'application doit attendre a ce que ce process fini
          }catch(Exception e) {
               System.out.println("erreur d'execution " + cmdNewTTL + e.toString());
          }
     }

     /** Set the old default OS TTL by default
     */
     public static void setDefaultTTL(){
          try {
               Runtime r = Runtime.getRuntime();
               Process p = r.exec("sh "+System.getProperties().get("user.dir")+cmdDefaultTTL);
               p.waitFor();//si l'application doit attendre a ce que ce process fini
          }catch(Exception e) {
               System.out.println("erreur d'execution " + cmdDefaultTTL + e.toString());
          }
     }
}
```

```
package jDHCP;

import java.io.IOException;
import java.net.*;

/**
 * This class represents a Socket for sending DHCP Messages
 * @author Jason Goldschmidt
 * @version 1.1.1  9/06/1999
 * @see java.net.DatagramSocket
 */
public class DHCPSocket extends DatagramSocket  {

    static protected int PACKET_SIZE = 1500; // default MTU for ethernet

    /**
     * Constructor for creating DHCPSocket on a specific port on the local
     * machine.
     * @param inPort the port for the application to bind.
     * @throws IOException
     */
    public DHCPSocket (int inPort) throws IOException{
           super(inPort, InetAddress.getByName("0.0.0.0"));
    }
```

```
     /**
     * Sends a DHCPMessage object to a predifined host.
     * @param inMessage well-formed DHCPMessage to be sent to a server
     */
     public void send(DHCPMessage inMessage) throws java.io.IOException {
           byte data[] = new byte[PACKET_SIZE];
           data = inMessage.externalize();
           InetAddress dest = null;
           try {
                dest = InetAddress.getByName(inMessage.getDestinationAddress());
           } catch (UnknownHostException e) {}

           DatagramPacket outgoing = new DatagramPacket(data, data.length,
                                                  dest, inMessage.getPort());

           send(outgoing);
     }

     /**
      * Receives a datagram packet containing a DHCP Message into
      * a DHCPMessage object.
      * @return true if message is received, false if timeout occurs.
      * @param outMessage DHCPMessage object to receive new message into
      */

     public void receive(DHCPMessage outMessage)  {
      try {
           DatagramPacket incoming = new DatagramPacket(new byte[PACKET_SIZE],
                                                                  PACKET_SIZE);
           receive(incoming); // block on receive for SO_TIMEOUT
           outMessage.internalize(incoming.getData());

      } catch (java.io.IOException e) {
         }  // end catch
     }
}
```

```
package jDHCP;

import java.util.*;

/**
 * This class represents a linked list of options for a DHCP message.
 * Its purpose is to ease option handling such as add, remove, or change.
 * @author Jason Goldschmidt
 * @version 1.1.1 9/06/1999
 */
public class DHCPOptions {

    /**
     *This inner class represent an entry in the Option Table
     */

    class DHCPOptionsEntry extends Object {
        protected byte code;
        protected byte length;
        protected byte[] content;


        public DHCPOptionsEntry(byte entryCode, byte entryLength,
                    byte[] entryContent) {
          code = entryCode;
          length = entryLength;
          content = entryContent;
        }
    }

    private Hashtable<Byte, DHCPOptionsEntry> optionsTable = null;

    public DHCPOptions () {
        optionsTable = new Hashtable<Byte, DHCPOptionsEntry>();
    }
```

```
/**
 * Removes option with specified bytecode
 * @param entryCode The code of option to be removed
 */

public void removeOption(byte entryCode) {
 optionsTable.remove(new Byte(entryCode));
}


/*
 * Returns true if option code is set in list; false otherwise
 * @param entryCode The node's option code

 */

//* @return true if option is set, otherwise false

public boolean contains(byte entryCode) {
 return optionsTable.containsKey(new Byte(entryCode));
}

/**
 * Determines if list is empty
 * @return true if there are no options set, otherwise false
 */
public boolean isEmpty() {
 return optionsTable.isEmpty();
}

/**
 * Fetches value of option by its option code
 * @param entryCode The node's option code

 */
```

```
    //* @return byte array containing the value of option entryCode.
    //*         null is returned if option is not set.

    public byte[] getOption(byte entryCode) {
     if (this.contains(entryCode)) {
         DHCPOptionsEntry ent =
           (DHCPOptionsEntry) optionsTable.get(new Byte(entryCode));
         return ent.content;
     } else {
         return null;
     }
    }

    /**
     * Changes an existing option to new value
     * @param entryCode The node's option code
     * @param value Content of node option
     */
    public void setOption (byte entryCode, byte[] value) {
     DHCPOptionsEntry opt =
         new DHCPOptionsEntry(entryCode, (byte) value.length, value);
          optionsTable.put(new Byte(entryCode), opt);
    }

    /**
     * Returns the option value of a specified option code in a byte array
     * @param length Length of option content
     * @param position Location in array of option node
     * @param options The byte array of options

     */
//   * @return byte array containing the value for the option
    private byte[] getArrayOption(int length, int position, byte[] options) {
     byte[] value = new byte[(int) length];
     for (int i = 0; i < (int) length; i++)
```

```
        value[i] = options[position + i];
     return value;
    }


    /**
     * Converts an options byte array to a linked list
     * @param optionsArray The byte array representation of the options list
     */
    public void internalize(byte[] optionsArray) {

     /* Assume options valid and correct */
     int pos = 4;          // ignore vendor magic cookie
     byte code, length;
     byte[] value;

     while (optionsArray[pos] != (byte) 255) { // until end option
         code = optionsArray[pos++];
         length = optionsArray[pos++];
         value = getArrayOption(length, pos, optionsArray);
         setOption(code, value);
         pos += length;   // increment position pointer
     }
    }


    /**
     * Converts a linked options list to a byte array

     */

    //* @return array representation of optionsTable
    // todo provide overflow return
    public byte[] externalize() {
        byte[] options = new byte[312];
```

```
        options[0] = (byte) 99;    // insert vendor magic cookie
        options[1] = (byte) 130;
        options[2] = (byte) 83;
        options[3] = (byte) 99;

           int position = 4;

           for(Enumeration<DHCPOptionsEntry> e = optionsTable.elements(); e.hasMoreElements(); ) {
               DHCPOptionsEntry entry = (DHCPOptionsEntry) e.nextElement();
               options[position++] = entry.code;
               options[position++] = entry.length;
               for(int i = 0; i < entry.length; ++i) {
                   options[position++] = entry.content[i];
               }
           }
           options[position] = (byte) 255;  // insert end option
           return options;
       }


       /**
        *     Prints the options linked list: For testing only.
        */
       public void printList() {
        System.out.println(optionsTable.toString());
       }
}
```

```
package jDHCP;

import java.net.*;
import java.io.*;


/**
 * This class represents a DHCP Message.
 * @author Jason Goldschmidt and Nick Stone
 * @version 1.1.1 9/06/1999
 */

public class DHCPMessage extends Object {
    private byte op;                 // Op code
    private byte htype;                   // HW address Type
    private byte hlen;                    // hardware address length
    private byte hops;                    // Hw options
    private int  xid;                // transaction id
    private short secs;              // elapsed time from trying to boot
    private short flags;             // flags
    private byte ciaddr[] = new byte[4];  // client IP
    private byte yiaddr[] = new byte[4];  // your client IP
    private byte siaddr[] = new byte[4];  // Server IP
    private byte giaddr[] = new byte[4];  // relay agent IP
    private byte chaddr[] = new byte[16];       // Client HW address
    private byte sname[] = new byte[64];  // Optional server host name
    private byte file []= new byte[128];       // Boot file name
    private DHCPOptions optionsList = null; // internal representaton of
    // DHCP Options

    private int gPort;          // global port variable for object
    private InetAddress destination_IP;         // IP format of the servername
```

```
    /**
     * Default DHCP client port
     */
    public static final int CLIENT_PORT = 68; // client port (by default)

    /**
     * Default DHCP server port
     */
    public static final int SERVER_PORT = 67; // server port (by default)

    public static InetAddress BROADCAST_ADDR = null;

    // DHCP Message Types

    /**
     * Code for DHCPDISCOVER Message
     */
    public static final int DISCOVER = 1;

    /**
     * Code for DHCPOFFER Message
     */
    public static final int OFFER = 2;

    /**
     * Code for DHCPREQUEST Message
     */
    public static final int REQUEST = 3;

    /**
     * Code for DHCPDECLINE Message
     */
    public static final int DECLINE = 4;
```

```
/**
 * Code for DHCPACK Message
 */
public static final int ACK = 5;

/**
 * Code for DHCPNAK Message
 */
public static final int NAK = 6;

/**
 * Code for DHCPRELEASE Message
 */
public static final int RELEASE = 7;

/**
 * Code for DHCPINFORM Message
 */
public static final int INFORM = 8;

static {
 if (BROADCAST_ADDR == null) {
     try {
      BROADCAST_ADDR =
          InetAddress.getByName("255.255.255.255");
      // broadcast address(by default)
     } catch (UnknownHostException e) {}
 }
}

/** Creates empty DHCPMessage object,
 * initializes the object, sets the host to the broadcast address,
 * the local subnet, binds to the default server port. */

public DHCPMessage () {
 Initialize();
```

```
 destination_IP = BROADCAST_ADDR;
 gPort = SERVER_PORT;

}

/** Copy constructor
 * creates DHCPMessage from inMessage
 */
// This needs to be tested.
public DHCPMessage (DHCPMessage inMessage) {
 Initialize();
 destination_IP = BROADCAST_ADDR;
 gPort = SERVER_PORT;
 op = inMessage.getOp();
 htype = inMessage.getHtype();
 hlen = inMessage.getHlen();
 hops = inMessage.getHops();
 xid = inMessage.getXid();
 secs = inMessage.getSecs();
 flags = inMessage.getFlags();
 ciaddr = inMessage.getCiaddr();
 yiaddr = inMessage.getYiaddr();
 siaddr = inMessage.getSiaddr();
 giaddr = inMessage.getGiaddr();
 chaddr = inMessage.getChaddr();
 sname = inMessage.getSname();
 file = inMessage.getFile();
 optionsList.internalize(inMessage.getOptions());

}
/** Copy constructor
 * creates DHCPMessage from inMessage and sets server and port
 */

public DHCPMessage (DHCPMessage inMessage,
```

```
                InetAddress inServername,
                int inPort) {
    Initialize();

    this.destination_IP = inServername;
    this.gPort = inPort;

    op = inMessage.getOp();
    htype = inMessage.getHtype();
    hlen = inMessage.getHlen();
    hops = inMessage.getHops();
    xid = inMessage.getXid();
    secs = inMessage.getSecs();
    flags = inMessage.getFlags();
    ciaddr = inMessage.getCiaddr();
    yiaddr = inMessage.getYiaddr();
    siaddr = inMessage.getSiaddr();
    giaddr = inMessage.getGiaddr();
    chaddr = inMessage.getChaddr();
    sname = inMessage.getSname();
    file = inMessage.getFile();
    optionsList.internalize(inMessage.getOptions());
  }

  public DHCPMessage (DHCPMessage inMessage, InetAddress inServername) {
    Initialize();

    this.destination_IP = inServername;
    this.gPort = SERVER_PORT;

    op = inMessage.getOp();
    htype = inMessage.getHtype();
    hlen = inMessage.getHlen();
    hops = inMessage.getHops();
    xid = inMessage.getXid();
    secs = inMessage.getSecs();
```

```
        flags = inMessage.getFlags();
        ciaddr = inMessage.getCiaddr();
        yiaddr = inMessage.getYiaddr();
        siaddr = inMessage.getSiaddr();
        giaddr = inMessage.getGiaddr();
        chaddr = inMessage.getChaddr();
        sname = inMessage.getSname();
        file = inMessage.getFile();
        optionsList.internalize(inMessage.getOptions());
    }


    /** Creates empty DHCPMessage object,
     * initializes the object, sets the host to a specified host name,
     * and binds to a specified port.
     * @param inServername  the host name
     * @param inPort  the port number
     */

    public DHCPMessage (InetAddress inServername, int inPort) {
        Initialize();

        destination_IP = inServername;
        gPort = inPort;
    }

    /** Creates empty DHCPMessage object,
     * initializes the object, sets the host to a specified host name,
     * and binds to the default port.
     * @param inServername  the host name
     */

    public DHCPMessage (InetAddress inServername) {
        Initialize();

        destination_IP = inServername;
```

```
    gPort = SERVER_PORT;
  }

  /** Creates empty DHCPMessage object,
   * initializes the object, sets the host to the broadcast address,
   * and binds to a specified port.
   * @param inPort  the port number
   */

  public DHCPMessage (int inPort) {
    Initialize();

    destination_IP = BROADCAST_ADDR;
    gPort = inPort;
  }

  /** Creates empty DHCPMessage object,
   * initializes the object with a specified byte array containing
   * DHCP message information, sets the host to default host name, the
   * local subnet, and bind to the default server port.
   * @param ibuff  the byte array to initialize DHCPMessage object
   */

  public DHCPMessage (byte[] ibuff ) {
    Initialize();
    internalize(ibuff);

    destination_IP = BROADCAST_ADDR;
    gPort = SERVER_PORT;

  }


  /** Creates empty DHCPMessage object,
   * initializes the object with a specified byte array containing
   * DHCP message information, sets the host to specified host name,
```

```
     * and binds to the specified port.
     * @param ibuff  the byte array to initialize DHCPMessage object
     * @param inServername  the hostname
     * @param inPort  the port number
     */

    public DHCPMessage (byte[] ibuff, InetAddress inServername, int inPort) {
     Initialize();
     internalize(ibuff);

     destination_IP = inServername;
     gPort = inPort;

    }

    /** Creates empty DHCPMessage object,
     * initializes the object with a specified byte array containing
     * DHCP message information, sets the host to broadcast address,
     * and binds to the specified port.
     * @param ibuff  the byte array to initialize DHCPMessage object
     * @param inPort  the port number
     */

    public DHCPMessage (byte[] ibuff, int inPort) {
     Initialize();
     internalize(ibuff);

     destination_IP = BROADCAST_ADDR;
     gPort = inPort;

    }

    /** Creates empty DHCPMessage object,
     * initializes the object with a specified byte array containing
     * DHCP message information, sets the host to specified host name,
     * and binds to the specified port.
```

```
     * @param ibuff  the byte array to initialize DHCPMessage object
     * @param inServername  the hostname
     */

    public DHCPMessage (byte[] ibuff, InetAddress inServername) {
     Initialize();
     internalize(ibuff);

     destination_IP = inServername;
     gPort = SERVER_PORT;

    }

    // ********add port/server options for all constructors************
    // plus add constructer that takes DHCPMessage object parameter and
    // sets IP and port from input param. can we say pain in my arse!

    public DHCPMessage (DataInputStream inStream) {
     Initialize();
     try {
         op = inStream.readByte();
         htype = inStream.readByte();
         hlen = inStream.readByte();
         hops = inStream.readByte();
         xid = inStream.readInt();
         secs = inStream.readShort();
         flags = inStream.readShort();
         inStream.readFully(ciaddr, 0, 4);
         inStream.readFully(yiaddr, 0, 4);
         inStream.readFully(siaddr, 0, 4);
         inStream.readFully(giaddr, 0, 4);
         inStream.readFully(chaddr, 0, 16);
         inStream.readFully(sname, 0, 64);
         inStream.readFully(file, 0, 128);
         byte[] options = new byte[312];
         inStream.readFully(options, 0, 312);
```

```
        optionsList.internalize(options);
    } catch (IOException e) {
            System.err.println(e);
        }  // end catch

}


/*
 * DHCPMessage::Initialize
 * initializes datamembers in the constructors
 * every empty DHCPMessage object will by default contain these params.
 * Initializes options array from linked list form
 */

private void Initialize () {
 optionsList = new DHCPOptions();
}


/** Converts a DHCPMessage object to a byte array.
 * @return a byte array with information from DHCPMessage object.
 */

// Purpose: convert a DHCPMessage object to a byte array.
// Precondition: a "well-formed" DHCPMessage object
// Postconditon: a byte array representation of that object is returned

public synchronized byte[] externalize() {
 ByteArrayOutputStream outBStream = new ByteArrayOutputStream ();
 DataOutputStream outStream = new DataOutputStream (outBStream);

 try {
     outStream.writeByte(op);
     outStream.writeByte(htype);
     outStream.writeByte(hlen);
```

```
        outStream.writeByte(hops);
        outStream.writeInt(xid);
        outStream.writeShort(secs);
        outStream.writeShort(flags);
        outStream.write(ciaddr, 0, 4);
        outStream.write(yiaddr, 0, 4);
        outStream.write(siaddr, 0, 4);
        outStream.write(giaddr, 0, 4);
        outStream.write(chaddr, 0, 16);
        outStream.write(sname, 0, 64);
        outStream.write(file, 0, 128);
        byte[] options = new byte[312];
        if (optionsList == null) {
         Initialize();
        }
        options = optionsList.externalize();
        outStream.write(options, 0, 312);
    } catch (IOException e) {
           System.err.println(e);
       }  // end catch

    // extract the byte array from the Stream
    byte data[] = outBStream.toByteArray ();

    return data;
  }

  /** Convert a specified byte array containing a DHCP message into a
   * DHCPMessage object.
   * @return a DHCPMessage object with information from byte array.
   * @param  ibuff  byte array to convert to a DHCPMessage object
   */

  // Precondition: a byte array containg a DHCPMessage object.
  // Postconditon: the contents on the byte array are stored into
  // the datamembers of the DHCPMessage object.
```

```
public synchronized DHCPMessage internalize(byte[] ibuff) {
 ByteArrayInputStream inBStream = new ByteArrayInputStream
     (ibuff, 0, ibuff.length );
 DataInputStream inStream = new DataInputStream (inBStream);

 try {
     op = inStream.readByte();
     htype = inStream.readByte();
     hlen = inStream.readByte();
     hops = inStream.readByte();
     xid = inStream.readInt();
     secs = inStream.readShort();
     flags = inStream.readShort();
     inStream.readFully(ciaddr, 0, 4);
     inStream.readFully(yiaddr, 0, 4);
     inStream.readFully(siaddr, 0, 4);
     inStream.readFully(giaddr, 0, 4);
     inStream.readFully(chaddr, 0, 16);
     inStream.readFully(sname, 0, 64);
     inStream.readFully(file, 0, 128);
     byte[]  options = new byte[312];
     inStream.readFully(options, 0, 312);
     if (optionsList == null) {
      Initialize();
     }
     optionsList.internalize(options);
 } catch (IOException e) {
        System.err.println(e);
    }  // end catch

 return this;
}

/*************************************************************/
/* set* methods for changing DHCPMessage datamembers.        */
```

```
/*************************************************************/

/** Set message Op code / message type.
 * @param inOp  message Op code / message type
 */
public void setOp(byte inOp) {
 op = inOp;
}

/** Set hardware address type.
 * @param inHtype hardware address type
 */
public void setHtype(byte inHtype) {
 htype = inHtype;
}

/** Set hardware address length.
 * @param inHlen  hardware address length
 */
public void setHlen(byte inHlen) {
 hlen = inHlen;
}

/** Set hops field.
 * @param inHops hops field
 */
public void  setHops(byte inHops) {
 hops = inHops;
}

/** Set transaction ID.
 * @param inXid  transactionID
 */
public void setXid(int inXid) {
 xid = inXid;
}
```

```
/** Set seconds elapsed since client began address acquisition or
 * renewal process.
 * @param inSecs seconds elapsed since client began address acquisition
 * or renewal process
 */
public void setSecs(short inSecs) {
 secs = inSecs;
}

/** Set flags field.
 * @param inFlags flags field
 */
public void  setFlags (short inFlags) {
 flags = inFlags;
}

/** Set client IP address.
 * @param inCiaddr client IP address
 */
public void  setCiaddr (byte [] inCiaddr) {
 ciaddr = inCiaddr;
}

/** Set 'your' (client) IP address.
 * @param inYiaddr 'your' (client) IP address
 */
public void setYiaddr (byte [] inYiaddr) {
 yiaddr = inYiaddr;
}

/** Set address of next server to use in bootstrap.
 * @param inSiaddr address of next server to use in bootstrap
 */
public void  setSiaddr (byte [] inSiaddr) {
 siaddr = inSiaddr;
```

```
    }

    /** Set relay agent IP address.
     * @param inGiaddr relay agent IP address
     */
    public void setGiaddr (byte [] inGiaddr) {
     giaddr = inGiaddr;
    }

    /** Set client harware address.
     * @param inChaddr client hardware address
     */
    public void setChaddr (byte[] inChaddr) {
     chaddr = inChaddr;
    }

    /** Set optional server host name.
     * @param inSname server host name
     */
    public void setSname (byte [] inSname) {
     sname = inSname;
    }

    /** Set boot file name.
     * @param inFile boot file name
     */
    public void  setFile (byte [] inFile) {
     file = inFile;
    }

    /** Set message destination port.
     * @param inPortNum port on message destination host
     */

    public void  setPort (int inPortNum) {
     gPort = inPortNum;
```

```
}

/** Set message destination IP
 * @param inHost string representation of message destination IP or
 * hostname
 */
public void  setDestinationHost (String inHost) {
    try {
     destination_IP = InetAddress.getByName(inHost);
  } catch (Exception e) {
     System.err.println(e);
  }
}

/*************************************************************
 * get* accesser fuctions return value of private data members*
 *************************************************************/

/** Get message Op code / message type. */
public byte getOp() {
 return op;
}

/** Get hardware address type.*/
public byte getHtype() {
 return    htype;
}

/** Get hardware address length.*/
public byte getHlen() {
 return    hlen ;
}

/** Get hops field.*/
public byte  getHops() {
 return    hops;
```

```
    }

    /** Get transaction ID.*/
    public int getXid() {
     return xid;
    }

    /** Get seconds elapsed since client began address acquisition or
     renewal process.*/
    public short getSecs() {
     return     secs;
    }

    /** Get flags field.*/
    public short  getFlags () {
     return flags;
    }


    /** Get client IP address.*/
    public byte[]  getCiaddr () {
     return     ciaddr;
    }

    /** Get 'your' (client) IP address.*/
    public byte[] getYiaddr () {
     return     yiaddr;
    }

    /** Get address of next server to use in bootstrap.*/
    public byte[]  getSiaddr () {
     return     siaddr;
    }

    /** Get relay agent IP address.*/
    public byte[] getGiaddr () {
```

```
 return     giaddr;
}

/** Get client harware address.*/
public byte[] getChaddr () {
return chaddr;
}

/** Get client harware address into String .*/
 public String getMACintoString(){
      return byteToString(chaddr[0])+":"+
               byteToString(chaddr[1])+":"+
               byteToString(chaddr[2])+":"+
               byteToString(chaddr[3])+":"+
               byteToString(chaddr[4])+":"+
               byteToString(chaddr[5]);
 }

 /** Get byte into hexa into string*/
 private String byteToString(byte Byte){
      if (Byte >= (byte)0 & Byte <= (byte)15){
           return ("0"+Integer.toHexString(Byte & 0xFF));
      }
      else{
           return (Integer.toHexString(Byte & 0xFF));
      }
 }


/** Get optional server host name.*/
public byte[] getSname () {
 return     sname;
}

/** Get boot file name.*/
public byte[]  getFile () {
```

```
 return     file;
}

/** Get all options.
 *@return a byte array containing options
 */
public byte[] getOptions() {
 if (optionsList == null) {
     Initialize();
 }
 return optionsList.externalize();
}

/** Get message destination port
 * @return an interger representation of the message destination port
 */
public int getPort() {
 return gPort;
}

/** Get message destination hostname
 * @return a string representing the hostname of the message
 * destination server
 */
public String getDestinationAddress() {
 return destination_IP.getHostAddress();
}


/** Sets DHCP options in DHCPMessage. If option already exists then remove
 * old option and insert a new one.
 * @param inOptNum  option number
 * @param inOptionData option data
 */

// Precondition: an option number, the length of the input data and the
```

```
// the data to go into the option.
// Postconditon: Parameters are placed into the options field and the
// pointer to the last index is incremented to the end.

public void setOption (int inOptNum, byte[] inOptionData) {
 optionsList.setOption((byte) inOptNum, inOptionData);
}

/** Returns specified DHCP option that matches the input code. Null is
 *  returned if option is not set.
 * @param inOptNum  option number
 */

public byte[] getOption (int inOptNum) {
 if (optionsList == null) {
     Initialize();
 }
 return optionsList.getOption((byte) inOptNum);
}

/** Removes the specified DHCP option that matches the input code.
 * @param inOptNum  option number
 */

public void removeOption(int inOptNum) {
 if (optionsList == null) {
     Initialize();
 }
 optionsList.removeOption( (byte) inOptNum);
}

/** Report whether or not the input option is set
 * @param inOptNum  option number
 */

/*
```

```
     * DHCPMessage::IsOptSet
     * Purpose: to return is a certain option is already set.
     * Precondition: a option number to lookup and a output parameter to
     * the index of it into.
     * Postcodition: if option is found, true is returned and so is the
     * index of that option in the options array. If it is not found, false
     * is returned.
     */

    public boolean IsOptSet(int inOptNum) {
     if (optionsList == null) {
         Initialize();
     }
     return optionsList.contains((byte) inOptNum);
    }


    /* for testing only*/
    public void printMessage() {
     byte[] data = externalize();
     for(int i = 0; i < 100; i++) {
         System.out.print(data[i]);
         if ( ((i % 25) == 0)  && (i != 0)) {
          System.out.print("\n");
         } else {
          System.out.print(" ");
         }
     }
     System.out.print("\n");
     if (optionsList == null) {
         Initialize();
     }
     optionsList.printList();
    }

}
```

```
package jDHCP;
import java.io.*;

import java.util.Hashtable;
import java.util.StringTokenizer;


/**
 * This class extract information variable from a .txt file for the configuration of
 * the DHCP service. The other class can get the information by the get methods
 *
 * To add a new information, you need to add it into the configfile.txt and add it into the
 * default value method that add them into the Hashtable valueOfPMIPClient.
 * Don't forget to create a get method to return the value
 *
 * @author Hercher Yann
 * @version 1  14/11/2008
 */
public class DHCPReadConfig {
     static Hashtable<String, String> valueOfPMIPClient = new Hashtable<String, String>();

     /**
     * Method that open the configuration file of the DHCP service
     * and place the information into a Hashtable
     */
     public void DHCPReadConfig_Start(){
          String variableName, value;
          String nameFile ="DHCPConfig.txt";        // Name of the config file

          try{
               // Search the file the ConfigFile folder placed in the same folder as the program
               InputStream ips=new FileInputStream(System.getProperties().get
                                                        ("user.dir")+"/config/"+nameFile);
               InputStreamReader ipsr=new InputStreamReader(ips);
               BufferedReader br=new BufferedReader(ipsr);
```

```
                // Check and place the default value in the HashTable
                defaultValues();

                // Vairable that take every line of the config file
                String line;

                StringTokenizer st;

          // Turn still the end of the documents (the buffer in facts)
                while ((line=br.readLine())!=null){

                      // Detect if a line is empty to jump an exception
                      if (line.isEmpty() != true){

                            // Discard the # comentary ligne
                            if (line.charAt(0) != '#'){

                                  // Separate the name and the variable
                                  st = new StringTokenizer(line," ");
                                  variableName = st.nextToken();

                                  // Keep the variable only if name exist in the HashTable
                                  if (valueOfPMIPClient.get(variableName) != null){
                                            value = st.nextToken();
                                            // Delete to replace the variable in the hashTable
                                            valueOfPMIPClient.remove(variableName);
                                            valueOfPMIPClient.put(variableName, value);
                               }
                            }
                      }
                }
                br.close();
          }
          catch (Exception e){
                System.out.println("Can't find, open or read the "+nameFile+". Check the correct path,
name and integrity");
```

```
            }
      }

      private static void defaultValues(){
            // Set the default option of the variable in the Hashtable
         valueOfPMIPClient.put("InterfaceIP_DHCP", new String("0.0.0.0"));
         valueOfPMIPClient.put("LeaseTime", new String("4294967295")); // max time
         valueOfPMIPClient.put("DefaultGateWay", new String("0.0.0.0")); // max time
      }

      /************************************************************
      * get* accesser fuctions return value of private data members*
      ************************************************************/

      /** Get the InterfaceIP_DHCP in byte array*/
      public static byte[] getInterfaceIP_DHCP(){
            StringTokenizer st = new
StringTokenizer((String)valueOfPMIPClient.get("InterfaceIP_DHCP"),".");
            return new byte[]{(byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken()))};
      }

      /** Get the getLeaseTime in long*/
      public static byte[] getLeaseTime(){
            long leaseTime = Long.parseLong((String)valueOfPMIPClient.get("LeaseTime"));
            return new byte[]{(byte)(leaseTime >> 24 & 0xFF),
                              (byte)(leaseTime >> 16 & 0xFF),
                              (byte)(leaseTime >> 8 & 0xFF),
                              (byte)(leaseTime & 0xFF)};
      }

      /** Get the DefaultGateWay in byte array*/
      public static byte[] getDefaultGateWay(){
            StringTokenizer st = new StringTokenizer(
```

```
                    (String)valueOfPMIPClient.get("DefaultGateWay"),".");
            return new byte[]{(byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken())),
                              (byte)(Integer.parseInt(st.nextToken()))};
        }

}
```

```
import jDHCP.DHCPReadConfig;
import jPMIPClient.FixeSetTTL;
import jPMIPClient.PMIPReadConfig;
import java.util.Hashtable;

/**
 * This class lunch some instance of PMIP Client for each new mobility Client
 * it take the information of mobility form the NAS and wait the DHCP information
 * to informe the thread to renew the tunnel.
 * @author Hercher Yann
 * @version 1   29/10/2008
 */
public class PMIP_Client {
     private static Hashtable<String, PMIP_Worker> workers;

     /**
      * Constructor that load the PMIP Client localy configuration
      * it set a problem of TTL IP header for Linux only
      */
     public void PMIP_Client_Start(){
          //Load the DHCPClient Information from the configuration file
          PMIPReadConfig PMIPLoadConfig = new PMIPReadConfig();
          PMIPLoadConfig.ReadPMIPConfig();

          //Load the DHCPClient Information from the configuration file
          DHCPReadConfig DHCPLoadConfig = new DHCPReadConfig();
          DHCPLoadConfig.DHCPReadConfig_Start();

          //Determinate wich OS is running
          if (System.getProperties().getProperty("os.name").compareTo("Linux") == 0){
               // Fix the TTL's bug of the multicastsocket
               FixeSetTTL fixebugTTL = new FixeSetTTL();
               fixebugTTL.FixeSetTTL_Start();
          }

          // HashTable wich contein an instance of each PMIP Client lunched
```

```
        workers = new Hashtable<String, PMIP_Worker>();
    }


    /**
     * Method called by the NAS for giving to the PMIP Client the mobiltiy information of
     * the Client
     * @param newMNinformatation
     */
    public synchronized static void newPMIPClient(MN_Data_Mobile newMNinformatation){
        // Create in the Hash table indexed by the MAC Address of the Client
        // the new worker that contain all the mobile information of the Client
        workers.put(newMNinformatation.getMAC(), new PMIP_Worker(newMNinformatation));
    }

    /**
     * Method called by the DHCP to inform the PMIP Client that a client just arrived
     * it will unlock the PMIP Client instance thread for continue the tunnel negosciation
     * @param MAC
     */
    public static boolean DHCPInforme(String MAC){
        //Check if a worker exist with the same ID
        if (workers.containsKey(MAC)){
            workers.get(MAC).setStateOfMN(true);
            workers.get(MAC).threadPMIP.interrupt(); //Unlock the Thread
            return true;
        }
        else{ // if not return false and the Client GHCP will find a IP Address by an other way
            return false;
        }
    }

    /**
     * Method that return a boolean if the thread exist in the HashTable indexed by MAC
     * @param MAC
     */
```

```
public static boolean existingThread(String MAC){
     return workers.containsKey(MAC);
}

/**
 * Method that remove the worker of the HashTable
 * @param MAC
 */
public static void removeWorker(String MAC){
     workers.remove(MAC);
}

/**
 * Method that return the status of the Client, True if he is still connected
 * @param MAC
 */
public synchronized static boolean getClientStatus(String MAC){
     if (workers.containsKey(MAC)){
          return workers.get(MAC).getStateOfMN();
     }else{
          return false;
     }
}

/**
 * Method that check if there is an existing thread that match with the MAC address
 * then change his status
 * @param MAC
 * @param status
 */
public synchronized static boolean setClientStatus(String MAC, boolean status){
     if (workers.containsKey(MAC)){
          workers.get(MAC).setStateOfMN(status);
          return true;
     }else{
          return false;
```

```
        }
    }

    /**
     * Method that will set the value of the IP MNHomeAddress for the worker indexed by
     * the MAC Address
     * @param MAC
     * @param IPv4MNHome
     */
    public synchronized static void setMNHomeIPAddress(String MAC, byte[] IPv4MNHome){
        workers.get(MAC).setMNHomeIPAddress(IPv4MNHome);
    }

    /**
     * Method that return the MNHomeIPAddress by the MAC Address
     * @param MAC
     */
    public synchronized static byte[] getMNHomeIPAddress(String MAC){
        return workers.get(MAC).getMNHomeIPAddress();
    }

    /**
     * Method that return the Renewal Time of the PMIP Tunnel
     * @param MAC
     */
    public synchronized static int getRenewalTimeTunnel(String MAC){
        return workers.get(MAC).getRenewalTimeTunnel();
    }

    /**
     * Method that set the Renewal Time of the PMIP tunnel
     * @param MAC
     * @param renewalTimeTunnel
     */
    public synchronized static void setRenewalTimeTunnel(String MAC, int renewalTimeTunnel){
        workers.get(MAC).setRenewalTimeTunnel(renewalTimeTunnel);
```

```
	}

	/**
	 * Method that return the boolean value of the tunnel state (up or down = true or false)
	 * @param MAC
	 */
	public synchronized static boolean getTunnelUp(String MAC){
		return workers.get(MAC).getTunnelUp();
	}

	/**
	 * Method that set the state of the Tunnel (up or down = true or false)
	 * @param MAC
	 * @param tunnelUp
	 */
	public synchronized static void setTunnelUp(String MAC, boolean tunnelUp){
		workers.get(MAC).setTunnelUp(tunnelUp);
	}


	/**
	 * Main method that lunch all the program
	 * Began to start itself
	 * then the NAS
	 * finaly the DHCP service
	 * @param args
	 */
	public static void main(String[] args) throws Exception{

//		Properties props = System.getProperties();
//		props.setProperty("java.net.preferIPv4Stack","true");
//		System.setProperties(props);

		PMIP_Client PMIPclient = new PMIP_Client();
		PMIPclient.PMIP_Client_Start();
		System.out.println("**********************");
```

```
            System.out.println("***  PMIP lunched  ***");
            System.out.println("**********************");

            NAS nas = new NAS();
            nas.NAS_Start();
            System.out.println("**********************");
            System.out.println("***  NAS lunched  ***");
            System.out.println("**********************");

            DHCP_Dispatcher DHCP = new DHCP_Dispatcher();
            DHCP.DHCP_Start();
            System.out.println("**********************");
            System.out.println("***  DHCP lunched  ***");
            System.out.println("**********************");
      }
}
```

```
/**
 * This class contein when instencied the thread for each PMIP Client active on the PMA
 * it give by assor the value that the DHCP need for giving the address to the client
 * @author Hercher Yann
 * @version 1   29/10/2008
 */

public class PMIP_Worker {

	public PMIP_Instance threadPMIP;
	private boolean stateOfMN = false;
	private boolean tunnelUp = false;
	private byte[] MNHomeIPAddress = null;
	private int renewalTimeTunnel = 20;

	/**
	 * Constructor that create the PMIP Client instance threadand set the state of the MN to false
	 * @param newMNinformatation
	 */
	public PMIP_Worker(MN_Data_Mobile newMNinformatation){
		threadPMIP = new PMIP_Instance(newMNinformatation);
		stateOfMN = false;
	}

	/**
	 * assesor to set the state of the MN
	 * @param stateOfMN
	 */
	public synchronized void setStateOfMN(boolean stateOfMN){
		this.stateOfMN = stateOfMN;
	}

	/**
	 * assesor to get the state of the MN in boolean
	 */
	public synchronized boolean getStateOfMN(){
```

```
        return stateOfMN;
    }

    /**
     * assesor to get in byte array the MN Home IP Address
     */
    public synchronized byte[] getMNHomeIPAddress(){
        return MNHomeIPAddress;
    }

    /**
     * assesor to set by byte array the MN Home IP Address
     * @param MNHomeIPAddress
     */
    public synchronized void setMNHomeIPAddress(byte[] MNHomeIPAddress){
        this.MNHomeIPAddress = MNHomeIPAddress;
    }

    /**
     * assesor to get in integer the renewal tunnel's time
     */
    public synchronized int getRenewalTimeTunnel(){
        return renewalTimeTunnel;
    }

    /**
     * assesor to set by integer the renewal tunnel's time
     * @param renewalTimeTunnel
     */
    public synchronized void setRenewalTimeTunnel(int renewalTimeTunnel){
        this.renewalTimeTunnel = renewalTimeTunnel;
    }

    /**
     * assesor to get the value of the tunnel's state
     */
```

```
	public synchronized boolean getTunnelUp(){
		return tunnelUp;
	}

	/**
	 * assesor to set by boolean the tunnel's state
	 * @param tunnelUp
	 */
	public synchronized void setTunnelUp(boolean tunnelUp){
		this.tunnelUp = tunnelUp;
	}
}
```

```
import jPMIPClient.*;
import java.io.IOException;
import java.sql.Date;
import java.sql.Timestamp;
import java.util.StringTokenizer;
import java.util.concurrent.CountDownLatch;


/**
 * This class represents each PMIP Client by an instance that establish the tunnel
 * with the HA and try to maintain it until the mobile client leave the network
 * he contain all the logic for sending and receiving the messages and analyse them.
 * @author Hercher Yann
 * @version 1   29/10/2008
 */
public class PMIP_Instance implements Runnable{

     private static PMIPClientSocket socket;
     private PMIPClientMessage inMessage;
     private PMIPClientMessage messageOut;
     private Timestamp timeStampDate ;
     private long offsetTime;
     private CountDownLatch startSignal;
     private sleepTime t_waitingTime ;

     private Thread t;
     private MN_Data_Mobile newMNinformatation;


     /**
      * Constructor that start itself in thread and keep in local the mobility client informations
      * @param newMNinformatation
      */
     public PMIP_Instance(MN_Data_Mobile newMNinformatation){
          this.newMNinformatation = newMNinformatation;
          t = new Thread(this);
```

```
        t.start();
    }

    /**
     * Method that try to establish the tunnel between the FA and the HA
     * return false if the negosciation is interrupt or can't be terminated normally
     */
    private boolean tunnelMessageEstablishement(){
        // Create a socket on a random port that will be the same during all
        // the the tunnel's life
        try {
            socket = new PMIPClientSocket();

            //Create and send the PRRQ message
            socket.send(firstPRRQ());

            System.out.println("PMIP Client : sending PRRQ to HA");

            //Receive the PRRP message;
            socket.receive(inMessage = new PMIPClientMessage());
            System.out.println("PMIP Client : receive PRRP from HA");

            // turn until be stopped by a return
            while(true){
                //Check the validity and integrity of the message by authentication field
                if(checkAuthentication()){
                    // Check the status of the message by the code field
                    switch(messageAnalyzer((int)inMessage.getCode() & 0xFF)){
                    case 0 : return false;  // unknown error message
                    case 1 : return true;   // message ok continue normaly
                    default : // try to resolve the problem bye sendinf a new message to the HA
                                socket.send(messageOut);
                                System.out.println("PMIP Client : sending PRRQ to HA");
                                socket.receive(inMessage = new PMIPClientMessage());
                                System.out.println("PMIP Client : receive PRRP from HA");
                                break;
```

```
                    }
                }else{
                    System.out.println("Error : the message is corrupted");
                    // Discard the message ... Message corrupted or incerect key
                    return false;
                }
            }
        } catch (IOException e) {
            System.out.println("Problem to create the Socket for the PMIPClient");
            return false;
        }
    }

    /**
     * Method that receive the code of the incoming message, analyse him and change the PRRQ message
     * to send back
     */
    private int messageAnalyzer(int code){
        int code_to_return = code;
        switch (code){
        case 1:         break; // Correct message
        case 133 :
                        System.out.println("PMIP Client : error 133 Identification field not
correct");
                        // take the time in timestamp format of the server from the incoming message
                        long timeStampServeur = inMessage.getHigh32BitsIdentification();

                        // take the actualy time on the local machine in timestamp format
                        timestamp();

                        // create the offset Time
                        if (timeStampServeur > (timeStampDate.getTime()/1000)){
                            offsetTime = timeStampServeur - (timeStampDate.getTime()/1000);
                        }else{
                            offsetTime = (timeStampDate.getTime()/1000) - timeStampServeur;
                        }
```

```
                        // Set the new identification field in the PRRQ message

messageOut.setHigh4ByteIdentification(offsetTime+timeStampDate.getTime()/1000);

                        // Set the new authentication value of the Extension Security type 32
                        setAuthentication();

                        // Wait 1 second befor send a new message that's specified in the RFC3344
                        try {
                             startSignal = new CountDownLatch(1);
                             t_waitingTime = new sleepTime(1000, startSignal);
                             t_waitingTime.sleepTime_Start();
                             startSignal.await();
                        } catch (InterruptedException e) {}

                        break;
     default :  code_to_return = 0;
                        System.out.println("Don't " +
                                  "know this error PRRP message");
                        break;
     }
     return code_to_return;
}

/**
 * Method that set the new authenticator field for the PRRQ message (messageOut)
 * At first it find the position of the authenticator field
 * then he create a new authenticator object with the real authenticator field of the PRRQ to send
 * then it create a new hashMac md5 with the entire byte message buffer and the shareSecret.
 */
private void setAuthentication(){
     // Find where is the extension Security authentication type 32
     int posExtAuthentication = messageOut.getExtensionPos(32);

     // Create a new extension
```

```
            ExtensionTypeSecurityAuthentication securityExtension =
            (ExtensionTypeSecurityAuthentication)messageOut.getExtension(posExtAuthentication);

            // set the new Authenticator field in the extension
            securityExtension.setAuthenticator(newMNinformatation.getShareSecret(),
                        messageOut.externalize(newMNinformatation.getSPI()));

            // replace the old extension with the new one
            messageOut.setExtension(posExtAuthentication, securityExtension);
      }

       /** Check if the authentication field is the same by encoding a new time the
        * received message with his own secret key. If the corresponding is not true
        * that mean that secret key is not the same in the HA or the message has been
        * corrupted
        * return a boolean for respond by true or false
      */
      private boolean checkAuthentication(){
            boolean condition = false;
            //Search in the list the security extension type 32
            for (int i = 0; i < inMessage.getExtensionLength() ; i++){
                  int typeMessage = inMessage.getExtension(i).getType();
                  // if he find him he try to check the validity
                  // if not then discard the mesage because this extension MUST
                  //be in the message specified in RFC 3344
                  if (typeMessage == 32){
                        // Take the security extension in the list and place it into an object
                        ExtensionTypeSecurityAuthentication securityExtensionPRRP =
(ExtensionTypeSecurityAuthentication)inMessage.getExtension(i);

                        // Create a new security Extension with the message that just arrived
                        // with it own SPI and Secret KEY
                        ExtensionTypeSecurityAuthentication securityExtensionVerification = new
ExtensionTypeSecurityAuthentication();
                        securityExtensionVerification.setAuthenticator(newMNinformatation.getShareSecret(),
```

```
inMessage.externalize(newMNinformatation.getSPI()));


                        // Compare the twice authentication field
                        if
(securityExtensionPRRP.compareTo(securityExtensionVerification.getAuthenticator())){
                              //Match ID
                              condition = true;
                        }else{condition = false;} //dismatch ID => discard message
                  }else{
                        //No extension type 32 => discard message
                        condition = false;
                  }
            }
            return condition;
      }

      /**
       * Catch the time of the local machine and convert it into TimeStamp
      */
      private void timestamp(){
            //Take the Date From the local machine
            Date date = new Date(new java.util.Date().getTime());
            // Convert it to TimeStamp
            timeStampDate = new Timestamp(date.getTime());
      }

      /**
       * Create the initial messageOut message to establish the connection
      */
      private PMIPClientMessage firstPRRQ() {

            messageOut = new PMIPClientMessage();

            messageOut.setType((byte)1); // 1 for messageOut
```

```
messageOut.setS(newMNinformatation.getSimultaneousBindings());
messageOut.setB(newMNinformatation.getBroadcastDatagrams());
// not set the D because the Draft defined that MUST be set to 0
messageOut.setM(newMNinformatation.getMinimalEncapsulation());
messageOut.setG(newMNinformatation.getGreEncapsulate());
messageOut.setr((byte)0);
messageOut.setT((byte)1);//newMNinformatation.getreverseTunelling());
messageOut.setx((byte)0);

byte[] Lifetime = {(byte) (PMIPReadConfig.getMNDefaultTunnelLifetime() >> 8 & 0xFF),
                   (byte) (PMIPReadConfig.getMNDefaultTunnelLifetime() & 0xFF)};
messageOut.setLifetime(Lifetime);

messageOut.setHomeAddress(newMNinformatation.getMNHomeIPAddressV4());

messageOut.setHomeAgent(newMNinformatation.getHAIPAddressV4());

// get the COAddress from the config file and tokenize it with the "."
StringTokenizer st = new StringTokenizer(PMIPReadConfig.getCOAddress(),".");

// Convert the String tokenized into Integer befors cast to Byte
byte[] COAddress = {(byte) Integer.parseInt(st.nextToken()),
                    (byte) Integer.parseInt(st.nextToken()),
                    (byte) Integer.parseInt(st.nextToken()),
                    (byte) Integer.parseInt(st.nextToken())};
messageOut.setCOAddress(COAddress);

// Check the time of the local machine and convert im in timestamp format
timestamp();
// set the High 4 Byte Identification field with the Second of the TimeStamp
// TimeStamp need to by divide by 1000 to keep only the second
messageOut.setHigh4ByteIdentification(timeStampDate.getTime()/1000);

// set the fractional second in the lowest 4 byte of the identification Field
// and complet them with a good randomize
```

```
        int fractioOfSeconds = timeStampDate.getNanos();
        fractioOfSeconds += (int)(Math.random()*1000000);
        messageOut.setLow4byteIdentification(fractioOfSeconds);

        // add the security extension without the authentification field
        messageOut.addExtensions(new ExtensionTypeSecurityAuthentication("",32,
                                                    newMNinformatation.getSPI()));

        setAuthentication();

        return (messageOut);
    }

    /**
     * Create the initial messageOut message to establish the connection
    */
    private PMIPClientMessage renewalmessageOut(){
        timestamp();
        //set the new time in second in the 4 high byte Identification field
        messageOut.setHigh4ByteIdentification(offsetTime+timeStampDate.getTime()/1000);

        // and the fractional seconds in the low 4
        int fractioOfSeconds = timeStampDate.getNanos();
        fractioOfSeconds += (int)(Math.random()*1000000);
        messageOut.setLow4byteIdentification(fractioOfSeconds);

        // Set the new authentication value of the Extension Security type 32
        setAuthentication();

        return (messageOut);
    }


    public void run(){
```

```
          System.out.println("Thread lancer -> attend l'arriver du DHCP  (wait)"); //reste a
implÃ©menter un temp de timeOut

          try {
               t.sleep(Integer.parseInt(PMIPReadConfig.getTimeBeforeKillThread())*1000); //multiply by
1000 to obtain time in second
          } catch (InterruptedException e) {}

          if (PMIP_Client.getClientStatus(newMNinformatation.getMAC())) {

               // Continue if the Tunnel is set
               if (tunnelMessageEstablishement()){
                    PMIP_Client.setTunnelUp(newMNinformatation.getMAC(), true);

                    // Set la variable presence client a false
//                  PMIP_Client.setClientStatus(newMNinformatation.getMAC(), false);

                    // Give the tunnel life time to the PMIP value for the DHCP can recover this time
                    PMIP_Client.setRenewalTimeTunnel(newMNinformatation.getMAC(),
inMessage.getLifeTime());

                    // Give the MNHomeIPAddress to the PMIP value for the DHCP can recover this and
give to its client
                    PMIP_Client.setMNHomeIPAddress(newMNinformatation.getMAC(),
                                                        inMessage.getHomeAddress());

                    // Unlock the DHCP for respond to the client
                    DHCP_Dispatcher.unLockThreadDHCP(newMNinformatation.getMAC());

                    // Wait the tunnel renewal time before continue
                    startSignal = new CountDownLatch(1);
                    t_waitingTime = new sleepTime(inMessage.getLifeTime(), startSignal);
                    t_waitingTime.sleepTime_Start();
                    try {
                         startSignal.await();
                    } catch (InterruptedException e1) {
```

```
                    }

                    // Turn until the client leave the network
                    while(PMIP_Client.getClientStatus(newMNinformatation.getMAC())){
                         try {



//                              Set the state of the MN to false
                              PMIP_Client.setClientStatus(newMNinformatation.getMAC(), false);

                              // Send a renewal messageOut with a new Identification field and
                              // extension authenticator security type 32
                              socket.send(renewalmessageOut());
                              System.out.println("PMIP Client : Tunnel rennewing");


                              inMessage = new PMIPClientMessage();

                              //Receive the PRRP message;
                              socket.receive(inMessage);

                              // Leave if the message is corrupted or if is not correct
                              if (checkAuthentication() || inMessage.getCode() != (byte)1){
                                   break;
                              }

                              startSignal = new CountDownLatch(1);
                              t_waitingTime = new sleepTime(inMessage.getLifeTime(), startSignal);
                              t_waitingTime.sleepTime_Start();
                              startSignal.await();

                         } catch (InterruptedException e) {
                              System.out.println("PMIP Client : Error, can't renew the tunnel, close
the thread");
                         }
```

```
                    }
               }else{
                    System.out.println("PMIP Client : Can't established the tunnel, close the thread");
               }

               if (!PMIP_Client.getClientStatus(newMNinformatation.getMAC()))
                    System.out.println("PMIP Client : Client is gone, close the thread ");

          }
          else{
               //ne fait rien et ferme le Thread
               System.out.println("No DHCP client has become so close the Thread");
          }
          System.out.println("PMIP Client : Release the resource on the PMIP hashTable");
          PMIP_Client.setTunnelUp(newMNinformatation.getMAC(), false);
          PMIP_Client.removeWorker(newMNinformatation.getMAC());
    }

     /**
      * Method to interupt the sleep when the DHCP arrived for the first time
     */
     public void interrupt(){
          t.interrupt();
     }
}

/**
 * Internal class that lunch a thread wich will wait the Renewal tunnel time
 * before unlock the PMIP instance that will be send a new messageOut tunnel renewal
 */
class sleepTime implements Runnable{
     private int timeToSleep; //into miliseconds
     private Thread t;
     CountDownLatch startSignal;

     /**
```

```
	 * Constructor that get the time to sleep before unlock the PMIP Client
	 * @param timeToSleep
	 * @param startSignal
	 */
	public sleepTime(int timeToSleep, CountDownLatch startSignal){
		this.startSignal = startSignal;
		this.timeToSleep = timeToSleep;
	}

	/**
	 * Methode that start itself with the sleep time
	 */
	public void sleepTime_Start(){
		t = new Thread(this);
		t.start();
	}

	/**
	 * Method that lunch the thread and sleep during the renewal tunnel time
	 */
	public void run() {
		try {
			t.sleep(timeToSleep * 1000); // * 1000 to wait time in sec ...
			startSignal.countDown();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

```
import jDHCP.DHCPMessage;
import jDHCP.DHCPSocket;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.CountDownLatch;

/**
 * This class represents a message's Dispatcher who's coming from a socket.
 * this function is to determinate and separate the incoming packet between
 * the different worker
 * @author Hercher Yann
 * @version 1   29/10/2008
 */
public class DHCP_Dispatcher implements Runnable {
     static DHCPSocket mySocket;
     static private Map<String, workerDHCP> workers;  // Contains the different Client instance
                                                      // indexed by their MAC address in
String
     private Thread t;


     /**
     * method for creating DHCP_Dispatcher who listen the port 67
     * and receive the DHCP message for dispatching them between the different
     * workers mapped in a hashmap. Lunch itself in thread
     */
     public void DHCP_Start() {
          t = new Thread(this);
          t.start();
     }

     /**
     * Static Method that return the memory address of the running socket
     */
```

```
public static DHCPSocket getSocket(){
      return mySocket;
}

/**
* Method that listen the incoming packet on the port 67 and dispatch them between
* the different DHCP Client instance
*/
public void run() {
      DHCPMessage messageIn;

            try {
                  mySocket = new DHCPSocket(67);
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();                        // Future amelioration
            }

      workers = new HashMap<String, workerDHCP>();

      while (true){
            messageIn = new DHCPMessage();
            mySocket.receive(messageIn);

            //Search if the MAC Address is still in the HashMap
            if (workers.containsKey(messageIn.getMACintoString())) {
                  //he's so send only a new update message;
                  workers.get(messageIn.getMACintoString()).getNewMessage(messageIn);
            }else{
                  //he's not so create a new worker in the HashMap and send him the message
                  workers.put(messageIn.getMACintoString(), new workerDHCP(messageIn));
            }
      } //Return to the infinited While(true) for continue the listening
}

/**
```

```
     * Method to unlock the DHCP when he is waiting the MNHomeIPAddress
     * @param MAC
     */
     public static void unLockThreadDHCP(String MAC){
          workers.get(MAC).unLock();
     }
}

/**
 * internal class to create an object to place in the HashMap
 * it contein the insatnce of the DHCP client and the locker signal CountDownLatch
 */
class workerDHCP {
     private CountDownLatch startSignal;
     private DHCP_Instance newDHCP_Instance;

     /**
      * Constructor that create the new instance for the DHCP Client and give it the first
      * incoming message. Bind to it a startSignal to unlock him when he wait the MNHomeIPAddress
      * @param messageIn
      */
     public workerDHCP(DHCPMessage messageIn){
          startSignal = new CountDownLatch(1);
          newDHCP_Instance = new DHCP_Instance(messageIn, startSignal);
     }

     /**
     * Method to unlock the DHCP when he is waiting the MNHomeIPAddress
     */
     public void unLock(){
           startSignal.countDown();
     }

     /**
     * Method that give a new incoming message to an existing instance
     */
```

```
	public void getNewMessage(DHCPMessage messageIn){
		newDHCP_Instance.getNewMessage(messageIn);
	}
}
```

```
import java.io.IOException;
import java.util.concurrent.CountDownLatch;
import jDHCP.DHCPMessage;
import jDHCP.DHCPReadConfig;


/**
 * When have been instancied, this class lunch hitself into thread to respond when
 * It receive a message DHCP form the Client and know witch kind of response send back
 * It send some information to the PMIPClient to inform the presence of the Client.
 * @author Hercher Yann
 * @version 1  29/10/2008
 */
public class DHCP_Instance implements Runnable {
     private final CountDownLatch startSignal; // Lock the DHCP sequence and is unlock by
                                                          // the PMIPClient
     private DHCPMessage inMessage;                  // Message DHCP incoming
     private DHCPMessage messageOut;                 // Message DHCP outgoing
     private int halfTunnelTime = 0;                 // Time of the renwal and bind time of DHCP
                                                          // that is the half of the tunnel time
life.

     private byte[] value;                           // Byte array value of each option of the
                                                          // DHCP Message



     /**
     * Constructor wich call immediately the newmessage method that know what
     * to do with the incoming message (inMessage)
     * @param inMessage
     * @param startSignal
     */
     public DHCP_Instance(DHCPMessage inMessage, CountDownLatch startSignal){
          this.startSignal = startSignal;
          getNewMessage(inMessage);
     }
```

```
/**
* Method called for send the message when it's build.
 * @throws IOException
*/
private void send() throws IOException {
     DHCP_Dispatcher.getSocket().send(messageOut);
}


/**
 * Method getNewMessage analyze the Option field 53 (Message Type Field)
 * For determined which work run in the thread
 * @param messageIn
 */
public void getNewMessage(DHCPMessage inMessage){
     this.inMessage = inMessage;
     new Thread(this).start();
}

/**
 * Method getTimeFromTimeTunnel check the renewalTimeTunnel of the PMIP Client
 * This value is split by 2 for become the renewal and bind time of the DHCP Message
 */
private void getTimeFromTimeTunnel(){
     // Check if a thread existe with the mac address of the client
     if (PMIP_Client.existingThread(inMessage.getMACintoString()))
          halfTunnelTime = PMIP_Client.getRenewalTimeTunnel(inMessage.getMACintoString())/2;
}

/**
 * Method that create a new DHCP Offer message with the received the MNHomeIPAddress
 * form the PMIP Client
 * @param MNHomeIPAddress
 */
private void DHCPOffer (byte[] MNHomeIPAddress){
```

```
            messageOut = new DHCPMessage(); // Create a new instance of a DHCP Message
            messageOut.setPort(68);              // set the right destination port 68
            messageOut.setOp((byte)2);
            messageOut.setHtype((byte)1);   //1 = Ethernet
            messageOut.setHlen((byte)6);
            messageOut.setXid(inMessage.getXid());  // set the same Xid as the received message

            // Set the MNHomeIPAdress in the Yiadrr field of the DHCP Message
//          value = new byte[]{MNHomeIPAddress[0], MNHomeIPAddress[1],
//                             MNHomeIPAddress[2], MNHomeIPAddress[3]};
            value = new byte[]{(byte)192, (byte)168, (byte)0, (byte)5};
            messageOut.setYiaddr(value);

            // Set the serveur Address with the IP address of the DHCP service in the config file
            //messageOut.setSiaddr(DHCPReadConfig.getInterfaceIP_DHCP());

            // set the Hardware Client Address from the incoming Client message
            messageOut.setChaddr(inMessage.getChaddr());

            // ****************
            // * Option Field *
            // ****************

            // Option 53 = Define the message (Offer = 2 , Request, Ack = 5 , etc ..)
            value = new byte[]{2};               //value 2 Define a DHCPOffer message
            messageOut.setOption(53, value);

            // Check the current rewalTunnelTime of the PMIP to create the DHCP times
            getTimeFromTimeTunnel();

            // Option 58 = Renewal Time Value
            value = new byte[]{(byte)0, (byte)0 , (byte)((halfTunnelTime >> 8) & 0xff),
                                                  (byte)(halfTunnelTime & 0xff)};
            messageOut.setOption(58, value);

            // Option 59 = Rebinding Time Value
```

```
            value = new byte[]{(byte)0, (byte)0 , (byte)((halfTunnelTime >> 8) & 0xff),
                                                        (byte)(halfTunnelTime & 0xff)};

            messageOut.setOption(59, value);

            // Option 51 = IP Address Lease Time
            value = new byte[]{(byte)0, (byte)0 , (byte)1,
                        (byte)44};
            messageOut.setOption(51, value);
//          messageOut.setOption(51, DHCPReadConfig.getLeaseTime());

            // Option 1 = Subnet Mask
            value = new byte[]{(byte)255, (byte)255, (byte)255, (byte)0};  // need future amelioration
            messageOut.setOption(1, value);

            // Option 54 = Serveur Identifier
            messageOut.setOption(54, DHCPReadConfig.getInterfaceIP_DHCP());

            // Option 3 = Router Address = Default Gateway
            messageOut.setOption(3, DHCPReadConfig.getDefaultGateWay());

            // Option 6 = DNS server
            value = new byte[]{(byte)10, (byte)192, (byte)48, (byte)100,  // need future amelioration
                                    (byte)10, (byte)192, (byte)48, (byte)101};
            messageOut.setOption(6, value);

            try {
                  send();
            } catch (IOException e) {
                  // TODO Auto-generated catch block                        // need future amelioration
                  e.printStackTrace();
            }
      }


      /**
```

```
 * Method that create a new DHCP Ack message
 * @param MNHomeIPAddress
 */
private void DHCPAck (byte[] MNHomeIPAddress){
     messageOut = new DHCPMessage();
     messageOut.setPort(68);                          // set the right destination port 68
     messageOut.setOp((byte)2);
     messageOut.setHtype((byte)1);              //1 = Ethernet
     messageOut.setHlen((byte)6);
     messageOut.setXid(inMessage.getXid());  // set the same Xid as the received message

     // Set the MNHomeIPAdress in the Yiadrr field of the DHCP Message
     value = new byte[]{MNHomeIPAddress[0], MNHomeIPAddress[1],
                        MNHomeIPAddress[2], MNHomeIPAddress[3]};

     messageOut.setYiaddr(value);

     // set the Hardware Client Address from the incoming Client message
     messageOut.setChaddr(inMessage.getChaddr());

     // ****************
     // * Option Field *
     // ****************

     // Option 53 = Define the message (Offer , Request, Ack = 5 , etc ..)
     value = new byte[]{5};  //Define the DHCPAck message
     messageOut.setOption(53, value);

     // set the time value to return
     getTimeFromTimeTunnel();

     // Option 58 = Renewal Time Value
     value = new byte[]{(byte)0, (byte)0 , (byte)((halfTunnelTime >> 8) & 0xff),
                                                  (byte)(halfTunnelTime & 0xff)};
     messageOut.setOption(58, value);
```

```
            // Option 59 = Rebinding Time Value
//          value = new byte[]{(byte)0, (byte)0 , (byte)((halfTunnelTime >> 8) & 0xff),
//                                                          (byte)(halfTunnelTime & 0xff)};
            value = new byte[]{(byte)0, (byte)0 , (byte)0,
                        (byte)40};
            messageOut.setOption(59, value);

            // Option 51 = IP Address Lease Time
            value = new byte[]{(byte)0, (byte)0 , (byte)1,
                        (byte)44};
            messageOut.setOption(51, value);
//          messageOut.setOption(51, DHCPReadConfig.getLeaseTime());

            // Option 1 = Subnet Mask
            value = new byte[]{(byte)255, (byte)255, (byte)255, (byte)0};  // need future amelioration
            messageOut.setOption(1, value);

            // Option 54 = Serveur Identifier
            messageOut.setOption(54, DHCPReadConfig.getInterfaceIP_DHCP());

            // Option 3 = Router Address = Default Gateway
            messageOut.setOption(3, DHCPReadConfig.getDefaultGateWay());

            // Option 6 = DNS server
            value = new byte[]{(byte)10, (byte)192, (byte)48, (byte)100,  // need future amelioration
                               (byte)10, (byte)192, (byte)48, (byte)101};
            messageOut.setOption(6, value);

            try {
                  send();
            } catch (IOException e) {                                    // need future amelioration
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
```

```
	/**
	 * Method that create a refused Nack message
	 *
	 */
	private void DHCPNack(){
		messageOut = new DHCPMessage();
		messageOut.setPort(68);                          // set the right destination port 68
		byte[] value;
		messageOut.setOp((byte)2);    //2 = BootReply
		messageOut.setHtype((byte)1);   //1 = Ethernet
		messageOut.setHlen((byte)6);

		// set the same parameters that the incoming message DHCP to NACK
		messageOut.setXid(inMessage.getXid());
		messageOut.setFlags(inMessage.getFlags());
		messageOut.setGiaddr(inMessage.getGiaddr());
		messageOut.setChaddr(inMessage.getChaddr());

		// ****************
		// * Option Field *
		// ****************

		// Option 53 = Define the message (Offer , Request, Ack = 5 , etc ..)
		value = new byte[]{6};  //Define the DHCPAck message
		messageOut.setOption(53, value);

		// Option 54 = Serveur Identifier
		messageOut.setOption(54, DHCPReadConfig.getInterfaceIP_DHCP());

		try {
			send();
		} catch (IOException e) {                                    // need future
amelioration
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
```

```
	}

	/**
	 * Method called when a DHCPDiscover is incoming
	 */
	private void DHCPDiscover(){
		System.out.println("DHCP : DHCPDiscover receive : "+inMessage.getMACintoString());

		// Check if there is a existing waiting thread for this incoming client
		if (PMIP_Client.existingThread(inMessage.getMACintoString())){
			// check if this thread is still locked and the tunnel is not up
			if (!PMIP_Client.getTunnelUp(inMessage.getMACintoString())){
				try {
					// Informe the thread that a new DHCP client arrived and unlock it
					PMIP_Client.DHCPInforme(inMessage.getMACintoString());

					// Stop the DHCP negosciation until the HA have give the MNIPAdress back
					startSignal.await();

					byte[] MNHomeIPAddress =
PMIP_Client.getMNHomeIPAddress(inMessage.getMACintoString());

					if (MNHomeIPAddress == null){
						// The HA dont give it back a DHCP adress for the Mobile Node
						// So continue the negosciation with an normaly DHCP server
						// need future amelioration
						System.out.println("Relay the DHCP Message for the next server");
					}else{
						DHCPOffer(MNHomeIPAddress);
						System.out.println("DHCP : DHCPOffer send");
					}
				} catch (InterruptedException e) {
					e.printStackTrace();
				}

			}else{
```

```
                    // tunnel is still up so ignored Discover message
               }
          } else{ // continue the negosciation with an normaly DHCP server
               System.out.println("Relay the DHCP Message for the next server");
          }
     }

     /**
      * Method called when a DHCPRequest arrived
      *
      */
     private void DHCPRequest(){
          System.out.println("DHCP : DHCPRequest receive");

          // if the thread is still running
          if (PMIP_Client.existingThread(inMessage.getMACintoString())){
               // if the tunnel is still up respond normaly to the DHCP message
               if (PMIP_Client.getTunnelUp(inMessage.getMACintoString())){
                    // Informe the PMIP Client that the client is still connected
                    PMIP_Client.setClientStatus(inMessage.getMACintoString(), true);

                    // send the Ack response to the Client
                    DHCPAck(PMIP_Client.getMNHomeIPAddress(inMessage.getMACintoString()));
                    System.out.println("DHCP : DHCPAck send");
               }else{ // if not that's mean we have change of FA and the PMIP wiat to be unlock
                    PMIP_Client.DHCPInforme(inMessage.getMACintoString());
                    DHCPAck(PMIP_Client.getMNHomeIPAddress(inMessage.getMACintoString()));
                    System.out.println("plop");
               }
          }else{ // The thread doesn't exist so respond by a Nack for that client renew his IP
               DHCPNack();
               System.out.println("DHCP : DHCPNAck send");
          }
     }

     /**
```

```
       * Method that run the correct work to do for the incoming message
       */
      public void run() {
            switch(inMessage.getOption(53)[0]){
                  case 1:  DHCPDiscover(); break;
                  case 3:  DHCPRequest();  break;
                  default: System.out.println("Option field 53 not reconeized => did nothing"); break;
            }
      }
}
```

```
import jDHCP.DHCPReadConfig;

import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import jpcap.JpcapCaptor;
import jpcap.NetworkInterface;
import jpcap.NetworkInterfaceAddress;
import jpcap.packet.UDPPacket;

/**
 * This class simule a NAS (Network access server) for the PMIP Client
 * His real Job is to sniff the network to find any Radius accept message. When it catch
 * one of them, he take the name in the option field and search the mobile information
 * a text file. If it find a match corespondance between the name and the name infromation
 * in the text file, it directly inform the PMIP Client and give it the mobility informations
 * A future amelioration is to implement a Diameter NAS solution that will give to the PMIP
 * the mobile Client information.
 * @author Hercher Yann
 * @version 1   29/10/2008
 */
public class NAS implements Runnable{

     private Thread t;

     /**
      * method that run the NAS in thread itself
      */
     public void NAS_Start(){
          t = new Thread(this); // instanciation du thread
          t.start(); // demarrage du thread, la fonction run() est ici lancÃ©e
     }
```

```
/**
 * Method SnifferUDP listen on a specific interface determined by its IP address
 *
 */
private void SnifferUDP(){
     int index = 0; // index of the interface that need to be open.
     //Obtain the list of network interfaces
     NetworkInterface[] devices = JpcapCaptor.getDeviceList();
     //for each network interface
     for (int i = 0; i < devices.length; i++) {
          // take their address and compare it to find wich does we need
          for (NetworkInterfaceAddress a : devices[i].addresses)
               if (a.address.toString().compareTo("/"+DHCPReadConfig.getInterfaceIP_DHCP()) == 0)
                    System.out.println("sdyg");
                    index = i;
     }

     try {
          // Try to open the devices
          JpcapCaptor captor=JpcapCaptor.openDevice(devices[index], 65535, true, 20);
          // set a filter to listen only the Radius message on port 1812
          captor.setFilter("port 1812", true);
          while(true){
               UDPPacket radiusPacket = (UDPPacket) captor.getPacket();  // error
               if (radiusPacket != null && radiusPacket.data[0]==2)
                    infoClient(findIdClient(radiusPacket.data));
          }
     } catch (IOException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
     }
}

/**
 * Method to the name in the Radius message
```

```
     * @param ibuff
     */
    private String findIdClient(byte[] ibuff){
         // Convert the Radius Byte array message into a byte Stream
         ByteArrayInputStream inBStream = new ByteArrayInputStream(ibuff, 0, ibuff.length );
    DataInputStream inStream = new DataInputStream (inBStream);

    String idClient = ""; // Value of client identifier to return

    try {
         // Jump the header radius message
         byte[] dumpBuffer = new byte[20];
         inStream.readFully(dumpBuffer, 0, 20);

         int length;
         // Search the field number 1 wich contein the name
         while (inStream.available() > 0) {
              // Check in the AVP value the code 1 to find the name ( the idClient )
              if ((inStream.readByte()) == (byte)1){
                        length = inStream.readByte();
                        length &= 0xff;
                        length -= 2;
                        // Converte the bytes into a String name
                        for (int i = 0 ; i < length ; i++){
                             idClient += ""+(char)(inStream.readByte() & 0xff);
                        }
                        System.out.println("NAS : new client detected name : "+idClient);
                   }else{ // not the correct field jump it
                        length = inStream.readByte();
                        length &= 0xff;
                        length -=2;
                        dumpBuffer = new byte[length];
                        inStream.readFully(dumpBuffer, 0, length);
                   }
         }
    } catch (IOException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
           }  //code
      return idClient;
      }

      /**
       * Method that try to find the name into a configuration text file with the name from
       * the Radius message
       * @param idClient
       */
      private void infoClient(String idClient){
           try{
                // Open the Configuration text File
                InputStream ips=new
FileInputStream(System.getProperties().get("user.dir")+"/AAA/Clients.txt");
                InputStreamReader ipsr=new InputStreamReader(ips);
                BufferedReader br=new BufferedReader(ipsr);

              String line;

              // Create a new object for a new Client
              MN_Data_Mobile newMNinformatation;

           // Turn still the end of the documents (the buffer in facts)
                while ((line=br.readLine())!=null){
                     // Detect if a line is empty to jump an exception
                     if (line.isEmpty() != true){
                          // Discard the # comentary ligne
                          if (line.charAt(0) != '#'){
                               newMNinformatation = new MN_Data_Mobile(line);
                               // compare the twice name
                               if (newMNinformatation.getIdClient().compareTo(idClient) == 0)
                                    // Give the new information client to the PMIPClient
                                    if (!PMIP_Client.existingThread(newMNinformatation.getMAC()))
                                              PMIP_Client.newPMIPClient(newMNinformatation);
```

```
                              }
                        }
                  }
                  br.close();
            }
            catch (Exception e){
                  System.out.println(e.toString());
            }
      }

      /**
       * Method that run the SnifferUDP
       */
      public void run() {
            SnifferUDP();
      }
}
```

```
import java.util.StringTokenizer;

/**
 * This class create an object that contain all the information for each PMIP Client Needed
 * for the establishement of mobility with the HA
 * @author Hercher Yann
 * @version 1   29/10/2008
 */
public class MN_Data_Mobile {
     private String idClient;
     private String MAC;                  //Media Access Control (MAC address)
     private int SPI;                     //Security Parameter Index
     private String ShareSecret;               //Password (ShareSecret)
     private int TunnelLifeTime;               //TunnelLifeTime
     private byte[] MNHomeIPAddressV4 = new byte[4];      //IP Address of the MN
     private byte[] HAIPAddressV4 = new byte[4];               //IP Adress of the Home Agent
     private byte reverseTunelling;       //Byte that set the reverseTunelling mode
     private byte GreEncapsulate;              //Byte that set the tunnel encapsulation GRE
     private byte MinimalEncapsulation;   //Byte that set the minimal encapsulation
     private byte BroadcastDatagrams;          //Byte that set the BroadcastDatagrames
     private byte SimultaneousBindings;   //Byte that set the Simultaneous bindings
     private StringTokenizer st, ip;

     /**
     * Constructor that actualy receive a String from a text information file that contain
     * the information for the client mobility. this string is tokenized to bring the client
     * information
     * @param startSignal
     */
     public MN_Data_Mobile(String lineToTokenize){
          st = new StringTokenizer(lineToTokenize," ");

          idClient = st.nextToken();
          MAC = st.nextToken();
          SPI = Integer.parseInt(st.nextToken());
          ShareSecret = st.nextToken();
```

```
        TunnelLifeTime = Integer.parseInt(st.nextToken());

        ip = new StringTokenizer(st.nextToken(),".");
        MNHomeIPAddressV4[0] = (byte)Integer.parseInt(ip.nextToken());
        MNHomeIPAddressV4[1] = (byte)Integer.parseInt(ip.nextToken());
        MNHomeIPAddressV4[2] = (byte)Integer.parseInt(ip.nextToken());
        MNHomeIPAddressV4[3] = (byte)Integer.parseInt(ip.nextToken());

        ip = new StringTokenizer(st.nextToken(),".");
        HAIPAddressV4[0] = (byte)Integer.parseInt(ip.nextToken());
        HAIPAddressV4[1] = (byte)Integer.parseInt(ip.nextToken());
        HAIPAddressV4[2] = (byte)Integer.parseInt(ip.nextToken());
        HAIPAddressV4[3] = (byte)Integer.parseInt(ip.nextToken());

        reverseTunelling = getBoolean(st.nextToken());
        GreEncapsulate = getBoolean(st.nextToken());
        MinimalEncapsulation = getBoolean(st.nextToken());
        BroadcastDatagrams = getBoolean(st.nextToken());
        SimultaneousBindings = getBoolean(st.nextToken());
    }
    /*************************************************************
    * get* accesser fuctions return value of private data members*
    *************************************************************/

   /** Convert a String Value into boolean */
    private byte getBoolean(String Boolean){
        if(Boolean.compareToIgnoreCase("true")==0){
            return 1;
        }else{
            return 0;
        }
    }

    /** Get the idClient */
    public String getIdClient(){
        return idClient;
```

```
}

/** Get the MAC */
public String getMAC(){
     return MAC;
}

/** Get the SPI */
public int getSPI(){
     return SPI;
}

/** Get the ShareSecret */
public String getShareSecret(){
     return ShareSecret;
}

/** Get the TunnelLifeTime */
public int getTunnelLifeTime(){
     return TunnelLifeTime;
}

public byte[] getMNHomeIPAddressV4(){
     return MNHomeIPAddressV4;
}

/** Get the HAIPAddressV4 */
public byte[] getHAIPAddressV4(){
     return HAIPAddressV4;
}

/** Get the reverseTunelling */
public byte getreverseTunelling(){
     return reverseTunelling;
}
```

```
	/** Get the GreEncapsulate */
	public byte getGreEncapsulate(){
		return GreEncapsulate;
	}

	/** Get the MinimalEncapsulation */
	public byte getMinimalEncapsulation(){
		return MinimalEncapsulation;
	}

	/** Get the BroadcastDatagrams */
	public byte getBroadcastDatagrams(){
		return BroadcastDatagrams;
	}

	/** Get the SimultaneousBindings */
	public byte getSimultaneousBindings(){
		return SimultaneousBindings;
	}
}
```

Rapport de travail hebdomadaire

Semaine 1 (15 au 19 septembre)
- Installation et paramétrage du réseau *MIP* avec *OpenDynamics*
- Configuration du client *Linux* (*Interface* et *WPA_Supplicant*)
- Test de mobilité entre deux réseaux et capture *wireshark*
- Lecture de la *RFC3344* (*MIP*)

Semaine 2 (23 au 26 septembre)
- Recherche sur la possibilité de reprendre des éléments existants d'*OpenDynamics*
- Le réseau nécessite un changement d'infrastructure et la configuration du *MIP* doit être changé pour que la désencapsulation se fasse au niveau du *FA*.
- Premier problème de routing.
- Début du développement du service *DHCP* (3 jours de retard sur le planning)

Semaine 3 (29 au 03 octobre)
- Première itération du *DHCP* réussie le client reçoit une adresse *IP*
- Problème pour développer un service *DHCP* multi-clients simultanés.

Semaine 4 (06 au 10 octobre)
- Fin de l'iteration Gantt du développement du service *DHCP* en fin de semaine (délai non respecté mais 7 jours de retard)
- Début de l'étude de développement du *PMIP Client*

Semaine 5 (13 au 17 octobre)
- Développement du *package jPMIPClient*
- Recherche pour créer une signature en *Hmac-MD5* en *Java* pour l'extension d'identification
- Problème *TTL* dans les messages *IP* (*error 76*)

Semaine 6 (27 au 31 octobre)
- Rapport intermédiaire

Semaine 7 (03 au 07 novembre)
- Résolution du problème du champ identification contenant le *Timestamp* qui diffère du *HA*
- Recherche pour résoudre se problème de *TTL* et d'écoute sur un interface définit.
- Etude du protocole *Diameter* (2 semaines de retard)

Semaine 8 (10 au 14 novembre)
- Etude de différents services *Diameter libre*.
- Impossible d'installer *OpenDiameter* sur *Linux*
- Problème de *Socket Java* avec le *TTL* et l'interface
- Décision pour trouver une solution autre que le *AAA* pour continuer à avancer dans le projet (plus de 2 semaines de retard)

Semaine 9 (17 au 21 novembre)

- Détournement du système *AAA* permettant au moins l'utilisation du protocole en local sur un seul *PMA*
- Renouvellement du tunnel automatiquement
- Test avec le *PMIP Client* sur l'ordinateur du *FA* (problème de discussion en *localhost*)

Semaine 10 (24 au 28 septembre)

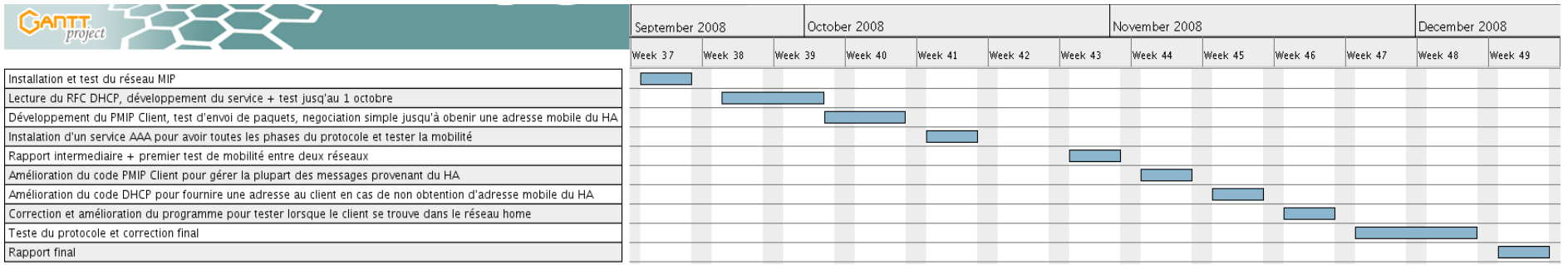
- Installation d'un nouveau *PC* pour contourner le problème du *local host*.
- Problème avec le service *DHCP* qui ne peut plus envoyer de message à l'adresse 255.255.255.255
- Instalation d'un autre *PC* dans le deuxième réseau afin de tester une mobilité basique (4 semaines de retard sur le planning Gantt)

Semaine 11 (01 au 05 septembre)

- Recherche pour l'utilisation d'un service *AAA* par *Radius*
- Utilisation d'un serveur *Radius* d'un autre diplômant
- Test de mobilité

Semaine 12 (05 au 12 septembre)

Rapport final


GANTT project
September 2008
October 2008
November 2008
December 2008
Week 37
Week 38
Week 39
Week 40
Week 41
Week 42
Week 43
Week 44
Week 45
Week 46
Week 47
Week 48
Week 49
Installation et test du réseau MIP
Lecture du RFC DHCP, développement du service + test jusq'au 1 octobre
Développement du PMIP Client, test d'envoi de paquets, negociation simple jusqu'à obenir une adresse mobile du HA
Instalation d'un service AAA pour avoir toutes les phases du protocole et tester la mobilité
Rapport intermediaire + premier test de mobilité entre deux réseaux
Amélioration du code PMIP Client pour gérer la plupart des messages provenant du HA
Amélioration du code DHCP pour fournire une adresse au client en cas de non obtention d'adresse mobile du HA
Correction et amélioration du programme pour tester lorsque le client se trouve dans le réseau home
Teste du protocole et correction final
Rapport final