UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté des Sciences Appliquées
Département d'Ingénierie Informatique

Développement d'extensions au Kernel Linux pour supporter le multihoming IPv6

Promoteur : Prof. O. **Bonaventure**

Mémoire présenté en vue
de l'obtention du grade
d'Ingénieur Civil en Informatique
par **Sébastien Barré**

Louvain-la-Neuve
Année académique 2005-2006

Table des matières

Introduction

A cause principalement du manque d'adresses IPv4 qui se fait sentir de plus en plus dans l'Internet actuel, une nouvelle version du protocole a été conçue : IPv6. Allant au-delà de la simple résolution du manque d'adresses (par l'usage d'adresses plus longues, sur 128 bits), cette nouvelle version a permis de repenser bien des aspects du protocole, afin d'en améliorer la sécurité et l'efficacité.

Le multihoming consiste en une connexion d'un équipement ou d'un réseau à plusieurs fournisseurs d'accès, généralement pour obtenir une meilleure tolérance aux pannes ou répartir le trafic entre les différents fournisseurs. Actuellement, cela est le plus souvent réalisé en annonçant le préfixe réseau à tous les fournisseurs par BGP. Mais cette technique suppose de disposer d'un numéro d'AS.

Avec l'arrivée d'IPv6 il deviendra normal qu'un équipement dispose de plusieurs adresses. Cela suggère une nouvelle possibilité de multihoming, dans laquelle chaque équipement réseau détecterait les pannes liées à ses propres connexions, et basculerait sur une nouvelle adresse en cas de panne de l'adresse utilisée. Un tel basculement possède cependant l'inconvénient de rompre toutes les connexions en cours. L'idée est donc apparue progressivement de séparer les deux sémantiques actuellement associées à l'adresse IP :

- Une adresse IP est un *localisateur*, dans le sens où elle permet une *localisation* du correspondant, en faisant usage des tables de routages sur le chemin parcouru par les paquets ;
- Une adresse IP est aussi un *identifiant*, c'est-à-dire qu'elle sert à la couche transport pour associer un paquet entrant à une connexion établie.

Si l'on sépare les concepts d'identifiant et de localisateur, il devient possible d'envisager n'importe quelle structure dans le rôle de l'identifiant. Les travaux de l'IETF ont finalement convergé vers une solution, `shim6`, où les identifiants sont des localisateurs, mais où les premiers peuvent être différents des seconds. En particulier, dans la solution `shim6`, une connexion démarre en utilisant des identifiants et localisateurs égaux, et change les localisateurs en gardant constants les identifiants lorsqu'une panne se produit. Cela nécessite l'insertion d'une couche supplémentaire dans la pile TCP/IP, qui se chargerait d'effectuer la traduction entre identifiants et localisateurs. *L'intérêt principal de cette solution est donc de maintenir les connexions établies en couche transport, même en cas de changement d'adresse (localisateur).*

Contributions de ce mémoire : `Shim6` est une solution nouvelle, encore en développement. A ce jour, aucune implémentation n'a encore été réalisée, et la définition du protocole elle-même a beaucoup évolué au cours de cette année. Récemment, un projet a vu le jour (*http ://unfix.org/-projects/shibby/*), mais il n'y a pas encore de code disponible.

Dans ce mémoire, nous proposons un prototype d'implémentation de `shim6` pour Linux. Celui-ci a pour objectifs d'identifier les difficultés liées à une telle implémentation, et de proposer des solutions pour les résoudre. Le résultat final est une version fonctionnelle de `shim6`, dans laquelle cependant certaines parties, telle la sécurité, ont été laissées comme 'future work' dans le but de garder pour vraie cible la question de la gestion du multihoming. Par ailleurs, une attention particulière a été

accordée à la robustesse du code, car il s'agit d'un aspect fondamental lorsque l'on modifie le noyau d'un système d'exploitation. Pour cette raison les questions liées au parallélisme (systèmes multi-processeurs, préemption) seront discutées de manière étendue dans ce mémoire, ainsi que l'approche adoptée pour les gérer dans le prototype.

Structure des chapitres : Loin de vouloir fournir une explication détaillée des protocoles shim6 et REAP (qui s'avérerait trop volumineuse), ce mémoire s'attache à donner un aperçu global de ce qui a été défini dans les drafts (chapitres 1,2), afin de fournir au lecteur une vue suffisante sur l'état des travaux (de l'IETF) pour étudier les aspects d'implémentation (chapitres 3,4).

Ce mémoire se divise en quatre chapitres. Alors que les deux premiers sont descriptifs et fournissent respectivement un aperçu de `shim6` et de l'architecture réseau dans le Kernel Linux, les deux suivants s'attachent à une étude de l'implémentation.

Le chapitre 1 est une introduction à IPv6 et au multihoming, et se termine par une description conceptuelle de `shim6`, tel qu'il était défini au début de ce mémoire. Il avait été décidé de réaliser le mémoire sur base de l'état des drafts à ce moment, afin de ne pas passer un temps inutile à suivre la cible mouvante des travaux de l'IETF. Nous voyons également dans ce chapitre certains aspects d'IPv6, telle l'autoconfiguration d'adresses, qui joueront un rôle important dans l'implémentation réalisée. Les aspects de sécurité de shim6 sont également décrits, bien qu'ils ne soient pas implémentés (en revanche, l'implémentation prévoit que les fonctions de sécurité soient insérées facilement par après).

Le chapitre 2 étudie les techniques d'implémentation d'usage courant dans le noyau Linux. Les questions de gestion du parallélisme suscitent une discussion étendue, car cette gestion est très différente de ce que l'on fait pour des applications normales (ces dernières bénéficiant justement du soutien du noyau). Y sont également décrits quelques outils disponibles dans les librairies du noyau, et qui se sont avérées utiles pour l'implémentation de notre prototype. Finalement, on trouve une étude de l'implémentation d'IPv6, qui pose les bases nécessaires à bien comprendre les choix faits par la suite, quant à l'insertion de la nouvelle couche `shim6` dans la pile TCP/IP.

Le chapitre 3 est une description plus précise de shim6, qui insiste sur les aspects implémentés du mécanisme, et décrit au fur et à mesure l'architecture de notre implémentation. Contrairement au chapitre 1, nous y prenons pour base le document [32] qui, bien que paru initialement en septembre 2005, a subi d'importants remaniements qui ont conduit à une version jugée suffisamment stable pour en faire une implémentation. Néanmoins, cette définition du protocole est encore en mouvement, comme en témoigne la parution d'une cinquième version du draft au moment d'écrire ces lignes.

Le chapitre 4 est consacré à l'étude d'un protocole de détection de joignabilité, et exploration de paires d'adresses en cas de panne. En effet [32] ne définit que le mécanisme de `shim6` et le protocole permettant d'échanger les localisateurs d'un correspondant, laissant la gestion de l'exploration à un autre protocole, REAP (REAchability Protocol), qui constitue un sous-protocole de `shim6`, défini en décembre 2005[7]. Malheureusement ce document n'est pas encore complet, et nécessite quelques révisions. En particulier, le document ne définit pas d'ordre précis d'exploration des adresses. Le chapitre 4 décrit donc ce qui est défini dans le document, et les solutions provisoires pour lesquelles nous avons opté afin d'obtenir un prototype fonctionnel, en dépit du fait que la définition de REAP soit incomplète à ce jour.

Chapitre 1

Le multihoming dans IPv6 : shim6

La pratique du multihoming consiste à connecter un ordinateur ou un réseau complet à plusieurs fournisseurs de service (ISPs, ou Internet Service Providers). Actuellement, cela est surtout réalisé par les grands réseaux ou les fournisseurs de service eux-mêmes. Les motivations sont multiples :

- Une connectivité permanente à Internet peut être primordiale (présence de gros serveurs, ou enjeux économiques). Le fait de posséder plusieurs fournisseurs permet de se protéger des pannes de lien, mais aussi des problèmes pouvant se produire chez le fournisseur lui-même, ou en amont de celui-ci. Dans l'exemple présenté à la figure 1.1, le réseau `123/8` peut utiliser indifféremment les chemins `c1` ou `c2` (en traits discontinus) pour atteindre `234/8`. Une application typique de ce schéma serait que le réseau `123/8` utilise `c1` comme lien par défaut pour envoyer son trafic. Dans ce cas, `c1` est habituellement un chemin à large bande passante, et `c2` est alors considéré comme un lien de secours, meilleur marché, sur lequel `123/8` peut basculer en cas de panne de sa route habituelle. Dans les situations réelles, les chemins ne diffèrent pas toujours de bout en bout comme ici, cependant il est intéressant (en termes de résistance aux pannes) de chercher à obtenir des chemins aussi différents que possible.
- Le réseau `123/8` peut aussi envoyer une partie de son trafic sur chacun des chemins `c1` et `c2`. Cette option peut être choisie pour des raisons économiques (contrats avec les fournisseurs), pour répartir simplement la charge, ou pour fournir des connexions de qualités différentes (en termes de débit/délai).

Il faut noter que le multihoming est un concept différent de ce qui est parfois appelé *multi-connecting*. Cette autre solution consiste à s'attacher à *un seul* fournisseur, mais avec plusieurs liens. C'est la solution choisie par `234/8` (fig. 1.1). Du point de vue de la tolérance aux pannes, `234/8` est seulement protégé des pannes de *lien* avec `ISP3`. Mais si `ISP3` venait à avoir un problème, `234/8` serait aussi hors service.

1.1 Le multihoming actuel : techniques utilisées

Nous parcourons ici les principales solutions de multihoming utilisées actuellement avec IPv4, à savoir l'emploi de BGP ou d'une passerelle NAT. Une description comparative plus détaillée peut être trouvée dans [18].

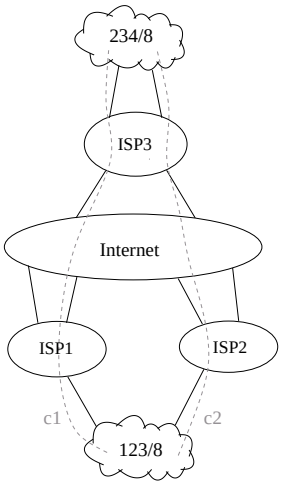


FIG. 1.1 – Exemple de multihoming et multi-connecting

1.1.1 Le multihoming avec BGP

Une solution très répandue est d'avoir recours à BGP. Ce protocole s'exécute sur les routeurs *frontières* d'un ensemble de réseaux placés sous une même administration (appelés usuellement *systèmes autonomes* ou ASs, Autonomous Systems). Un AS peut tirer profit du multihoming simplement en se connectant à plus d'un ISP. Dans ce cas, il peut annoncer ses routes à chacun des ISPs, qui à leur tour vont les propager, de sorte que l'Internet global dispose de plusieurs chemins pour accéder au même AS (fig. 1.2). Tout comme on peut annoncer des routes, on peut aussi en retirer, c'est ce que fait `ISP2` lorsqu'il s'aperçoit qu'il ne peut plus joindre son client `123/8` (message *withdraw*, fig. 1.2). Notons qu'il pourra par contre le joindre en passant par AS40, parce que son fournisseur lui transmettra sa connaissance de la route AS40 (message en gris à la figure 1.2).

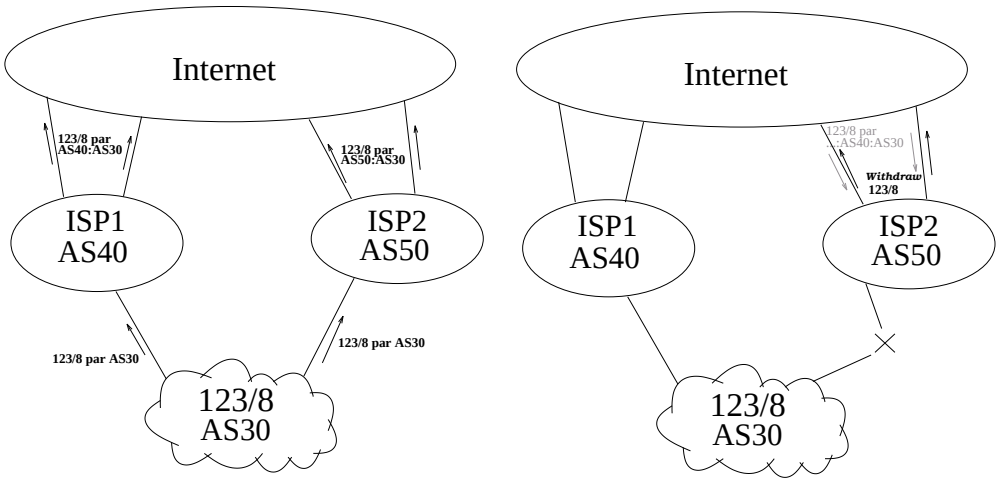
Cette manière de faire est très bien adaptée au cas des ISPs de transit, et ne semble pas devoir être modifiée. Cependant pour les plus petits réseaux, plusieurs inconvénients se présentent :

- Tout réseau désirant pratiquer le multihoming doit nécessairement posséder son propre numéro d'AS.
- BGP est un protocole avec état : il maintient des tables, reprenant toutes les informations reçues de l'extérieur. A cause de la taille actuelle d'Internet, ces tables deviennent très chargées dans les routeurs des ISPs de transit. La propagation de chemins différents par BGP nécessite une entrée supplémentaire pour chacun.

Pour les ISP de transit, on peut tolérer l'ajout d'entrées supplémentaires dans les tables, parce qu'ils devraient normalement rester peu nombreux. Par contre le nombre de petits ISPs augmente très rapidement. De ce fait s'ils utilisaient cette technique, les tables BGP du coeur de l'Internet deviendraient vite ingérables.

1.1.2 Emploi des NATs

Il est possible également de pratiquer le multihoming en faisant usage d'une passerelle NAT (Network Address Translation). Dans ce cas, il n'est pas nécessaire de disposer d'un numéro d'AS, et la solution est donc accessible à de plus petites organisations. Par contre il faut demander à chacun de ses ISPs un ensemble d'adresses PA (Provider Aggregatable), c'est-à-dire un sous-ensemble des adresses assignées à l'ISP. Ainsi l'entreprise “ABC" (fig. 1.3) se voit assigner le préfixe `40.0.123/24` de



Annonce d'une route Retrait d'une route

FIG. 1.2 – Utilisation de BGP pour le multihoming

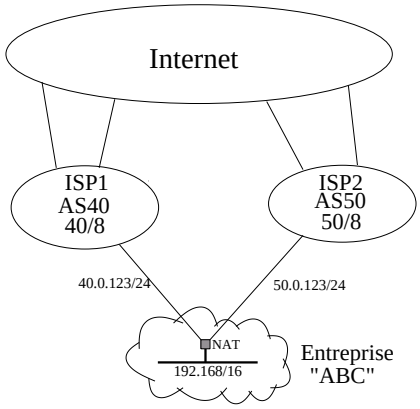


FIG. 1.3 – Multihoming avec NAT

la part de ISP1, et 50.0.123/24 de ISP2. Le réseau interne de l'entreprise utilise quant à lui uniquement des adresses locales (192.168/16). Lorsqu'une machine souhaite communiquer avec l'extérieur, son adresse locale est traduite par la passerelle NAT en une adresse publique appartenant à l'un des fournisseurs et envoyée sur le lien correspondant. La traduction inverse est réalisée lorsqu'un paquet est reçu.

Contrairement à la solution reposant sur BGP, celle-ci ne provoque pas d'accroissement de taille des tables BGP, parce que seuls sont annoncés sur Internet les préfixes de chacun des fournisseurs.

Tout ceci semble assez élégant, cependant un inconvénient majeur est lié à toutes les pratiques liées à l'usage d'une passerelle NAT : le contenu des messages associé à la couche application (payload) n'est pas modifié. Cela peut poser problème si l'adresse IP est manipulée au niveau de l'application (l'un ne connaîtra que l'adresse privée, l'autre que l'adresse publique). Certaines applications peuvent cesser de fonctionner, ou nécessiter l'implémentation d'un ALG (Application Level Gateway) qui se charge de faire les adaptations nécessaires, en collaboration avec la passerelle NAT.

Un autre inconvénient de NAT est qu'il est impossible de réaliser un changement d'adresse sans rompre toutes les connexions TCP en cours, parce que les connexions TCP sont identifiées par les quadruplets <port_src,port_dest,ip_src,ip_dest>. Notons que, dans le cas de l'usage de NAT, ce n'est pas le site multi-hébergé[1] qui change d'identifiant, puisque lui ne voit que son adresse locale tout le temps. Par contre son correspondant verra, lors d'un changement d'adresse, un changement dans le champ ip_src, et la connexion sera perdue.

Ce dernier point est en particulier l'un des problèmes que résout shim6, qui sera étudié plus en détail à la section 1.3. Mais la technologie shim6 étant destinée à une utilisation avec IPv6, nous présenterons d'abord les caractéristiques essentielles de cette nouvelle version du protocole IP.

1.2 Le nouveau protocole IPv6

IPv6 est la nouvelle version de l'actuel IPv4 (Internet Protocol, version 4). Ce protocole, en cours de développement depuis 1998, a pour principal objectif de parer au problème d'allocation d'adresses IP. En effet, dans IPv4, une adresse est codée sur 32 bits, permettant (théoriquement) l'allocation de 2^{32} adresses. Cela avait un goût d'infini à l'époque du déploiement d'IPv4. Mais le développement très rapide d'Internet a eu pour conséquence que l'on doit maintenant recourir à des astuces telles que NAT ou DHCP pour pouvoir recycler des adresses. Mais ces moyens ne sont que des systèmes d'appoint, permettant seulement de retarder le moment où toutes les adresses seront allouées.

Donc pour résoudre ce problème les adresses IPv6 sont codées sur 128 bits. Il s'agit là de la principale nouveauté. Cependant ce changement a été l'occasion de repenser complètement le protocole, et de faire beaucoup de recherche, afin d'améliorer également la fiabilité, l'efficacité et la sécurité du protocole. Un grand effort a aussi été fourni pour faciliter l'utilisation et la configuration du protocole (connexion "plug & play"). Les fondements d'IPv6 sont définis dans [19] et [25].

Dans cette section, nous décrirons seulement quelques aspects d'IP version 6, qui sont importants dans le cadre du multihoming. Le nouveau mécanisme d'adresses sera d'abord présenté. Ensuite nous nous intéresserons au fonctionnement du DNS. Finalement, le système d'autoconfiguration d'adresses et les aspects de sécurité qui y sont liés seront étudiés.

[1] *site multi-hébergé* est la traduction que nous avons choisie pour le terme anglais *multihomed site*

1.2.1 Mécanisme d'adresses IPv6

Du fait que les adresses IPv6 sont codées sur 128 bits, il a fallu trouver une autre notation que celle d'IPv4, ainsi on utilise l'hexadécimal. Une adresse IPv6 complète est constituée de 8 groupes de 4 chiffres hexadécimaux, séparés par le caractère " :", comme illustré sur l'exemple suivant :

```
1234:0000:0000:0000:5678:9ABC:0000:0023
```

Comme c'est très long, une notation courte a été définie :

- Un groupement, ou plusieurs groupements consécutifs de 4 zéros peuvent être ignorés. Pour l'exemple ci-dessus, on peut donc aussi être écrite `1234::5678:9ABC:0000:0023`. Cependant ce raccourci ne peut être utilisé qu'une fois dans l'adresse, donc il serait incorrect d'utiliser la notation `1234::5678:9ABC::0023`.
- Dans chaque champ de l'adresse, les zéros de tête peuvent être omis. Ainsi on peut écrire : `1234::5678:9ABC:0:23`.

La notation des préfixes réseau reste identique au fonctionnement dans IPv4 (ex. : `1234::/16`).
Les adresses réservées dans IPv6 sont les suivantes [25] :

- L'adresse `::` est en fait une absence d'adresse. Elle ne peut être utilisée que dans le champ source, et sert par exemple lors de la configuration automatique d'adresse.
- `::1` est l'adresse loopback (comme l'était `127.0.0.1` dans IPv4).
- *Adresses Multicast* : `FF00::/8`
- Les adresses `::/96` (96 zéros suivis d'une adresse sur 32 bits) sont utilisées pour permettre le passage de paquets IPv6 dans une infrastructure IPv4 à travers un tunnel. Pour cela les noeuds extrémités du tunnel reçoivent une adresse de ce type. L'intérêt est qu'il suffit d'ôter les 96 zéros de tête pour obtenir une adresse IPv4 valide. Ces adresses sont appelées *IPv4-compatible IPv6 adresses* et leur utilisation est détaillée dans [23].
- Le préfixe `::FFFF:0:0/96` est lui aussi réservé pour un usage avec des adresses IPv4, mais ici il s'agit de pouvoir communiquer avec des hôtes dépourvus du support d'IPv6. Ces adresses sont appelées *IPv4-mapped IPv6 addresses*.
- *link-local addresses* : elles servent principalement à la configuration automatique d'adresses et ne peuvent servir qu'à l'intérieur d'un réseau directement connecté. Le préfixe réservé à cet usage est `FE80::/64`.
- *site-local addresses* : comme le nom l'indique, ce sont les adresses locales à un site : cela signifie qu'elles peuvent servir à l'interconnexion de machines de manière locale, mais ne peuvent être annoncées sur l'Internet global. Le préfixe est `FEC0::/48`

Les autres adresses sont du type *global unicast* (ou anycast, ces dernières possédant la même forme). Une adresse IPv6 *global unicast* est constituée de trois champs : le premier est un préfixe de routage public, codé sur 48 bits. Ensuite vient le préfixe de sous-réseau, sur 16 bits, et finalement un identifiant d'interface codé sur 64 bits au format EUI-64 modifié[25]. Ce nombre référence une interface plutôt qu'une machine. Ainsi chaque interface possède une adresse différente. Une interface peut par ailleurs posséder plusieurs adresses (par exemple une *link-local* et une *global unicast*).

L'identifiant sera en général calculé à partir de l'adresse de couche 2, par exemple l'adresse MAC. Comme cette adresse est codée sur 48 bits, un mécanisme est défini pour la convertir en une adresse EUI-64[17]. Parfois il n'est pas possible de l'extraire, ou on ne souhaite pas le faire pour des raisons de confidentialité. Dans ce cas on peut définir un EUI-64 dont l'unicité est seulement garantie localement, la seule contrainte étant que chaque adresse IPv6 doit être unique. Donc il faut seulement s'assurer qu'il soit impossible de rencontrer deux identifiants EUI-64 identiques à l'intérieur d'un même sous-réseau (le préfixe de 64 bits assure l'unicité globale).

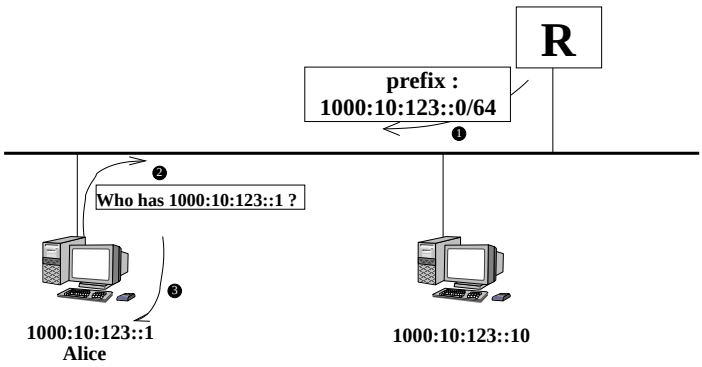


FIG. 1.4 – Attribution d'une adresse à un équipement

Deux bits d'un identifiant EUI-64 ont une signification particulière :
- Le bit 'u' (Universal/local), lorsqu'il vaut 1, indique que l'identifiant est unique globalement, sinon son unicité n'est garantie que localement.
- le bit 'g' (Individual/group) indique s'il est à 1 que l'identifiant correspond à un groupe. Cela revient à dire que l'adresse formée à partir d'un tel identifiant est de type multicast.

Ces bits sont respectivement les septième et huitième bits en partant de la gauche de l'identifiant.

L'existence de tous ces types d'adresses montre clairement qu'une seule machine va souvent disposer d'adresses multiples. La mise en oeuvre du multihoming ajoutera encore d'autres adresses à cette liste (plusieurs *global-unicast* pour joindre une seule machine). Au moment de démarrer une nouvelle communication, le problème du choix d'une paire d'adresses se pose donc. Un algorithme qui établit un ordre de préférence entre les couples d'adresses à utiliser est proposé dans le RFC3484 [20].

1.2.2 DNS avec IPv6

Comme dans IPv4, le DNS sera utilisé pour résoudre un nom d'hôte en une adresse IPv6. Afin de pouvoir différentier les adresses IPv4 des adresses IPv6, un nouveau type de champ DNS est défini : `AAAA`. Par ailleurs, il est possible de faire une requête DNS inverse, c'est-à-dire obtenir un nom de machine à partir de son adresse IP. Cela se fait en s'adressant au domaine `IP6.ARPA`, et en écrivant l'adresse que l'on souhaite résoudre à l'envers, en séparant chaque nombre hexadécimal par un point. On peut par exemple obtenir le nom d'hôte correspondant à l'adresse `1234::5678:9ABC:0:23` en résolvant le nom DNS ci-dessous :

```
3.2.0.0.0.0.0.0.C.B.A.9.8.7.6.5.0.0.0.0.0.0.0.0.0.0.0.0.0.4.3.2.1.IP6.ARPA
```

Les extensions du DNS pour le support d'IPv6 sont décrites dans [37].

1.2.3 Autoconfiguration d'adresses et Neighbor Discovery

Le protocole IPv6 a été conçu dans un souci de permettre un fonctionnement immédiat des équipements au moment de leur connexion au réseau, avec un minimum (ou pas du tout) de configuration nécessaire.

Pour éviter cette configuration manuelle, un mécanisme a été imaginé, pour permettre à une interface nouvellement connectée :
- De découvrir les routeurs connectés sur le même lien, qui permettront d'accéder à Internet. Si plusieurs routeurs sont détectés, l'un d'eux est choisi comme passerelle par défaut pour envoyer le trafic.

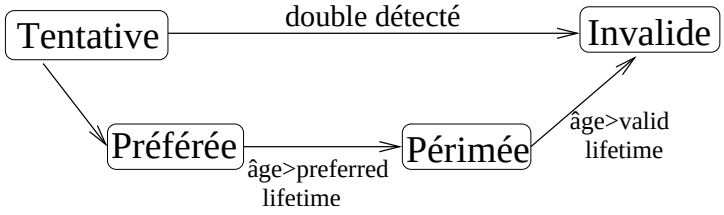


FIG. 1.5 – Etats successifs d'une adresse IPv6

– De s'attribuer des adresses automatiquement, par concaténation d'un préfixe (les préfixes sont annoncés par les routeurs) et d'un identifiant d'interface de 64 bits.

L'autoconfiguration d'adresses sans état est décrite dans [38] (une configuration avec état reste possible grâce au protocole DHCPv6 [21]). Le protocole Neighbor Discovery est défini dans [29].

Un exemple d'attribution d'une adresse est illustré à la figure 1.4, où Alice vient de se connecter à un réseau. Les étapes suivies par l'équipement d'Alice sont les suivantes :

– Le routeur `R` annonce périodiquement l'ensemble des préfixes qui peuvent être utilisés sur le lien. Dans ce cas il s'agit du préfixe `1000:10:123::0/64`. Alice peut provoquer l'envoi de ce message (*Router Advertisement*) par l'envoi d'un *Router Solicitation* destiné à l'adresse multicast `all routers` (`ff02::2`).
– Alice génère un identifiant d'interface, destiné à remplir les 64 bits de poid faible de son adresse. Plusieurs méthodes existent pour générer cet identifiant, voir par exemple [17, 10, 28].
– Puisque l'identifiant d'interface est généré localement, il faut maintenant vérifier son unicité (sur le lien uniquement, puisque le préfixe annoncé par le routeur assure l'unicité dans l'Internet global).

Cette dernière étape (numéro 2 dans la figure 1.4), est prise en charge par l'algorithme DAD (Duplicate Address Detection [38]) : Alice demande si quelqu'un possède déjà l'adresse qu'elle souhaite prendre. Comme personne ne répond, elle l'enregistre et peut l'utiliser.

Durée de vie d'une adresse : [38] définit une durée de vie pour les adresses IPv6, durant laquelle une adresse donnée peut être utilisée par une interface. Le but est de permettre une renumérotation automatique d'un site, en cas de changement de préfixe par exemple. Une adresse ne peut cependant pas être supprimée abruptement, car on perdrait les connexions en cours. La solution est de définir deux durées de vie. Le principe est illustré à la figure 1.5. Pendant une durée appelée *preferred lifetime*, l'adresse est valide et peut être utilisée sans restriction. Ce temps *preferred lifetime* est plus court que le *valid lifetime*, au bout duquel il faudra supprimer complètement l'adresse. Entre les deux elle est 'périmée'. Dans cette situation, son emploi est encore permis, mais les nouvelles connexions ne devraient plus y faire appel.

L'état 'tentative' de la figure 1.5 correspond à la période durant laquelle l'algorithme de détection d'adresses dupliquées s'exécute. Durant cette période l'adresse ne peut être utilisée.

Sécurisation de Neighbor Discovery : Le principe énoncé ci-dessus fonctionne bien si l'on a confiance dans les personnes qui se connectent au réseau. C'est de moins en moins le cas actuellement, il suffit pour s'en rendre compte de penser au développement que connaissent les réseaux sans fil dans les aéroports, les stations d'autoroute, ...

Si l'on reprend la figure 1.4, on observe qu'un utilisateur malintentionné peut intervenir dans l'algorithme de détection d'adresses dupliquées, en répondant à toutes les requêtes DAD, prétendant

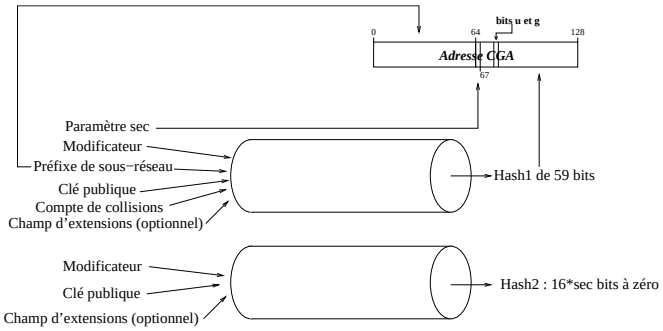


FIG. 1.6 – Génération d'une adresse CGA

qu'il possède n'importe quelle adresse. Ainsi chaque adresse *tentative* est directement invalidée, et Alice est incapable de se connecter. D'autres examples d'attaques possible de Neighbor Discovery peuvent être trouvés dans [5].

La solution évidente au problème est de signer les messages du protocole Neighbor Discovery. C'est ce qui est proposé dans [8], qui définit SEND (SEcure Neighbor Discovery).

Les messages incluent désormais une signature, ainsi que la clé publique permettant de vérifier le message. Si l'on reprend l'exemple ci-dessus, où Alice tente d'acqérir l'adresse A, et si l'on suppose que Bob (dont la machine n'est pas représentée ur le dessin) possède vraiment cette adresse, celui-ci devra envoyer un message NA (Neighbor Advertisement) indiquant qu'il la détient déjà. Son message est signé et la clé publique est jointe. On observe que la clé publique sert ici à prouver, non pas que l'émetteur est bien Bob, mais plutôt que l'émetteur est bien le détenteur de l'adresse. C'est pourquoi, plutôt que de se baser sur un mécanisme de certificats, on liera la clé publique de Bob à son adresse, définissant ainsi un nouveau type d'adresse. Ce sont les CGAs, qui font l'ojet de la section suivante.

1.2.4 Cryptographically Generated Addresses

Les adresses IPv6 CGA (Cryptographically Generated Addresses) se distinguent des autres uniquement par le champ *identifiant d'interface*, sur 64 bits, de l'adresse. Comme indiqué à la section 1.2.1, cet identifiant d'interface peut être calculé à partir de l'adresse de couche 2 (MAC par exemple) ou généré par un autre procédé.

Ici, le but est de lier une adresse et une clé publique. Pour cela, un équipement qui souhaite utiliser une adresse va d'abord générer un couple <clé publique,clé privée>. La clé publique servira ensuite d'entrée au calcul d'un hash, qui fournit un résultat sur 59 bits. Ces 59 bits, associés aux bits u et g, ainsi qu'au trois bits de codage du paramètre de sécurité (décrit plus loin) permettent de former les 64 bits de l'identifiant d'interface (voir figure 1.6). L'association de ces 64 bits aux 64 bits de préfixe donne l'adresse finale utilisable.

Les bases de CGA sont posées. Maintenant nous constatons qu'il est dangereux de limiter la taille du hash à 59 bits : une attaque de type brute force pourrait casser le hash en $\mathcal{O}(2^{59})$. C'est là qu'intervient le paramètre sec : selon le niveau de sécurité désiré, le paramètre sec peut prendre une valeur entre 0 et 7, de manière à allonger *artificiellement* la longueur du hash de $sec * 16$ bits. Le principe, puisque ne sont inclus dans l'adresse que 59 bits, est de contraindre l'un de ses paramètres de génération (le modificateur, voir fig. 1.6), de telle manière que le calcul d'un hash sur ce modificateur donne la valeur 0. Malheureusement celui qui génère l'adresse doit, pour ce faire, vérifier cette contrainte lui aussi d'une manière brute, soit en $\mathcal{O}(2^{16*sec})$: il s'agit d'incrémenter le modificateur et recalculer le hash, hash2 sur la figure, jusqu'à ce qu'il possède ses 16*sec bits de poid fort à zéro.

Ce coût supplémentaire reste acceptable puisque le calcul n'est à effectuer qu'une fois par adresse, et *ne doit pas être fait par celui qui vérifie*, car il reçoit le modificateur parmi les paramètres de vérification.

En résumé, au prix d'un calcul en $\mathcal{O}(2^{16*sec})$ chez le créateur de l'adresse, on oblige un attaquant à faire un calcul brute force en $\mathcal{O}(2^{59+16*sec})$.

Un attaquant pourrait aussi se constituer un dictionnaire de couples <clé publique, adresse> (précalculés) et essayer de scanner des réseaux à la recherche d'une adresse qu'il connaît. C'est pour compliquer ce type d'attaque que le préfixe de sous-réseau est inclus dans le calcul du hash : l'attaquant est maintenant obligé de se constituer un dictionnaire par sous-réseau (puisque le résultat du hash, donc l'adresse, devient dépendant du sous-réseau).

La manière de générer une adresse CGA rend improbables les collisisions mais pas impossibles. Pour rappel l'algorithme DAD (Duplicate Address Detection) permet de vérifier si une adresse est déjà attribuée. La technique de génération d'une adresse CGA tient compte de la possibilité d'apparition d'une collision et prévoit pour cela le paramètre 'compte de collisions'. Il est incrémenté chaque fois que quelqu'un prétend déjà détenir l'adresse, fournissant ainsi directement une nouvelle adresse(en $\mathcal{O}(1)$, hash2 ne doit plus être recalculé). Il faut noter que ce paramètre ne peut dépasser 2, car cela faciliterait le travail d'un attaquant (lui aussi pourrait essayer plusieurs adresses pour le même prix), et de toute façon un nombre de collisions supérieur à trois indique à coup sûr une anomalie[10].

Finalement, il a été prévu que l'on puisse vouloir lier autre chose qu'une clé publique à l'adresse générée. Dans ce but, un champ d'extensions peut être joint comme entrée au calcul du hash. Nous verrons l'usage d'une telle extension à la section 1.3.3.

Si l'on reprend l'exemple de l'attaque sur la détection d'adresse dupliquée, voici ce qui se passe lorsqu'une adresse CGA est utilisée :

- Alice reçoit le préfixe `1000:10:123::0/64` du routeur R, et s'en sert pour générer une adresse CGA. L'identifiant généré est 1, ce qui conduit, par concaténation du préfixe et de l'identifiant, à la construction de l'adresse `1000:10:123::1`. (Puisqu'un identifiant calculé par CGA n'est ni universel, ni associé à un groupe, les bits u et g valent 0).
- Alice envoie un message *Neighbor Solicitation* pour demander si quelqu'un possède déjà cette adresse. Comme Alice n'affirme rien, mais se contente de demander une information, il est inutile de signer.
- Si un utilisateur normal possède la même adresse, alors il dispose d'une clé privée (pas nécessairement la même que celle d'Alice) lui permettant de signer un message NA (Neighbor Advertisement). Celui-ci contient évidemment l'adresse, mais aussi une signature faite avec la clé privée associée à la clé publique qui a servi pour générer l'adresse. L'ensemble des paramètres de génération sont également inclus dans le message (à savoir la clé publique, le compte de collisions, le modificateur et les champs d'extensions). Alice pourra vérifier la signature **et** l'association clé publique-adresse, avec comme conséquence l'incrémentation du compte de collision pour obtenir une nouvelle adresse.
- Si par contre un utilisateur malveillant veut *prétendre* posséder l'adresse, alors il devra signer son message avec une certaine clé privée. Il devra trouver une clé privée dont la clé publique associée permet d'obtenir la même adresse par l'algorithme de génération décrit ci-dessus. Cela lui prendra $\mathcal{O}(2^{59+16*sec})$.

Le lecteur intéressé par une description plus complète de CGA et SEND en trouvera une étude claire dans [9] et [5].

1.3 SHIM6

1.3.1 Pourquoi shim6 ?

De nombreuses propositions ont été faites pour la mise en oeuvre du multihoming dans IPv6. Celles-ci peuvent être divisées en trois catégories, selon l'emplacement du mécanisme de multihoming [18] :

- Routing : les mécanismes de gestion du multihoming sont implantés dans les routeurs d'Internet. *IPv6 multihoming with BGP* appartient à cette classe. Il s'agit de l'adaptation de la technique présentée à la section 1.1.1 pour IPv6. De ce fait, comme pour le cas d'IPv4, cette solution n'est envisageable que pour les ISPs de transit.
- Middle-box : dans ce cas le support du multihoming est installé dans un équipement placé entre le réseau multi-hébergé et ses fournisseurs de service. Par exemple pour *IPv6 Multihoming with NAT*, l'équipement est une passerelle NAT. Son fonctionnement est similaire à celui du multihoming avec NAT dans IPv4 (section 1.1.2), et présente les mêmes inconvénients (difficultés dans les couches transport et application).
  De manière générale, l'usage d'un équipement intermédiaire (la 'middle-box') crée des problèmes pour les configurations dispersées géographiquement.
- Host-centric : c'est la classe à laquelle appartient `shim6`. Les mécanismes regroupés sous ce nom sont implantés dans les hôtes, c'est-à-dire dans les machines qui communiquent aux extrémités de la ligne. Ce système a l'avantage de ne nécessiter ni l'ajout d'un équipement intermédiaire (*middle-box*), ni une modification ou configuration spéciale dans les routeurs d'Internet (*routing*). Par contre il est nécessaire d'installer le nouveau protocole sur chaque hôte nécessitant le support du multihoming.

C'est cette dernière solution qui a été choisie par l'IETF. L'installation des mécanismes de multihoming dans les hôtes implique que c'est l'hôte lui-même qui va choisir son fournisseur. Pour cela, il se voit attribuer plusieurs adresses PA (Provider Aggregatable), comme c'était le cas avec le système NAT (section 1.1.2). La différence est que le choix de l'adresse n'appartient plus à *un* dispositif, mais à chaque machine du réseau, qui décide elle-même quelle adresse utiliser, et sous quelles conditions changer d'adresse. Cela est illustré à la figure 1.7. Puisque le choix du chemin employé se fait au moyen d'une sélection de l'adresse source par l'émetteur, les routeurs de son réseau doivent être configurés pour diriger les paquets correctement, selon l'adresse source fournie.

Une difficulté apparaît néanmoins avec cette solution : il est plus difficile de faire du traffic engineering. Du fait que chaque hôte décide chez quel fournisseur il va envoyer ses paquets, on n'a plus de dispositif central où gérer le traffic engineering. La mise en oeuvre du traffic engineering dans ce cas est largement discutée dans [18].

Comme indiqué à la section 1.1.2, l'un des inconvénients de l'usage d'un NAT est qu'un changement d'adresse provoque une coupure de la connexion transport. Dans l'approche *host-centric*, cela peut être évité, puisque on travaille dans l'hôte lui-même. On peut dès lors :

- Soit modifier les protocoles de couche transport pour les rendre capable de changer d'adresse dans une même connexion (Une modification de TCP : TCP-MH).
- Soit créer un nouveau protocole tranport (SCTP,DCCP).
- Soit créer une nouvelle couche dans la pile TCP/IP, assurant une traduction, de manière similaire à ce que faisait le système NAT, pour présenter au protocole transport un identifiant constant, quels que soient les changements d'adresse IP (HIP, SHIM,SIM,NOID,CB64,WIMP). La raison pour laquelle les connexions TCP sont persistantes à travers les changement d'adresses est que la traduction se fait *aux deux extrémités de la connection*, de telle manière que les deux corres-

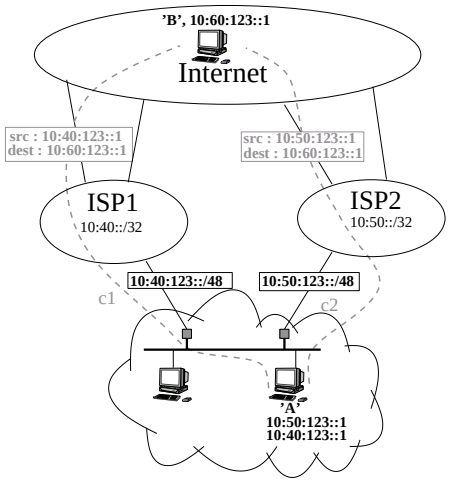


FIG. 1.7 – Multihoming de type *host-centric*

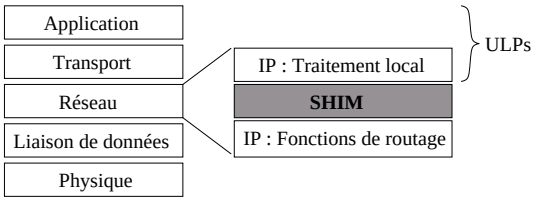


FIG. 1.8 – Insertion de `shim6` dans la pile TCP/IP

pondants voient en permanence des identifiants constants (au-dessus de la couche réseau). Bien sûr cela suppose que le protocole soit supporté par les deux extrémités de la communication.

La dernière de ces options a été retenue, car elle permet de réutiliser tels quels les protocoles existants. La mise à jour des machines pour supporter le multihoming revient alors seulement à insérer un module “multihoming" dans la pile IP (fig. 1.8). C’est ce que nous tenterons de faire concrètement dans ce mémoire, pour le système d’exploitation Linux.

1.3.2 Etude de l’architecture de shim6

Le principe de base de `shim6` est illustré à la figure 1.9. Le rôle de la couche Shim est de modifier les champs source et destination de l’en-tête IP, aussi bien des paquets venant des couches transport et application[2], que de ceux qui viennent du réseau. Le but de cette manipulation est de séparer les concepts de *localisateur* (l’adresse IP) du concept d’*identifiant* pour les couches supérieures (ULID, Upper Layer IDentifier), qui est aussi l’adresse IP si l’on n’utilise pas Shim. Il faut par ailleurs veiller à ne pas confondre la notion d’ULID avec celle d’identifiant d’interface IPv6 : le premier est codé sur 128 bits et a le format d’IPv6, alors que le second est codé sur 64 bits et respecte le format EUI-64. Pour limiter les ambiguïtés, nous nous référrons au premier le plus souvent sous le terme *ULID*.

Un état est maintenu dans la couche Shim, liant une paire d’ULIDs à une paire de groupe de localisateurs. Ainsi, dans l’exemple de la figure 1.9, les ULIDs sont `ulid_a` et `ulid_b`. Lorsque `'A'` envoie un message à `'B'`, la couche Shim se sert du couple `(ulid_a, ulid_b)`, pour re-

[2]On parle généralement de ces couches comme les *protocoles de niveau supérieur* (en comparaison à la couche Shim), ou ULPs (Upper Layer Protocols)

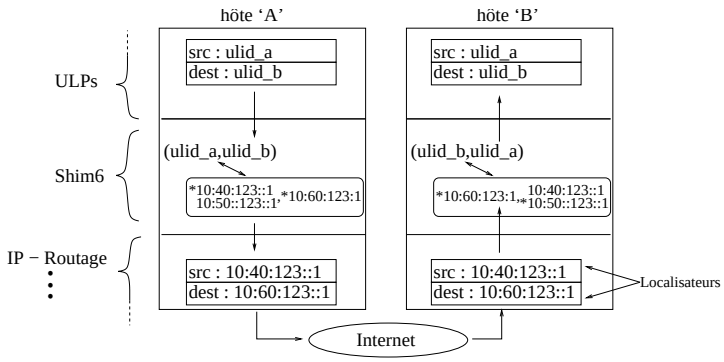


FIG. 1.9 – Modification des paquets dans la couche Shim (communication en cours)

trouver l'état correspondant à la connexion, et ainsi l'ensemble des localisateurs source et destination. Le localisateur actuellement utilisé est marqué d'une étoile. `'A'` utilise pour le moment le localisateur source `10:40:123::1`, de sorte que le paquet empruntera le chemin `c1` (fig. 1.7). L'hôte `'B'` supporte `shim6`, mais n'est pas multi-hébergé, on a donc un seul choix de localisateur pour lui.

A l'arrivée dans `'B'`, le paquet atteint la couche Shim "par en-dessous", et cette fois c'est le couple de localisateurs `10:40:123::1,10:60:123::1` qui permettra de retrouver l'état, et restituer le paquet aux ULPs avec les identifiants source et destination corrects. Néanmoins, nous verrons plus loin que le couple de localisateurs peut, seul, ne pas suffir à identifier un état `shim6`.

Dans l'hôte `'B'`, le localisateur courant utilisé pour joindre son correspondant `'A'` (marqué d'une étoile) est `10:50:123::1`. Cela implique qu'un chemin différent sera emprunté pour les deux sens de la communication : dans le sens `'B'` → `'A'`, le chemin `c2` (fig. 1.7) sera utilisé.

Contraintes sur le choix du ULID : Le ULID doit être un nombre codé sur 128 bits, puisqu'il sera utilisé par les ULPs (Upper Layer Protocols), qui s'attendent à travailler avec des adresses IP.

Dans `shim6`, le choix qui a été fait pour le ULID est de prendre l'un des localisateurs. Cette approche a l'avantage de permettre d'ignorer la couche Shim aussi longtemps que la communication se déroule correctement : aucune traduction n'est effectuée, on a *ULID=localisateur*. C'est seulement lorsqu'un changement de localisateur est déclenché que la couche Shim commencera à travailler.

Interaction avec des machines sans support de shim6 : Grâce au système de ULID qui vient d'être décrit, la communication avec une machine ne possédant pas de couche Shim se fait de la manière habituelle jusqu'à ce que l'un des hôtes décide d'initialiser le protocole de multihoming. A ce moment, la procédure *Shim6 capability detection* est exécutée. [13] conseille d'utiliser un mécanisme de type *Host based dynamic discovery*, qui consiste en un unique échange entre les deux stations. Par exemple, lorsque la station `'A'` veut tenter d'initialiser le multihoming avec `'B'`, elle lui enverra un message *"Can you speak shim6 ?"*, auquel `'B'` répondra *"Yes, i can !"*. `'A'` sait alors qu'il peut entamer la procédure d'initialisation (décrite au paragraphe suivant). En pratique, ce message peut être inclus dans le premier échange d'initialisation, épargnant la durée d'un échange dans le processus complet.

Initialisation du protocole shim6 entre deux machines : Cette initialisation *doit* se faire avant qu'un changement d'adresse ne devienne nécessaire [13]. Ceci est dû au fait qu'au moins l'un des

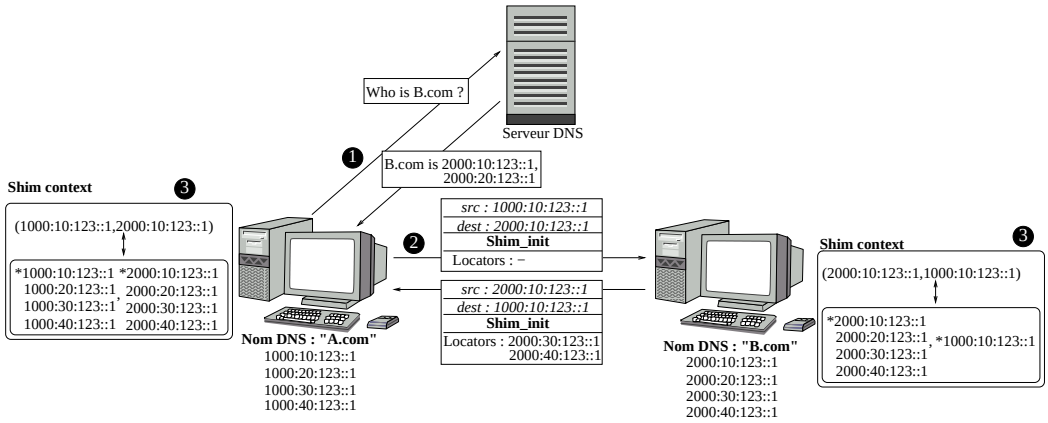


FIG. 1.10 – Etablissement du contexte `shim6`

hôtes doit connaître d'autres localisateurs pour son correspondant, puisqu'il ne peut utiliser un localisateur alternatif sans avoir la liste échangée initialement. Plus précisément, [13] recommande de définir un protocole d'établissement de contexte qui permette d'échanger autant de localisateurs que l'on souhaite, ce nombre pouvant être 0.

Le choix du nombre de localisateurs échangés devrait être un compromis entre le degré de tolérance aux pannes voulu et la charge des paquets échangés durant l'initialisation : par exemple pour améliorer la tolérance aux pannes on voudrait échanger beaucoup de localisateurs. A l'autre extrême, si la connexion est jugée suffisamment sûre, on peut décider de n'en envoyer aucun, l'ULID étant alors utilisé comme unique localisateur. Cependant, même dans ce cas, il est encore possible de changer de lien, car même si aucune adresse alternative n'est échangée, chaque machine connaît ses propres localisateurs, et peut donc au moins essayer de changer l'adresse source.

Une application de l'initialisation de `shim6` est présentée à la figure 1.10. Il s'agit du début d'un échange entre deux stations supportant `shim6` :

1. La station `A.com` souhaite communiquer avec `B.com`, et demande donc son adresse IP à un serveur DNS. Il se fait que pour cette station le serveur possède deux adresses possibles, `2000:10:123::1` et `2000:20:123::1`[3]. `A.com` choisit l'une de ces adresses à la fois comme localisateur courant et comme ULID.

2. `A.com` envoie un message `Shim_init` à `B.com`, mais décide de ne lui transmettre aucun localisateur. En réalité il en transmet un implicitement via l'adresse source, qui sera le seul que pourra utiliser `B.com` pour joindre `A.com`.

   `B.com` répond en donnant seulement deux de ses localisateurs, puisqu'il sait que les deux autres sont fournis par le serveur DNS.

   L'échange de ces messages permet également aux deux stations de savoir que leur correspondant supporte effectivement `shim6` (`B.com` n'aurait pas répondu s'il ne supportait pas `shim6`).

   En réalité l'échange de messages a été légèrement simplifié ici pour des raisons de clarté. A la section 1.3.3, nous verrons les modifiations qu'il faut lui apporter pour se protéger contre certaines attaques.

3. Les deux stations peuvent alors créer leur contexte de couche Shim. Ce contexte est, au départ, tel que les localisateurs courants (marqués d'une étoile) sont les ULIDs, ce qui signifie que

[3]Plus de détails sur l'interaction avec les serveurs DNS dans le cadre de `shim6` seront donnés à la section 1.3.4

la couche Shim n'effectue aucune traduction. Plus tard si un changement de localisateur est déclenché, la couche Shim travaillera comme le montre la figure 1.9.

Nous donnons ici un exemple où le contexte est créé immédiatement lors du premier contact avec le correspondant. Ceci n'est pas une obligation [33]. On pourrait décider d'établir un contexte `shim6` plus tard dans la communication, mais tant que le contexte n'est pas établi, les stations fonctionnent comme si `shim6` n'était pas supporté et ne pourront donc pas profiter du mécanisme de récupération en cas de panne.

Communication entre deux machines : Une fois le contexte établi, la communication se fait comme indiqué à la figure 1.9. Cependant l'identification du contexte associé à un paquet reçu ou émis demande encore quelques précisions. Imaginons par exemple que `A.com` possède déjà, grâce au mécanisme décrit précédemment, une connexion TCP avec `B.com` utilisant la paire d'ULIDs `(1000:10:123::1,2000:10:123::1)`. Maintenant `A.com` décide de démarrer une seconde connexion TCP avec la même machine `B.com`, mais en faisant cette fois appel à la paire d'ULIDs `(1000:10:123::1,2000:20:123::1)`, utilisant donc la seconde adresse fournie par le serveur DNS. Lorsqu'un paquet arrive à `A.com` en provenance de `B.com`, la station réceptrice ne saura pas si le localisateur du champ IP *source* doit être traduit en `2000:10:123::1` ou s'il doit l'être en `2000:20:123::1`. Pourtant TCP s'attend à voir toujours la même adresse !

La résolution de ce problème est actuellement en cours de discussion dans le groupe de travail `shim6`. Il y a globalement deux options :

- S'arranger pour que ce soit toujours le même localisateur qui soit utilisé comme ULID.
- Ajouter une information supplémentaire (*context tag*) dans les paquets, servant de discriminant entre les couples possibles d'ULIDs. Il est question de placer ce context tag dans le champ *flow label* de l'en-tête IPv6, ou de créer une extension d'en-tête spécifique à `shim6`.

Les possibilités d'identification du contexte pour les paquets entrants sont étudiées plus en détails à la section 9 de [33].

Déclenchement d'un changement d'adresse : Le multihoming avec `shim6` s'appuie sur le fait que l'hôte multi-hébergé dispose d'adresses multiples. Il reste maintenant à décider sous quelles conditions la couche shim décide d'effectuer un changement d'adresse. Ce problème est étudié en détail dans [6].

Un changement d'adresse est généralement motivé par une perte de lien avec le correspondant, la congestion de ce lien ou le simple souhait de l'utilisateur de changer le chemin employé pour son trafic. Dans de nombreux cas, l'information peut être apportée par les couches inférieures (réseau/liaison de données) ou supérieures (transport/application). Par exemple la couche liaison de données peut indiquer qu'un lien n'est plus utilisable ; TCP peut fournir des indications de congestion, et l'utilisateur peut indiquer son souhait de changer d'adresse via une interface adéquate de la couche application. Shim peut recevoir des informations des autres couches de la pile de deux manières :

- *Feedback positif* : la couche Shim reçoit régulièrement des messages lui indiquant un fonctionnement normal de la communication. Un changement d'adresse est déclenché, sur expiration d'un temporisateur, lorsque les messages de feedback ne sont plus reçus (ex. : TCP annonce à `shim6` de temps en temps qu'il parvient à échanger des segments sans problème).
- *Feedback négatif* : la couche Shim est alertée explicitement lorsqu'un changement d'adresse devient nécessaire (ex. : expiration du temporisateur dans la couche liaison de données, indication de congestion TCP, requête de la couche application).

Une fois le saut déclenché, il faut choisir une paire d'adresses à utiliser en remplacement. Bien sûr, lorsque de nouveaux localisateurs sont choisis, il est nécessaire de vérifier s'ils sont effectivement joignables avant de réellement faire le changement (une rupture de lien par exemple pourrait rendre inaccessibles plusieurs adresses). Cette vérification peut se faire au moyen d'un simple échange de messages : si le correspondant répond, alors il est joignable.

L'ensemble du processus de changement d'adresse doit se faire très rapidement (avant expiration du temporisateur TCP, sans quoi la connexion est perdue, ce que `shim6` cherche précisément à éviter). Il est donc primordial d'ordonner les adresses de manière à essayer d'abord celles qui sont le plus vraisemblablement joignables. Diverses solutions à ce problème sont proposées dans [6].

1.3.3 Aspects de sécurité

Dans cette section, nous étudierons quelques attaques possibles dans le contexte de `shim6`, et verrons comment elles peuvent être contrées. Le mécanisme HBA (Hash Based Addresses), proche de CGA (voir section 1.2.4), sera ensuite proposé comme solution à certaines des attaques.

Parcours des attaques relatives à shim6

La principale caractéristique de `shim6` est la possibilité de rediriger un flux vers un autre localisateur que celui utilisé actuellement. Un attaquant pourrait vouloir profiter de cela pour rediriger un flux vers son adresse. On pourrait imaginer qu'un attaquant tente d'inclure son adresse dans l'échange d'initialisation de shim. Ensuite, soit directement, soit après rehoming (un rehoming peut par exemple être provoqué par une attaque DoS sur le lien utilisé par le couple courant d'adresses), le flux est redirigé vers l'attaquant. Une attaque MITM (Man-In-the-Middle) est alors possible, si l'attaquant effectue une traduction de localisateurs et relaie les paquets, éventuellement modifiés, à la destination. Ce type d'attaque, où l'attaquant se place dans le chemin 'au bon moment', puis le quitte pour en récupérer des avantages plus tard, est appelé *time shifting attack* ou *future attack*.

On peut aussi, par ce même mécanisme de redirection, envisager une attaque DoS en redirigeant un flux important d'un serveur vers une machine ou un lien[11].

On verra par la suite que le mécanisme HBA fournit une protection contre ce type de problème.

Comme pour tout protocole, un point critique est l'initialisation : il est important de s'assurer que le premier message d'initialisation demande le moins de calcul possible, pour limiter les risques d'attaques DoS par bombardement de messages d'initialisation. A cette fin, [13] propose un échange d'initialisation en 4 étapes. Si l'on utilise ce système, un contexte n'est créé chez le destinataire que lorsque le demandeur de connexion aura prouvé que lui-même possède déjà un contexte.

Une autre attaque envisageable est de jouer sur le déclenchement du rehoming [6] : on peut imaginer qu'un attaquant envoie des messages *ICMP network unreachable* pour forcer une procédure de rehoming. Si l'opération est répétée, une station passera son temps à changer de localisateurs, le tout résultant en une attaque DoS. La manière de s'en protéger est simplement de ne pas déclencher la procédure de rehoming sur simple réception d'un message ICMP. Cependant, on peut s'en servir comme déclencheur de la procédure de test de joignabilité [6].

HBA : Hash Based Addresses

L'idée d'HBA est de lier l'ensemble des adresses utilisées par une machine, de telle manière qu'un correspondant puisse vérifier, connaissant une adresse, qu'une autre fait partie du même groupe.

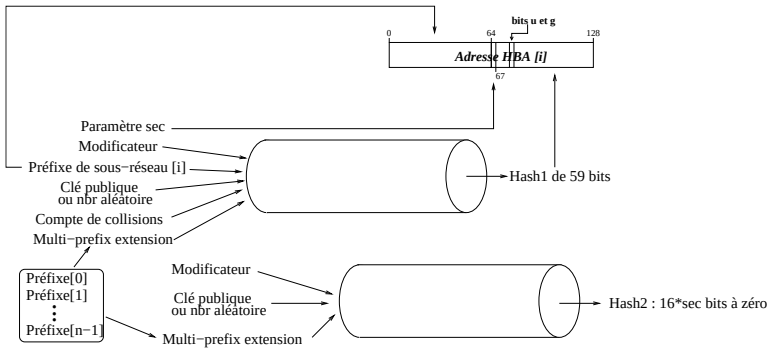


FIG. 1.11 – Génération d'adresses HBA

Il a déjà été dit qu'un identifiant d'interface pour IPv6 peut être généré de multiples manières à condition de se conformer au format EUI-64 modifié (section 1.2.1). Une adresse HBA est constituée d'un préfixe (reçu du fournisseur de services) et d'un identifiant d'interface généré par hashage de telle manière que celle-ci contienne l'information sur l'ensemble des préfixes utilisés par la machine concernée. Pour cela une variante de l'algorithme CGA est utilisée. Celle-ci est présentée à la figure 1.11. En comparant avec la figure 1.6, on s'aperçoit que seules apparaissent les différences suivantes :

- Au lieu de donner un seul préfixe, on fournit l'ensemble de ceux-ci, en faisant usage du champ d'extension prévu par l'algorithme CGA. Cela permet d'inclure l'ensemble des préfixes dans le calcul du hash, et ainsi de lier les adresses entre elles. Cependant le champ préfixe de l'algorithme CGA reste utilisé : pour générer `HBA[i]` (fig 1.11), le champ préfixe prend la valeur `préfixe[i]`. Cela permet que chaque adresse du groupe soit différente, améliorant donc la confidentialité.
- Le champ `Clé publique` peut être remplacé par un nombre aléatoire : l'utilité de HBA n'est pas la même que celle de CGA, néanmoins ils peuvent se compléter de manière utile. Ainsi on peut fournir une clé publique si l'on souhaite que les adresses du groupe soient à la fois HBA et CGA, ou un nombre aléatoire si la fonctionnalité HBA suffit.

CGA est essentiellement une protection contre le spoof, et pour cette raison permet à son utilisateur de prouver qu'il est bien celui qui a créé l'adresse. Dans HBA, le but est de prouver, au moment de fournir une nouvelle adresse, que celle-ci appartient bien au même groupe d'adresses que celle que le correspondant connaît déjà.

Par exemple, appliquons le mécanisme HBA aux schéma de la figure 1.10 (cet exemple est basé sur celui fourni dans [11]) : notons d'abord que dans cette figure, aucune machine n'utilise des adresses HBA, puisque les mêmes identifiants d'interface sont utilisés pour des préfixes différents. Comme expliqué précédemment, cela ne peut se produire avec HBA. Voici une suite d'étapes qui pourraient se produire dans un échange `shim6` :

- `A.com` génère ses adresses HBA en choisissant un modificateur (aléatoirement), une clé publique (afin de décrire dans ce même exemple le rôle que peut jouer CGA), et l'ensemble des préfixes dont il dispose. `B.com` fait de même. Cette étape se fait normalement hors ligne, lorsque l'ordinateur reçoit de nouveaux préfixes, donc assez rarement. Néanmoins si l'on souhaite un niveau de confidentialité important, on peut regénérer l'ensemble d'adresses n'importe quand en utilisant un modificateur différent.
- L'initialisation de `shim6` se fait exactement comme décrit à la section 1.3.2 et présenté à la figure 1.10. Dans le message `Shim_init`, une structure de données contenant l'ensemble des paramètres HBA doit également être échangée.

- `A.com` sait qu'il peut joindre `B.com` par les adresses `2000:i0:123::hba_b_i` (où i vaut resp. 1,2,3,4). Il vérifie que toutes les adresses sont bien liées entre elles en appliquant l'algorithme de vérification à chacune d'entre elles. Suite à la vérification, `A.com` a la garantie que l'ensemble des adresses ont été générées par le même équipement. De plus, l'une de ces adresses est utilisée dans la communication courante. La conclusion qui peut être tirée est qu'un changement d'adresse fera nécessairement basculer sur un localisateur appartenant à la même machine que celle avec qui `A.com` communique, en l'occurrence `B.com`.
  `B.com` ne peut par contre pas faire la vérification puisque pour le moment il n'a qu'une adresse pour joindre `A.com`. Néanmoins, puisqu'il dispose de la structure de données de paramètres HBA, il pourra vérifier les adresses que `A.com` lui enverra éventuellement plus tard.
- Supposons maintenant qu'une nouvelle adresse devienne disponible chez `A.com`, par exemple `1000:50:123::1`, et qu'il souhaite l'annoncer à `B.com`. Cette adresse n'est ni HBA ni CGA. Il ne peut pas l'inclure au groupe déjà généré, à moins de regénérer tout le groupe, ce qui en changerait toutes les adresses. C'est là qu'apparaît l'utilité de CGA : la nouvelle adresse peut être envoyée dans un message signé avec la clé privée correspondante à la clé publique des paramètres HBA. Il faut par ailleurs que la nouvelle adresse (`1000:50:123::1`) ait été générée avec cette même clé publique (voir section 1.2.4).
- `B.com`, après vérification de la signature et du lien entre la clé publique et l'adresse, peut inclure la nouvelle adresse dans son ensemble de localisateurs pour `A.com`.

Voyons maintenant dans quelle mauvaise posture nous plaçons un attaquant de type 'time shifting', lorsque le mécanisme HBA est utilisé : comme décrit plus haut, il faut dans une telle situation que l'attaquant parvienne à se mettre sur le chemin au moment où l'échange initial est effectué, car c'est là qu'il faut modifier les paquets pour ajouter des localisateurs. Donc imaginons que le paquet `Shim_init` soit intercepté par Trudy (il est notoire que les utilisateurs malveillants aiment s'appeler ainsi...). L'adresse qu'ajoutera Trudy devra être liée à celles qui sont annoncées, ou au moins au localisateur courant utilisé (si l'on peut tromper sur les localisateurs alternatifs, on ne le peut pas sur celui qui est en cours d'utilisation : cela romprait la communication). Il devra pour cela trouver une valeur pour le paramètre modificateur qui permette d'obtenir le localisateur courant, en plus de l'adresse qu'il souhaite ajouter. A cause de l'extension artificielle de la longueur de hash, identique à celle décrite à la section 1.2.4, cette opération se fera en $\mathcal{O}(2^{59+16*sec})$.

Une description précise du niveau et de l'étendue de la sécurité fournie par HBA est présentée à la section 9 de [11].

1.3.4 Interaction avec les couches supérieures et le DNS

Jusqu'ici, `shim6` a surtout été étudié 'par en-dessous', c'est-à-dire du point de vue des localisateurs. Cette section s'intéresse cette fois à l'utilisation de l'ULID dans les applications, et à l'usage du DNS.

Un choix de design important qui a été fait dans `shim6` (contrairement à d'autres protocoles de multihoming) est de ne pas dépendre du DNS pour fonctionner. Ainsi par exemple, il est possible mais pas requis d'enregistrer plusieurs localisateurs sous forme de champs DNS. Le moyen standard pour s'échanger des localisateurs reste l'échange d'initialisation de `shim6`.

Ce choix de ne pas dépendre du DNS a été fait pour deux raisons :

- Le protocole est pensé pour nécessiter un minimum de modifications du matériel/logiciel existant pour fonctionner.
- Bien que certaines adresses peuvent être stables et donc enregistrées éventuellement dans un serveur, d'autres peuvent changer souvent, rendant les informations du DNS inexactes à certains

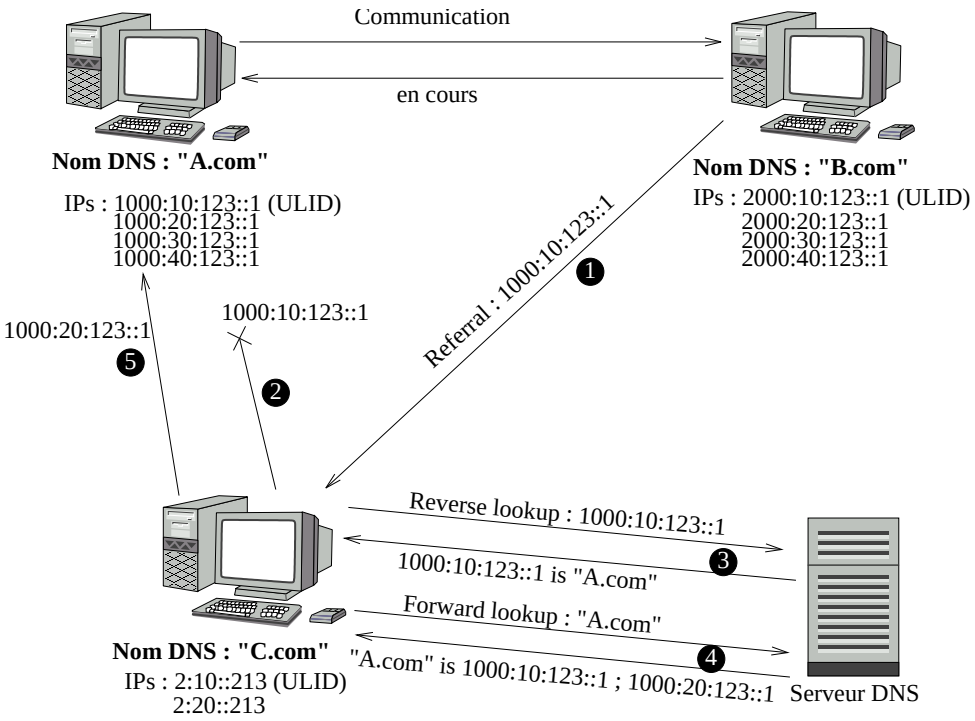


FIG. 1.12 – Passage d'une adresse à une tierce partie

moments si l'on recourt à cette méthode.

Une utilisation particulière du DNS pour une station multi-hébergée est de faire appel à l'arbre DNS inversé (décrit à la section 1.2.2). En particulier, ce sera le cas lorsque une station fait un *referral* (référencement d'une autre station à une tierce-partie au niveau de la couche application). Ceci est illustré à la figure 1.12 :

1. `A.com` et `B.com` sont en conversation. Dans le cadre de l'application utilisée, `B.com` envoie à `C.com` la référence `1000:10:123::1` afin qu'il puisse lui aussi parler avec `A.com`. Cette référence ne peut être un localisateur (au sens strict du terme : plus précisément, comme un ULID est toujours un localisateur particulier, on pourrait dire que la référence est toujours un localisateur qui appartient à l'ensemble des ULIDs possibles), puisque les applications ne voient que les ULIDs.
2. `C.com` cherche à joindre l'adresse qui lui a été donnée, sans succès.
3. `C.com` veut donc tenter d'utiliser un autre localisateur pour joindre A, mais il n'a pas encore de contexte `shim6`. Donc le seul moyen d'espérer obtenir d'autres localisateurs pour cette même machine est d'avoir recours au DNS, pour voir si le serveur DNS peut fournir d'autres adresses que celle-là. Mais comme le nom DNS de l'adresse n'est pas connu, il faut faire une requête inversée (*Reverse lookup*) suivie d'une requête directe (*forward lookup*).
4. Heureusement, on obtient ainsi une nouvelle adresse que l'on pourra tenter de joindre, à savoir `1000:20:123::1`. Il faut noter qu'à ce stade, il n'y a encore aucune distinction entre ULID et localisateur, précisément parce que le système DNS n'a pas été modifié pour faire la différence.
5. `C.com` tente, cette fois avec succès, de joindre l'adresse `1000:20:123::1`. Il a alors le choix d'avoir ou non recours au multihoming. S'il le souhaite, il peut lancer une initialisation de session `shim6` avec `A.com`, ce qui lui permettrait de récupérer ses autres adresses. Dans ce

cas, `A.com` travaillera avec deux ULIDs différents : il gardera son ULID courant pour la communication avec `B.com`, et créera un contexte `shim6` avec pour ULID `1000:20:123::1` pour la communication avec `C.com`.

L'exemple ci-dessus montre l'utilité de fournir plusieurs adresses alternatives dans le DNS. Pour cela il est bien sûr recommandé de choisir des adresses stables. Il montre également que la présence d'adresses multiples fournies par le DNS a pour conséquence l'utilisation (possible) d'ULIDs multiples dans l'hôte visé.

Bien que cette façon de gérer les *referrals* puisse être utile, en particulier pour éviter de devoir modifier des applications qui transmettent seulement une adresse IP, elle possède également des inconvénients. [30] cite comme cas problématique un ordinateur personnel connecté à deux fournisseurs, qui chacun gère indépendamment le DNS et le DNS inversé de ses clients. Dans ce cas, le schéma de la figure 1.12 reste valable jusqu'à l'étape 4, mais alors `C.com` ne recevrait qu'une adresse, qui serait celle qu'il a déjà (parce que les noms DNS correspondants aux deux adresses seraient différents). La manipulation est donc inutile dans ce cas.

Pour cette raison, [30] examine d'autres possibilités, qui consiteraient à modifier les applications pour leur permettre de profiter pleinement du multihoming. Par exemple, une solution serait de fournir une interface à la couche application, lui permettant d'accéder aux localisateurs, et ainsi d'envoyer l'ensemble complet de localisateurs à la tierce partie.

1.4 Conclusion

Le multihoming est une pratique déjà courante actuellement. Cependant il possède certains inconvénients (posséder un numéro d'AS, ou faire appel à un NAT) qu'un passage à IPv6 permettra de résoudre. La solution qui s'est imposée à l'IETF est celle, de type *host-centric*, où des identifiants de couches supérieures (ULIDs) sont transformés en localisateurs par une couche intermédiaire, `shim6`. Cette couche trouve place au-dessus des fonctions de routage d'IPv6.

Nous avons vu également brièvement les nouveautés introduites pas IPv6, en particulier le concept nouveau d'autoconfiguration d'adresses, dont le but est de donner des caractéristiques “plug & play" à la connectivité réseau. Nous verrons dans les chapitres suivants que ce mécanisme peut utilement être manipulé dans le cadre d'une implémentation de `shim6`.

Chapitre 2

Implémentation des fonctionnalités réseau dans le noyau Linux

Dans ce chapitre, l'architecture réseau de Linux est étudiée, en particulier les parties qui devront être modifiées pour supporter `shim6`. Nous commencerons par introduire divers éléments spécifiques à la programmation dans le noyau Linux, ensuite nous verrons plus spécifiquement le fonctionnement de l'architecture réseau, avec un accent spécial sur IPv6.

Le noyau étudié ici est la version 2.6.15.

2.1 Techniques d'implémentation dans le noyau Linux

Dans cette section, nous passons en revue brièvement quelques techniques d'usage courant dans le noyau, que l'on n'est pas habitué à trouver dans d'autres contextes.

Cette partie n'a pas l'ambition de fournir une introduction à la programmation noyau, qui sortirait de l'objectif de ce mémoire (de bons textes peuvent être trouvés à ce sujet dans [1, 2, 36]). Le but est plutôt de fournir les éléments nécessaires à une bonne compréhension de l'implémentation décrite dans le chapitre suivant.

2.1.1 Le parallélisme dans le noyau Linux

Le noyau Linux est conçu pour pouvoir fonctionner aussi bien sur des équipements UP (Uni-Processors) que SMP (Symmetric Multi-Processors). De plus, à partir du noyau 2.6, Linux est devenu préemptible, avec pour conséquence que même un uni-processeur adopte un comportement similaire à un multi-processeurs.

> Le fait qu'un noyau soit *préemptible* signifie que le contrôle peut être retiré à une tâche noyau pour laisser place à une autre tâche de priorité plus élevée. L'idée est similaire à ce qui se passe pour les processus utilisateurs. C'est le rôle de l'ordonnanceur de gérer la politique de gestion des tâches à exécuter à un instant donné.

Mais pour se rendre compte des conséquences du parallélisme, il faut d'abord avoir connaissance des différentes formes d'activité existant dans le noyau, et leur manière de s'interrompre entre elles. On distingue :

– Les contextes utilisateurs (*user contexts*) : ce sont les parties 'immergées' des processus utilisateurs. Il s'agit de l'exécution d'un processus qui a fait un appel système et travaille donc dans le noyau.

- Les interruptions hardware (*Hard IRQs*) : ces parties de codes sont exécutées sur déclenchement par le hardware, comme le nom l'indique. Elles doivent absolument être traitées très vite et ne peuvent être interrompues. Prenons par exemple le cas du signal d'horloge, dont le traitement immédiat est nécessaire pour assurer la précision des opérations temporisées.
  Les interruptions hardware doivent toujours posséder un code très court. Seul est fait ce qui est urgent, le reste est laissé à plus tard. Ceci est motivé par le fait que si une routine de traitement d'interruption hardware prenait trop de temps, elle empêcherait le traitement d'autres interruptions. D'une manière plus intuitive on se rend compte qu'une interruption hardware, de par sa haute priorité, bloque tout le système pendant son exécution.
- Les interruptions software (*Soft IRQs*) : ce sont des tâches à exécution différée. Par exemple une interruption hardware, plutôt que de faire un travail important en bloquant tout le système, va activer une interruption software. Celle-ci s'exécutera dès que possible par la suite, cette fois sans bloquer le noyau, puisque des interruptions hardware peuvent se produire pendant l'exécution d'une interruption software.

Une différence fondamentale entre un contexte utilisateur et les deux autres modes d'activité se trouve dans le fait que le contexte utilisateur est le seul à avoir accès à l'espace d'adresses d'un processus, donc à pouvoir par exemple faire des copies de données depuis l'espace utilisateur vers l'espace noyau.

Voyons ce qui se passe dans le cas des réseaux. Lorsqu'une trame arrive sur un dispositif réseau, une interruption hardware est déclenchée. Elle ne fait que le travail urgent, qui consiste à copier la trame en mémoire, dans une file d'entrée, et activer l'interruption software `NET_RX`. Dès que possible la tâche associée s'exécutera, de telle manière que la trame mise en file puisse traverser les couches de la pile TCP/IP jusqu'à être transmise à un processus. A ce moment elle est copiée en espace utilisateur et n'appartient plus au noyau.

L'émission de paquets depuis la couche application se fera toujours suite à un appel système, donc en contexte utilisateur. Les exceptions à cette règle sont les suivantes :

- *forwarding* de paquets : on est dans le cas d'un paquet entrant, mais non destiné à l'hôte. Dans ce cas particulier, c'est l'interruption `NET_RX` qui gère à la fois la réception, le routage et l'émission du paquet sur une autre interface.
- Paquets à envoi différé : on peut vouloir cela par exemple si l'on fait du traffic shaping. Dans ce cas le paquet descendra la pile TCP/IP en contexte utilisateur. Ce n'est que lorsque la trame sera prête à être envoyée qu'elle sera mise en file, et que l'interruption software `NET_TX` sera activée. Dès que possible, ou à un instant imposé (cas du traffic shaping), la routine de traitement de l'interruption s'exécutera et enverra la trame à l'interface de sortie.

2.1.2 Verrouillage

Maintenant que les diverses formes d'activité existant dans le noyau sont connues, il reste à spécifier que :

- Lorsque l'on travaille dans un contexte utilisateur, on peut être interrompu par une *soft IRQ* ou une *hard IRQ*.
- Dans une interruption software, on peut être interrompu par une *hard IRQ*.
- Dans une interruption hardware, on ne peut interrompu, sauf dans le cas d'une interruption de type lent. Les interruptions hardware de types lent ont une durée d'exécution trop longue pour se permettre de bloquer le système durant ce temps, c'est pourquoi elles acceptent d'être interrompues elles-même par d'autres interruptions hardware.

Du point de vue du verrouillage, il faudra protéger les structures qui peuvent être atteintes par plusieurs tâches en même temps, ou de manière 'pseudo-simultanée' (préemption). A cause de la grande variété de situations que l'on peut rencontrer dans le noyau, on rencontre aussi une grande variété de verrous. Nous ne verrons ici que les verrous utilisés dans le cadre de l'implémentation de `shim6`. Le lecteur intéressé trouvera des informations complémentaires dans [35].

Sémaphores (*include/asm/arch/semaphore.h*) : Ils permettent à un nombre limité de tâches d'accéder à une zone de code donnée. En pratique, on voudra généralement que ce nombre soit une seule tâche, ce qui est le cas du mutex. Un mutex s'emploie comme suit :

```
static DECLARE_MUTEX(mutex); /*déclaration*/
...
if (down_interruptible(&mutex)) return -EINTR;
__critical_function(); /*Code à protéger*/
up(&new_ctx_mutex);
```

Si un contexte utilisateur A tente d'obtenir le verrou, il l'obtient et exécute le code protégé. Si un autre contexte utilisateur B tente de l'acquérir, il dormira jusqu'à ce que le premier ait terminé (`up`). C'est pour cette raison que les sémaphores ne peuvent être utilisés que dans des contextes utilisateurs : ils sont les seuls à pouvoir passer à l'état `sleeping`.

Le fait que l'on utilise `down_interruptible()` plutôt que `down()`, qui existe aussi, pour réclamer le verrou, permet au contexte de continuer à recevoir des signaux pendant qu'il dort. On en perçoit de suite l'utilité lorsque l'on veut tuer le processus : dans ce cas `down_interruptible()` sera donc interrompu et retournera un code d'erreur `-EINTR`, ce qui permettra de sortir le processus de sa léthargie. C'est pour cette raison que l'on préfèrera en général utiliser la version interruptible de la fonction d'acquisition de mutex.

Une autre observation que l'on peut faire ici est que le nom de la fonction protégée par mutex est précédé de deux caractères de soulignement. C'est ainsi que l'on marque dans le noyau les fonctions qui ne devraient pas être utilisées directement, ou dont l'appel doit être encadré par les verrouillages appropriés.

Spinlocks (*include/linux/spinlock.h*) : Les sémaphores, de par leur limitation aux contextes utilisateurs, sont peu utilisés. Dans la plupart des cas en effet, les données partagées sont gérées par des interruptions software, ou encore à la fois par des interruptions software et des contextes utilisateurs. Dans ces situations il faut faire appel aux spinlocks. Ce type de verrou s'initialise comme suit :

```
spinlock_t lock;
...
spin_lock_init(&lock);
```

Ensuite pour réclamer et libérer le verrou, *si la portion de code concernée s'exécute toujours dans une interruption software* :

```
spin_lock(&lock);
... /*Section critique*/
spin_unlock(&lock);
```

`spin_lock()` est en fait une macro, qui est transformée différemment selon que l'on fonctionne en mode UP, SMP ou avec préemption :

- En mode SMP (Symmetric Multi-Processor) : le parallélisme est réel, et le spin_lock fait du 'busy waiting' (test continuel du verrou jusqu'à ce qu'il se libère).
- En mode UP (Uni-Processor) : Les interruptions software sont désactivées. Le busy waiting est inutile dans ce cas, puisque l'on n'est plus interrompu. Bien sûr on peut toujours être arrêté par

une interruption hardware, mais c'est sans conséquence si la routine de traitement de celle-ci ne manipule pas elle-même la structure de données partagée.
– Sur une machine avec préemption (c'est une option de configuration du noyau) : la préemption est désactivée, inhibant le pseudo-parallélisme (bien sûr il est rétabli par `spin_unlock()`).

Cet usage élégant du pré-processeur permet de réaliser du code portable en supposant toujours que l'on travaille sur un multi-processeur. Les macros permettront au code de toujours fonctionner sur un uni-processeur.

Voyons maintenant ce qui se passe dans le cas d'une structure manipulée aussi bien en contexte utilisateur qu'au sein d'une interruption software. On utilisera alors la variante suivante :

```
spin_lock_bh(&lock)
... /*Section critique*/
spin_unlock_bh(&lock);
```

Si la *soft IRQ* intervient quand le contexte utilisateur possède le verrou, elle attendra indéfiniment qu'il se libère, puisqu'elle ne rendra pas le contrôle au contexte utilisateur. Ce serait un deadlock. C'est pourquoi la variante `spin_lock_bh` désactive les interruptions software *sur le processeur local uniquement* (inutile de le faire pour les autres processeurs, l'important étant seulement de s'assurer que l'on ne bloque pas le contexte utilisateur).

Notons qu'à l'intérieur de l'interruption software, il est permis d'utiliser `spin_lock()` au lieu de sa variante, puisqu'une interruption software ne peut être interrompue par une autre.

rwlocks (*include/linux/spinlock.h*) : Les rwlocks ne nécessitent pas beaucoup plus de commentaires, car ils sont la variante lecture/écriture des spinlocks, qui permet d'avoir plusieurs lecteurs simultanés, mais un seul écrivain. Ils s'utilisent comme dans l'exemple suivant :

```
rwlock_t lock;

rwlock_init(&lock); /*Initialisation*/

write_lock(&lock);
... /*Modification de la structure partagée*/
write_unlock(&lock);
...
read_lock_bh(&lock);
... /*Consultation de la structure partagée*/
read_unlock_bh(&lock);
```

L'exemple met en évidence le fait qu'ici aussi,la variante `_bh` existe, puique le problème est identique à la situation précédente.

Plus de détails sur les spinlocks et rwlocks peuvent être trouvés dans [35] et [39].

Opérations atomiques (*/include/asm/arch/atomic.h*) : L'interface *atomic.h* permet de protéger un nombre *entier* (32 bits), généralement un compteur, pour faire sur lui des opérations de type 'test and set' atomiques. Ce mécanisme peut également être implémenté avec des spinlocks, mais permet une plus grande clarté du code et une meilleure efficacité, car il est possible de tirer parti des possibilités offertes par le processeur. L'exemple suivant présente l'équivalence (conceptuelle) entre les deux mécanismes.

```
atomic_t number;

/* Initialisation */
atomic_set(&number,5);
...
/* Consommation */
a=atomic_inc_return(&number);
```

```
/* Initialisation */
int number=5;
...
/* Consommation */
spin_lock(&some_lock);
number++;
a=number;
spin_unlock(&some_lock);
```

Nous disposons donc d'une interface qui évite d'employer un verrou explicite. Bien-entendu, ceci est au prix de l'usage d'un type spécial, `atomic_t`, qui nous oblige à faire appel aux fonctions de l'interface `atomic.h` même pour une assignation (`atomic_set`).

Beaucoup d'autres fonctions atomiques sont définies et bien commentées dans l'en-tête *atomic.h*. Il est par exemple possible d'additionner ou soustraire une valeur quelconque au nombre et retourner le résultat, toujours de manière atomique.

Une utilisation très courante dans le noyau de cette interface est l'utilisation de compteurs de références. C'est ce que nous étudierons à la section suivante.

2.1.3 Compteurs de références

En plus du verrouillage, une technique doit être utilisée pour s'assurer que plus personne ne dispose de pointeur vers un objet avant de le supprimer. En Java par exemple, ce principe est intégré à la machine virtuelle sous la forme d'un garbage collector qui libère la mémoire occupée par l'ensemble des objets qui ne sont plus référencés. En C, nous devons faire cela à la main. Une technique classique dans le noyau Linux est de placer un compteur (en général appelé `refcnt`, pour reference count) dans la structure que l'on veut protéger. Celui-ci est incrémenté chaque fois qu'une nouvelle référence vers l'objet est créée, et décrémenté lorsqu'une référence disparaît. Lorsque ce compteur arrive à 0, la dernière référence a donc disparu et on peut libérer l'espace mémoire.

Le mécanisme de compte de références peut s'implémenter comme suit :

```
struct shared_data {
    ... /*Champs de données*/
    atomic_t refcnt;
};

struct shared_data* get_data()
{
    struct shared_data* data;
    spin_lock(&lock);
    ... /*Obtention de la donnée
    (dans une hastable par ex.)*/
    atomic_inc(&data->refcnt);
    spin_unlock(&lock);
}
```

```
void use_data()
{
  struct shared_data* data =
                        get_data();
  ... /*Usage de la donnée*/
  if (atomic_dec_and_test(
                &data->refcnt))
                        kfree(data);
}

void delete_data(
        struct shared_data* data)
{
  spin_lock(&lock);
  ... /*Suppression de son
        environnement (par ex.
        table de hashage)*/
   if (atomic_dec_and_test(
                &data->refcnt))
                        kfree(data);
   spin_unlock(&lock);
}
```

C'est la fonction qui renvoie un pointeur frais vers un objet qui provoque l'augmentation du nombre de références vers celui-ci. Elle incrémente donc le compteur de références. Dans la fonction qui utilise le pointeur, il faudra dès lors décrémenter ce compteur dès que la référence n'est plus

utilisée. On a alors deux situations possibles :

- Si l'objet n'a pas été supprimé (`delete_data`), alors typiquement le compteur revient à 1 (sauf si d'autres tâches y accèdent en même temps) et la mémoire n'est pas libérée ;
- Dans le cas où par hasard `delete_data` a été appelé entretemps, alors cette fonction de suppression n'a en fait que descendu le compteur. Mais alors, lorsque `use_data` le descend à son tour, le compteur arrive à zéro et la mémoire est libérée. C'est exactement le comportement voulu : Normalement `delete_data` libère la mémoire, mais si quelqu'un possède encore une référence, c'est le dernier à utiliser la référence qui libère la mémoire.

Maintenant, pourquoi utiliser malgré tout le verrou `lock` ? C'est pour se protéger contre une situation d'exécution parallèle où `use_data` obtiendrait le pointeur, mais avant qu'il n'ait le temps d'incrémenter le compteur de références, `delete_data` le décrémente et libère la mémoire. Dans cette situation, `use_data` travaille ensuite avec un pointeur invalide. En pratique, l'usage d'un verrou n'est qu'un inconvénient mineur car il sera généralement nécessaire pour protéger la structure que l'on modifie en supprimant l'objet (une hashtable par exemple).

Il est néanmoins intéressant de voir que `use_data` n'a pas besoin d'utiliser le verrou : C'est parce que tant que `delete_data` n'a pas été invoqué, `use_data` ne risque de toute façon pas de libérer la mémoire. Par ailleurs si `delete_data` est invoquée entretemps, alors plus personne n'obtiendra le pointeur, puisque l'objet aura été rendu indisponible (ôté de la hashtable). La valeur du compteur après suppression sera égale au nombre de tâches qui l'auront obtenues *avant* la suppression, si bien que sans emploi de verrou supplémentaire (grâce au fait justement que l'opération de décrémentation est atomique), la mémoire sera correctement libérée par la dernière tâche à se servir du pointeur.

Puisque cette technique est largement utilisée dans l'ensemble du noyau, une structure a été proposée pour unifier la gestion du compte de références, et ainsi augmenter la clarté du code et en diminuer les erreurs. Il s'agit d'une interface simple disponible dans *include/linux/kref.h*, dont l'emploi est décrit dans [26].

L'exemple ci-dessus devient alors :

```
struct shared_data {
    ... /*Champs de données*/
    struct kref kref;
};

/*Initialisation*/
kref_init(&data->kref);

struct shared_data* get_data()
{
    struct shared_data* data;
    spin_lock(&lock);
    ... /*Obtention de la donnée
          (dans une hastable par ex.)*/
    kref_get(&data->refcnt);
    spin_unlock(&lock);
}

void data_release(struct kref* kref)
{
    struct shared_data* data =
        container_of(kref,
             struct shared_data,kref);
    kfree(data);
}
```

```
void use_data()
{
    struct shared_data* data =
                          get_data();
    ... /*Usage de la donnée*/
    kref_put(&data->kref,data_release);
}

void delete_data(
        struct shared_data* data)
{
    spin_lock(&lock);
    ... /*Suppression de son
          environnement (par ex.
          table de hashage)*/
    kref_put(&data->kref,data_release);
    spin_unlock(&lock);
}
```

`kref_put` reçoit comme argument la fonction de suppression de contexte, car chaque fois qu'on l'appelle, on est *susceptible* (si le compteur arrive à zéro) de devoir libérer la mémoire.

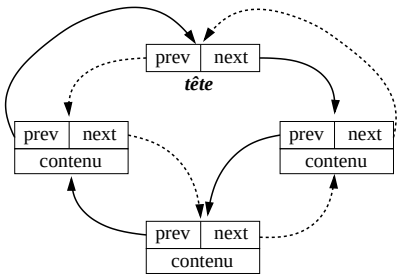


FIG. 2.1 – Liste circulaire doublement chaînée, *list.h*

2.1.4 Temporisateurs(*include/linux/timer.h*)

Les temporisateurs permettent l'exécution d'une tâche à un instant précis. Ils s'utilisent comme suit :

```
void handler(unsigned long data)
{
   struct foo* arg=(struct foo*)data;
   ...  /*Code à exécution différée*/
}

struct timer_list timer;
init_timer(&timer); /*Initialisation*/
timer.expires=jiffies+10*HZ;
timer.function=&handler;
timer.data=arg;
add_timer(&timer); /*Activation*/
```

Dans un temporisateur, il faut spécifier à quel instant on veut faire exécuter la routine de traitement, `handler(data)`. Cet instant doit être spécifié de manière absolue, en unité de *jiffies*. Un jiffie correspond à une interruption de temporisateur (la variable globale `jiffies` est incrémentée à chaque interruption de temporisateur ; sachant qu'il y a `HZ` interruptions de temporisateur par seconde, on peut trouver la valeur à donner au champ `expires`. Ici la fonction `handler(data)` s'exécutera donc dans 10 secondes.

L'exemple montre aussi un usage courant concernant l'argument passé au gestionnaire : comme un entier long sera souvent une information insuffisante, on préfère s'en servir pour stocker un pointeur vers une structure complète. Il suffit alors de faire un casting pour récupérer la structure dans la routine du temporisateur.

Finalement `add_timer(&timer)` place la structure `timer_list` dans la liste de temporisateurs en attente (d'où le nom de la structure) pour exécution. Par la suite on pourra toujours modifier l'échéance (`mod_timer`) ou supprimer le temporisateur (`del_timer`).

2.1.5 Listes et tables de hashage

Les listes sont très utilisées dans le noyau, ainsi que les tables de hashage. Leur implémentation varie selon les besoins. Souvent on crée des listes simplement en ajoutant dans une structure un pointeur vers l'élément suivant. De même les tables de hashage sont souvent faites 'à la main', avec un hash spécialisé pour l'application voulue, ou calculé au moyen de l'interface *include/linux/jhash.h* ou *include/linux/hash.h*

Cependant lorsque l'on veut souvent itérer à travers une liste, ajouter ou supprimer des éléments, alors l'interface *include/linux/list.h* devient très utile : elle définit une liste circulaire doublement chaînée, qui se construit de la manière suivante :

```
/*——Extrait de list.h——*/
struct list_head {
        struct list_head *next,*prev;
};
/*——Fin de l'extrait——*/

LIST_HEAD(mylist);
struct foo {
        int data;
        /*other data*/
        struct list_head foo_list;
};
```

Si l'on compare cet exemple à la figure 2.1, on a comme tête de liste `mylist` (qui est du type `struct list_head`, et est automatiquement initialisé à une liste vide par la macro). Les noeuds sont des structures `foo`. Par contre les valeurs pointées par les champs `next` et `prev` correspondent non pas à la structure `foo` suivante ou précédente, mais bien au champ `foo_list` de celle-ci. Il est cependant toujours possible de remonter à la structure 'conteneur' grâce à la macro `container_of` (*include/linux/kernel.h*). Néanmoins il est normalement inutile de s'en servir directement dans le programme, l'interface *list.h* fournissant tous les moyens d'accès utiles.

Parmi celles-ci, une macro permet d'itérer sur la liste de manière simple :

```
list_for_each_entry(pos, head, member)
```

- `pos` : un pointeur vers la structure sur laquelle on veut travailler (ex. `struct foo* it`). La macro se charge de nous donner la suivante (en remplissant le pointeur) à chaque itération et de s'arrêter en fin de liste ;
- `head` : Un pointeur vers la tête de liste (ex. : `&mylist`) ;
- `member` : Le nom du champ correspondant à la liste, à l'intérieur de la structure sur laquelle on itère. Dans l'exemple, c'est `foo_list`.

Une variante, qui nous sera utile pour l'implémentation de `shim6`, est :

```
list_for_each_entry_safe(pos,n,head,member)
```

Cette variante peut être utilisée de manière identique à la précédente, à la différence près que l'on a ici le droit de supprimer un objet de la liste en cours de traversée. C'est pour cela que l'argument `n` est ajouté : il faut y donner un pointeur temporaire vers le même type d'objet que `pos`, que la macro se charge de remplir et qu'elle utilise pour garder une référence vers l'objet suivant.

2.1.6 L'initialisation du noyau (*include/linux/init.h*)

Il est souvent nécessaire, lorsqu'on ajoute une fonction au noyau, d'en permettre l'initialisation au moment du démarrage de la machine. On obtient ce résultat facilement par le code suivant :

```
int __init init_function(void) {...}
module_init(init_function);
```

La directive `__init` est placée dans la signature de la fonction. Cela indique au noyau de placer le code correspondant dans une zone spécifique de la mémoire, qui sera libérée après le boot. On évite ainsi d'encombrer inutilement la RAM avec des fonctions qui ne servent qu'une fois.

Ensuite la macro `module_init` indique que cette fonction doit être exécutée au moment du boot.

En réalité, ce code fonctionne aussi dans le cas où l'on compile du code pour un emploi en tant que module (un module est un bloc de code que l'on insère/enlève dynamiquement dans le noyau en cours d'exécution). Dans ce cas, la directive `__init` est sans effet, et la macro `module_init()` indique que la fonction doit être exécutée au moment de l'insertion dans le noyau (plutôt qu'au boot).

2.1.7 Interaction avec le noyau en cours d'exécution

Lorsque le noyau s'exécute, il est intéressant de pouvoir en modifier le comportement dynamiquement. Il est également utile de pouvoir extraire de l'information sur ce qui s'y produit, par exemple pour faire des statistiques ou détecter des anomalies du comportement (debugging).

Plusieurs moyens existent, aussi bien pour modifier le comportement du noyau que pour en obtenir des informations :

Usage de `printk()` (*`/include/linux/kernel.h`*) **:** Une possibilité, comme dans n'importe quel programme, est d'avoir recours à la commande `printk()`, variante de `printf()` pour usage dans le noyau. Elle peut être utilisée n'importe où, même dans un contexte d'interruption. Bien que l'usage général s'apparente très fort à `printf()`, une notion de *niveau de log* y est ajoutée. Le niveau est spécifié en début de chaîne (optionnellement, sans quoi une valeur par défaut est utilisée), comme dans l'exemple suivant :

```
printk("<0>Message urgent\n");
printk("<7>Message de debugging\n");
```

Mais il est recommandé d'utiliser à la place les macros `KERN_xxx` définies dans *include/linux/kernel.h*, on écrit alors :

```
printk(KERN_EMERG "Message urgent\n");
printk(KERN_DEBUG "Message de debugging\n");
```

Reste ensuite à déterminer le niveau de log minimal qui suscite un affichage sur le terminal. Si l'on souhaite que tous les messages arrivent sur le terminal, on écrira simplement (en tant que root) :

```
echo 8 > /proc/sys/kernel/printk
```

Le fonctionnement de cette commande sera expliqué plus tard dans cette section.

Dans tous les cas, les messages du noyau sont pris en charge par les démons `klogd` et `syslogd`. S'ils sont en fonctionnement, et selon leur configuration, ils enverront les messages vers un ou plusieurs fichiers de journal, la destination habituelle étant :

```
/var/log/messages
```

Afin d'éviter d'encombrer les journaux système avec une grande quantité de messages qui ne seront pas lus, on souhaite généralement être capable de désactiver en une fois tous les messages de debugging. Cela peut se faire facilement, en remplaçant l'usage de `printk` par une macro `PDEBUG` par exemple, définie comme suit :

```
#undef PDEBUG /*Désactive la macro, si elle a été définie ailleurs*/
#ifdef DEBUG_SOMETHING
# define PDEBUG(fmt, args...) printk( KERN_DEBUG "something: " fmt,##args)
#else
# define PDEBUG(fmt, args...)
#endif
```

Si l'on place ce code en haut d'un fichier `something.c`, et que l'on code les messages de debugging en utilisant `PDEBUG` plutôt que `printk`, alors on activera ces messages simplement en plaçant quelque part un `#define DEBUG_SOMETHING`, ou en donnant au compilateur l'option `-DDEBUG_SOMETHING`. Un bon texte sur les méthodes de debugging dans le noyau Linux (dont est inspiré cet exemple), peut être trouvé au chapitre 4 de [2].

Le système de fichiers `procfs` (*`/include/linux/proc_fs.h`*) **:** Alors que l'usage des journaux système permet d'obtenir une information sur le fonctionnement dynamique du noyau, le système de fichiers `procfs` permet d'obtenir *l'état* de différentes parties du système. Par exemple `/proc/net/arp` permet de visualiser la table `ARP`.

Ci-dessous est présenté le code que l'on utiliserait pour créer un fichier `/proc/net/mydir/myfile`,
qui permettrait d'afficher le contenu d'une structure `useful_data` :

```
1  static int myfile_seq_show(struct seq_file *s, void *v)
2  {
3          struct some_struct* useful_data=s->private;
4          seq_printf(s,"Contenu de useful_data : ");
5          ...
6          return 0;
7  }
8
9  static int myfile_open(struct inode *inode, struct file* file)
10 {
11         struct proc_dir_entry* pde=PDE(inode);
12         return single_open(file, myfile_seq_show,pde->data);
13 }
14
15 static struct file_operations myfile_ops = {
16         .open     = myfile_open,
17         .read     = seq_read,
18         .llseek   = seq_lseek,
19         .release  = single_release
20 };
21
22 /*Création d'une entrée dans /proc/net*/
23 struct proc_dir_entry* mydir = proc_mkdir("mydir",proc_net);
24 struct proc_dir_entry* myfile=create_proc_entry("myfile",0444,mydir);
25 if (myfile) {
26                 myfile->proc_fops=&myfile_ops;
27                 myfile->data=useful_data;
28 }
```

La première étape est de créer le répertoire `mydir` (ligne 23). Le second argument de la fonction est un pointeur vers le `struct proc_dir_entry` du répertoire parent. Ici il s'agit de `proc_net`, qui est une variable globale. Ensuite le fichier `myfile` est créé, avec les permission `0444` (lecture seule pour tout le monde). Ensuite il faut attacher au fichier la structure de description d'opérations, `struct file_operations`. Puisque l'on travaille avec l'abstraction d'un système de fichiers, les données seront accédées par des appels systèmes tels que `read` ou `seek`. La structure `myfile_ops` détermine quelle fonction sera exécutée dans chaque cas. Notons qu'il ne s'agit pas ici, comme dans un système de fichiers 'normal', d'aller chercher des données sur un support de mémoire, mais plutôt de les produire à la demande. Cette production se faisait jusqu'il y a peu en utilisant uniquement l'interface `procfs`. Comme elle s'est avérée lourde à manipuler (les déplacements à l'intérieur du fichier –`seek`– posaient des difficultés), l'interface `seq_file` (*/include/linux/seq_file.h*) a été incorporée au noyau 2.6.

La structure `file_operations` est en réalité beaucoup plus grande qu'elle n'y paraît dans l'exemple, mais seuls les champs indiqués sont nécessaires dans ce contexte, donc ils sont les seuls à devoir être initialisés. Grâce à une extension spécifique au compilateur C de GNU, les autres champs sont automatiquement initalisés à NULL. Parmi les champs initialisés, on laisse l'interface `seq_file` s'occuper de trois fonctions, nous laissant seulement à programmer une fonction `open`, et une autre `show`, qui est responsable d'afficher le contenu voulu.

Traçons maintenant le chemin suivi par le pointeur `useful_data` (`myfile->data` est un

`void*`), car il s'apparente quelque peu à un jeux de piste, le but du jeu étant d'avoir accès à ce pointeur dans la fonction `myfile_seq_show` :

- La structure `proc_dir_entry` possède un champ `data` dans lequel le programmeur peut mettre ce qu'il souhaite. Nous y mettons donc le pointeur `useful_data`.
- La fonction `myfile_open` reçoit un inode, dont on peut extraire le `proc_dir_entry` original, donc la donnée. C'est le moment de spécifier que `seq_read` devra faire appel à `myfile_seq_show` pour l'affichage des données. On lui passe aussi le pointeur, *qu'il copie dans le champ* `private` de la structure `seq_file` qu'il passera à `myfile_seq_show`.
- Nous pouvons maintenant récupérer notre pointeur dans la fonction `myfile_seq_show` ! (ligne 3)

Dans un souci de brièveté, nous n'avons décrit ici que les éléments effectivement utilisés dans le cadre de l'implémentation de `shim6`. Plus de détails sur l'interface `procfs` pourront être trouvés dans [27] ; l'usage de `seq_file` est introduit dans [16].

L'interface sysctl (*include/linux/sysctl.h*) : Grâce à cette interface, on peut agir sur le comportement du noyau. C'est ainsi que par exemple on active le forwarding IPv6 par :

```
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Un autre exemple était donné ci-dessus pour la configuration de `printk`.

Puisque nous agissons ici aussi sur un système de fichiers, il faut insérer une entrée dans l'arborescence `/proc/sys`. Le contenu de la racine (la racine pour le système sysctl est `/proc/sys`, pas `/proc`) est spécifié dans `kernel/sysctl.c`. En voici un extrait :

```
static ctl_table root_table[] = {
        {
                .ctl_name       = CTL_KERN,
                .procname       = "kernel",
                .mode           = 0555,
                .child          = kern_table,
        },
        /* ... */
#ifdef CONFIG_NET
        {
                .ctl_name       = CTL_NET,
                .procname       = "net",
                .mode           = 0555,
                .child          = net_table,
        },
#endif
        /* ... */
        {
                .ctl_name       = CTL_DEV,
                .procname       = "dev",
                .mode           = 0555,
                .child          = dev_table,
        },

        { .ctl_name = 0 }
};
```

Si l'on observe l'entrée correspondant au répertoire `net` (qui sera donc `/proc/sys/net`), on voit que l'on fournit les paramètres suivants :

- `ctl_name=CTL_NET` : identifiant unique donné au répertoire `net`
- `procname="net"` : le nom du répertoire, tel qu'il apparaîtra dans `/proc/sys`
- `mode=0555` : Le mode d'accès au fichier, ici 555, donc lecture et exécution autorisés pour tous. On donne ce type de permission parce qu'il s'agit d'un répertoire.

– `child` : Un pointeur vers un tableau d'objets `ctl_table` terminé, comme c'est le cas ici, par un élément `{.ctl_name=0}`, qui sert de marqueur de fin de tableau. Les éléments du tableau spécifié dans le champ `child` correspondent donc au contenu du répertoire. Si l'entrée était un fichier plutôt qu'un répertoire, alors ce champ vaudrait `NULL`, et ne serait donc pas indiqué explicitement (notation spécifique à gcc : initialisation explicite partielle, comme pour la structure `file_operations`, voir plus haut).

Pour poursuivre notre exemple, suivons la trace du répertoire `net`.

Le tableau `net_table` est défini dans *net/sysctl_net.c*. Etrangement, on n'y trouve pas la définition du répertoire IPv6, pourtant présent dans l'arborescence. La raison est qu'IPv6 peut être compilé comme module et utilise un autre moyen pour enregistrer ses fonctions `sysctl`. On trouve sa table dans *net/ipv6/sysctl_net_ipv6.c*, celle-ci étant ajoutée dynamiquement à l'arborescence au moyen de la fonction `register_sysctl_table()`. Si l'on retire le module, il faudra alors retirer l'arborescence IPv6 par un appel à `unregister_sysctl_table()`.

Voyons maintenant comment est défini un fichier : nous prendrons pour ça l'exemple simple du fichier `/proc/sys/net/ipv6/icmp/ratelimit` (*net/ipv6/icmp.c*) :

```
static int sysctl_icmpv6_time = 1*HZ;

ctl_table ipv6_icmp_table[] = {
        {
                .ctl_name       = NET_IPV6_ICMP_RATELIMIT,
                .procname       = "ratelimit",
                .data           = &sysctl_icmpv6_time,
                .maxlen         = sizeof(int),
                .mode           = 0644,
                .proc_handler   = &proc_dointvec
        },
        { .ctl_name = 0 },
};
```

Puisque c'est un fichier, le champ `child` n'est pas défini, et prend donc par défaut la valeur `NULL`. Par contre l'on définit :

– `data` : l'adresse de la donnée à laquelle l'interface sysctl permettra d'accéder.

– `maxlen` : Longueur maximale inscriptible à partir de l'adresse `data`. Ce champ est d'une importance fondamentale, parce qu'on laisse l'utilisateur écrire dans le noyau, il faut donc contrôler de manière stricte tout ce qui provient de lui. En programmation noyau, l'hypothèse est toujours faite que l'utilisateur est malintentionné (ou maladroit).

– `proc_handler` : Un pointeur vers la fonction qu'il faudra appeler pour toute lecture ou écriture sur le fichier `ratelimit`. Des fonctions prédéfinies existent pour cela, selon le type de données que l'on souhaite configurer. Comme dans cet exemple c'est un entier qui est modifié, la fonction `proc_dointvec` sera utilisée. Des fonctions adaptées pour d'autres types de données peuvent être trouvées dans *include/linux/sysctl.h*. Elles sont implémentées dans *kernel/sysctl.c*.

Grâce à tout ceci, on dispose maintenant d'un fichier `ratelimit` qui permet de modifier la valeur de la variable `sysctl_icmpv6_time` dans le noyau en cours d'exécution. Par contre il faut être root pour pouvoir le faire, car les permissions d'accès sont 0644 (lecture/écriture pour root, lecture seule pour les autres).

Une description de l'interface sysctl est disponible dans [34].

2.2 Construction et envoi d'un paquet sur le réseau

2.2.1 Réservation de la mémoire

Lorsque l'on souhaite créer un nouveau paquet, il faut commencer par réserver de la place en mémoire pour l'accueillir. Idéalement, il est souhaitable de pouvoir prédire dès l'allocation de mémoire la quantité qui sera utilisée par le paquet. En effet, si à un moment donné on tombait à cours de ressources, il faudrait alors allouer un nouvel emplacement plus grand, et recopier toutes les données, ce qui est possible mais prend du temps. Pour cette raison, et comme le paquet ne restera de toute façon pas longtemps en mémoire, on préfère travailler avec des valeurs maximales. Par exemple on réserve toujours `MAX_HEADER` (*include/linux/netdevice.h*) octets pour l'en-tête liaison de données.

Un exemple simple de réservation de mémoire pour un paquet est le cas d'un segment TCP RESET (*net/ipv6/tcp_ipv6.c*) :

```
struct sk_buff* buff=alloc_skb(MAX_HEADER+sizeof(struct ipv6hdr)+
        sizeof(struct tcphdr),GFP_ATOMIC);
```

Cette instruction réserve à la fois de la mémoire pour une structure `struct sk_buff` et pour le segment RST que l'on construira ensuite. Le premier argument donne la taille à réserver *pour le paquet* (la taille de la structure est connue). Pour un segment RST, cette taille est la taille maximale occupée par une en-tête de couche 2, à laquelle on ajoute la taille des en-têtes IPv6 (sans options) et TCP.

Le second argument détermine si l'allocation doit être atomique (argument `GFP_ATOMIC`) ou peut être interrompue (mettre le processus en état `sleeping`, `GFP_KERNEL`). Comme on l'a vu précédemment seuls les contextes utilisateurs peuvent dormir. Un segment RST est envoyé en réponse à un segment reçu, donc dans l'interruption software `NET_RX`, c'est pourquoi `GFP_ATOMIC` *doit* être utilisé. `GFP` signifie *get free page*. De fait, ce qui provoquerait la mise en attente d'un processus serait la copie sur disque (swap) d'une page de mémoire permettant d'obtenir une page libre. Avec `GFP_ATOMIC`, on impose de trouver une page libre dans la mémoire directement. Si elle n'est pas trouvée, comme on ne peut utiliser le swap, la fonction d'allocation de mémoire retourne une erreur.

2.2.2 Manipulation d'un paquet

Une fois la mémoire allouée, il va falloir la manipuler pour construire le paquet. Cette manipulation a de particulier qu'à chaque couche de la pile réseau, des champs sont susceptibles d'être ajoutés aussi bien en début qu'en fin de paquet. Pour cette raison la mémoire du paquet est divisée en une zone appelée *headroom* (en-tête), une autre appelée *data* et finalement une zone de queue, *tailroom*. Ces zones sont délimitées par des pointeurs de la structure `sk_buff` (voir figures 2.2 et 2.3). Cette même structure contient aussi des pointeurs vers chacune des en-têtes transport (`h`), réseau (`nh`), ou liaison de données (`mac`). Puisque plusieurs protocoles existent pour chacune des couches, une `union` est utilisée pour éviter de devoir faire des castings. Par exemple `skb->nh.ipv6h` et `skb->nh.raw` contiennent *le même* pointeur, mais le type qui lui est attaché est différent. Ceci est particulièrement intéressant pour se déplacer dans les paquets en utilisant l'arithmétique de pointeurs :

- `skb->nh.ipv6h+1` pointe vers le premier octet qui suit l'en-tête IPv6 (utile pour lire l'en-tête suivante).
- `skb->nh.raw+1` pointe vers le second octet de l'en-tête IPv6.

Les pointeurs `head, data, tail, end` ne devraient jamais être manipulés directement. L'interface `include/linux/skbuff.h` fournit pour cela une batterie de fonctions permettant d'ajuster les champs de la structure de manière adéquate.

```
struct sk_buff
{
        ...
        union {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                struct ipv6hdr  *ipv6h;
                unsigned char   *raw;
        } h;

        union {
                struct iphdr    *iph;
                struct ipv6hdr  *ipv6h;
                struct arphdr   *arph;
                unsigned char   *raw;
        } nh;

        union {
                unsigned char   *raw;
        } mac;

        struct dst_entry        *dst;
        ...
        unsigned int            len;
        ...
        unsigned int            *head,*data,*tail,*end;
}
```

FIG. 2.2 – Parties de la structure de manipulation de paquets `struct sk_buff`

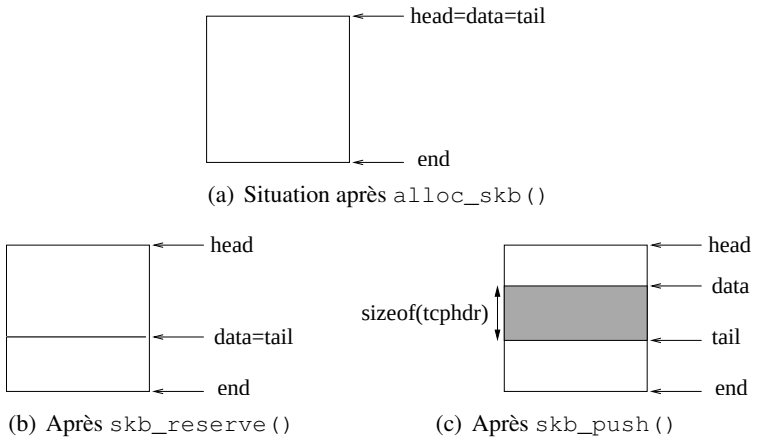


FIG. 2.3 – Représentation d'un paquet

L'une de ces fonctions est `skb_reserve(struct sk_buff skb, unsigned int len)`. Elle sert à réserver de l'espace *headroom* en décalant la zone de données vers le bas. Cette opération ne peut se faire que sur un paquet vide (juste après l'allocation), puisque l'on décale le pointeur `data` sans recopier la zone pointée.

L'espace *headroom* est maintenant alloué définitivement (plus exactement, l'allocation devient définitive lorsque l'on écrit dans la zone de données). S'il la zone d'en-tête devait par la suite s'avérer trop petite, il faudrait allouer un nouveau tampon plus grand, à l'aide d'un appel à la fonction `skb_realloc_headroom(skb,newheadroom)`. Celle-ci renvoie un pointeur vers le nouveau tampon créé, à l'image du premier mais possédant la nouvelle taille d'en-tête.

Tout ceci étant fait, nous ne pouvons pas encore écrire dans le tampon, car l'on n'écrit que dans la zone *data*, les autres sont toujours considérées comme de l'espace libre. Il faut donc réserver de la mémoire dans la zone de données, en avançant sur la partie d'en-tête ou celle de queue. Le premier cas s'obtient par `skb_push(skb,len)`, le second par `skb_put(skb,len)`. Si l'on reprend l'exemple de l'envoi d'un segment TCP RST, le code suivant est utilisé, et mène au résultat présenté à la figure 2.3(c) :

```
t1=(struct tcphdr*)skb_push(buff,sizeof(struct tcphdr));
```

Il est acceptable d'allouer trop de mémoire pour un paquet, ou de réserver trop d'en-tête. Par contre lorsque l'on réserve de l'espace de données avec `skb_push()` ou `skb_put()`, alors il faut fournir la taille exacte que l'on souhaite utiliser. Cette fois chaque couche fait grandir la zone de données selon ses besoins exacts. La raison est que ces fonctions de réservation en zone *data* augmentent également le champ `len` de `sk_buff`, qui servira finalement à déterminer combien de bits il faut envoyer à l'interface de transmission.

Maintenant nous pouvons écrire. Souvent la première chose qui est faite est de mettre toute la mémoire à zéro, puisque les fonctions de réservation de mémoire dans le noyau n'effacent jamais le contenu de la zone allouée (cela est parfois inutile, si bien qu'on laisse le demandeur s'en charger en cas de nécessité). L'instruction est `memset(t1,0,sizeof(*t1))`. Cette fonction existe aussi en espace utilisateur (*string.h*, et dans le noyau *include/linux/string.h*).

Finalement, il faut remplir le paquet. Dans notre exemple TCP, nous devons donner une valeur à chacun des champs. Mais se pose alors le problème de l'ordre des bits et octets. C'est le sujet de la section suivante.

2.2.3 Attribution de valeurs aux champs d'un paquet

Une fois que l'on dispose d'une zone de mémoire de taille appropriée, et d'un pointeur vers le début de celle-ci, on lui applique une structure (par casting) qui permettra de placer chaque champ au bon endroit (avec une granularité aussi fine qu'un seul bit, ainsi que l'exigent les protocoles réseau). Malheureusement, on est alors gêné par le fait que certaines machines fonctionnent en mode *little-endian* (processeurs Intel) et d'autres en mode *big-endian* (processeurs Motorola).

> En stockage *big-endian*, les bits de poids fort ('big end') sont placés en premier. C'est la notation intuitive, telle qu'un humain l'écrirait sur un papier. C'est aussi l'ordre de bits employés pour les transmissions réseau. Dans un système *little-endian*, les bits de poids faible ('little end'), sont placés en premier lieu.

Un exemple des deux modes de représentation est donné à la figure 2.4 (exemple tiré de [24]). On observe sur cet exemple que non seulement l'ordre des bits diffère, mais aussi celui des octets.

Voyons les implications de ceci pour la réalisation d'une structure d'en-tête TCP, en prenant appui sur la figure 2.5 :

FIG. 2.4 – Méthodes de représentation des nombres ($0x0a0b0c0d$)

```
struct tcphdr {
        __u16   source;
        __u16   dest;
        __u32   seq;
        __u32   ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
                res1:4,
                cwr:1,
                ece:1,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1;
#else
#error  "Adjust your <asm/byteorder.h>"
#error  "defines"
#endif
        __u16   window;
        __u16   check;
        __u16   urg_ptr;
};
```

FIG. 2.5 – Représentation de l'en-tête TCP, *include/linux/tcp.h*

D'abord il faut tenir compte du fait que les octets sont correctement traduits au moment du passage sur l'interface. Cela signifie que si l'on attribue la valeur 28 à un caractère (un octet), sur une machine little-endian, les bits de cet octet seront inversés au moment du passage sur l'interface, et c'est bien la valeur 28 qui sera comprise par le correspondant.

Si les octets sont correctement traduits, il n'en est pas de même des groupes d'octets. Les octets sont correctement traduits parce qu'il s'agit de l'unité d'adressage. Par contre pour les groupes d'octets, il est impossible de savoir, au moment de la copie sur l'interface de sortie, comment sont structurés les octets. Le noyau ne voit à ce moment là qu'un bloc d'octets qu'il ne peut copier que dans l'ordre où ils apparaissent. Ainsi, lors d'un envoi de paquet, il faudra toujours inverser les groupes d'octets (`short` et `int`), si l'on est sur une machine *little-endian*. Linux permet de faire abstraction du type de machine en appelant dans tous les cas une macro, qui adopte le comportement adéquat en fonction du type de machine. Quatre macros sont définies (*include/linux/byteorder/generic.h*) :

- `htonl(x)` : convertit un entier `x` de 32 bits depuis l'ordre d'octets de la machine vers celui du réseau. Par exemple à l'émission d'un paquet, on écrira un numéro de séquence de 123 par `t1->seq=htonl(123)`
- `ntohl(x)` : le contraire de la précédente. Pour notre exemple le receveur exécutera l'instruction `seq=ntohl(t1->seq)`
- `htons(x)` et `ntohs(x)` : ces fonctions jouent le même rôle, mais pour des nombres de 16 bits (`short`).

Reste à fixer ce qui se produit lorsque l'on subdivise un entier de deux ou quatre octets en bits ou groupes de bits (cas des drapeaux TCP) : dans ce cas les octets sont toujours stockés dans l'ordre que l'on a imposé, il faut donc considérer individuellement chaque octet, et placer les octets dans l'ordre

du réseau (c'est simplement l'ordre dans lequel les champs apparaissent dans les standards). Quant aux bits, la situation la plus simple est celle d'une machine *big endian*, où l'on place les bits dans l'ordre du réseau, ils sont stockés et recopiés tels quels. Si la machine est *little endian*, alors il faut inverser les bits. Elle les stockera dans l'ordre que l'on a imposé, mais les renversera au moment de la copie sur l'interface de sortie (comme n'importe quel octet), on arrive donc au bon ordre au moment de l'envoi.

Cette différence de traitement selon la machine se résout par des instructions pré-processeur `#if`, `#else`, `#endif`, comme à la figure 2.5. Par la suite on pourra remplir les champs sans se préoccuper désormais du type de machine.

2.2.4 Routage et transmission du paquet à la couche IPv6

La manière la plus simple de passer un paquet à la couche IPv6 est de faire appel à la fonction (*net/ipv6/ip6_output.c*) :

```
ip6_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl,
         struct ipv6_txoptions *opt, int ipfragok)
```

Elle prend en charge l'envoi de n'importe quel paquet `skb` sur base des informations fournies dans `fl` (*include/net/flow.h*). `ip6_xmit()` utilise la structure `flowi` pour obtenir les informations suivantes :

- les adresses IPv6 source et destination : `fl->fl6_src` et `fl->fl6_dst`. Nous verrons plus loin comment ces adresses sont déterminées.
- Le numéro de protocole : `fl->proto`. Ce numéro est l'une des valeurs `IPPROTO_xxx` définies dans *include/linux/in.h*. Si l'on fait usage d'extensions d'en-tête, alors le numéro d'extension est l'une des valeurs `IPPROTO_xxx` définies dans *include/linux/in6.h*.
- L'étiquette `flowlabel` : `fl->fl6_flowlabel`. Si on ne s'en sert pas, elle vaut habituellement zéro.

La structure `struct flowi` possède d'autres champs, mais ils ne sont pas utilisés par la fonction `ip6_xmit()`.

En plus de fournir la structure `flowi` à `ip6_xmit()`, il faut s'assurer avant l'appel que le pointeur `skb->dst` est défini. Un oubli aurait pour conséquence une 'kernel panic' !

Cette structure fournit l'information sur l'interface de sortie, et est obtenue par un appel à la fonction (*net/ipv6/ip6_output.c*)

```
int ip6_dst_lookup(struct sock *sk, struct dst_entry **dst,
       struct flowi *fl)
```

Cette dernière constitue en réalité l'interface avec la table de routage, et reçoit aussi ses informations du même `struct flowi* fl`. Le second argument est un double pointeur, ce qui permet à la fonction d'écrire son pointeur directement au bon endroit, si l'on fournit comme argument `&skb->dst`.

On remarque que les deux dernières fonctions citées ont pour premier argument `struct sock *sk`. Il est utilisé dans le cas où l'on envoie un paquet dans le contexte d'un socket. Si ce n'est pas le cas, cet argument peut sans problème prendre la valeur `NULL`.

La fonction `ip6_dst_lookup()` se charge aussi de fournir une adresse source *si elle n'a pas déjà été déterminée dans `fl`*, par un appel à (*net/ipv6/addrconf.c*)

```
int ipv6_get_saddr(struct dst_entry *dst,
                   struct in6_addr *daddr, struct in6_addr *saddr)
```

Cette fonction applique l'algorithme proposé dans le RFC3484 [20] pour déterminer `daddr`, sur base de `saddr` et `dst`, ce dernier étant optionnel (il est permis de fournir un argument `NULL`).

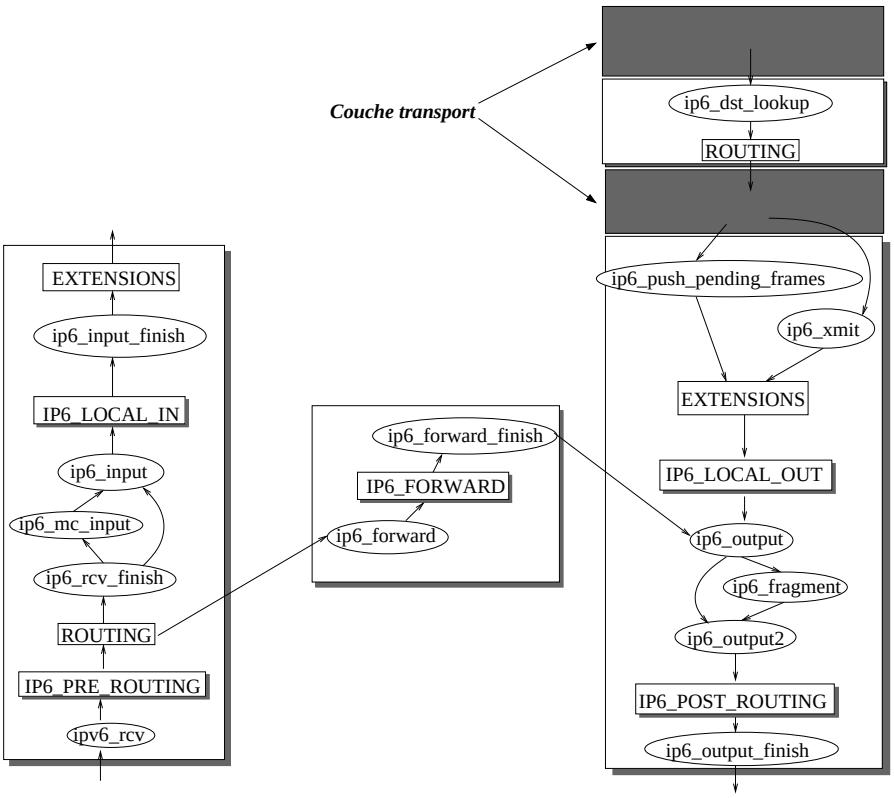


FIG. 2.6 – Trajet d'un paquet à travers la couche IPv6

Il faut noter que la fonction `ip6_xmit()`, que nous avons décrite, n'est pas la seule manière de transmettre des paquets à la couche IPv6. En particulier, si le paquet à envoyer est attaché à un socket (ce qui en réalité est presque toujours le cas), alors on utilisera plutôt la fonction suivante :

```
ip6_push_pending_frames(struct sock* sk)
```

Cette fonction ne sera cependant pas décrite ici car elle n'est pas utilisée dans `shim6` (qui travaille sans sockets).

2.3 Schéma général de traversée de la couche IPv6

La figure 2.6 décrit le chemin poursuivi par un paquet à travers la couche IPv6, telle qu'elle est implémentée dans le noyau 2.6.15. Cette figure est inspirée et mise à jour (l'implémentation d'IPv6 dans le noyau Linux est encore en mouvement) de la figure 23-3 de [4]. Le schéma doit se lire comme suit :

- Les boîtes ovales représentent des fonctions
- Les rectangles ombrés sont des branchements vers NetFilter (hooks). En chacun de ces points NetFilter est appelé pour appliquer ses règles (filtrage de paquets, transformations ou statistiques réseau), en fonction de l'étape de traitement.
- Les autres boîtes mettent en évidence les endroits auxquels sont traités le routage et les extensions d'en-tête.

La partie gauche du schéma représente la réception de paquets, tandis que la partie droite représente la transmission. Puisque l'implémentation d'IPv6 est capable de jouer la fonction de routeur,

une jonction existe entre ces deux tours.

Au-dessus de la couche IP se trouve la couche transport. Nous l'avons représentée sur la partie droite pour mettre en évidence le saut temporaire qui est fait vers IP, afin d'effectuer le routage et l'attribution d'adresse source (le cas échéant). Ceci est nécessaire pour que TCP puisse faire son calcul de checksum, qui nécessite la connaissance des adresses IP. Ainsi, chaque couche transport est responsable d'appeler `ip6_dst_lookup` avant de faire descendre le paquet par `ip6_xmit` ou `ip6_push_pending_frames`.

Du côté de la réception, les paquets à destination de l'hôte arrivent après filtrage et traitement multicast (`ip6_mc_input`) dans la fonction `ip6_input_finish`. Là, l'envoi à l'instance de protocole transport concernée est effectué. Pour permettre cette opération, chaque protocole transport doit définir une structure `struct inet6_protocol`, et l'enregistrer au moyen de la fonction

```
inet6_add_protocol(struct inet6_protocol, unsigned char num)
```

Le second argument est le code de protocole. Munie de cette information, `ip6_input_finish` peut retrouver la structure adéquate sur base du code 'next header' et exécuter la routine de traitement pointée par le champ `handler` de la structure `inet6_protocol`. Le paquet est alors entre les mains de la couche transport.

Alors qu'anciennement (le caractère ancien est à considérer au sens informatique) les extensions d'en-tête étaient traitées par une fonction `ipv6_parse_exthdrs`, elles le sont maintenant (sauf pour l'option *hop by hop*) comme des protocoles, si bien qu'une structure `inet6_protocol` est définie pour chacune de ces extensions.

2.4 L'autoconfiguration d'adresses (*net/ipv6/addrconf.c*)

Le principe de l'autoconfiguration d'adresses IPv6 a été décrit à la section 1.2.3. Tout ce mécanisme, ainsi que la configuration manuelle, est implémenté dans le fichier *net/ipv6/addrconf.c*. On y trouve une table de hashage, `inet6_addr_lst`, qui reprend l'ensemble des adresses IPv6 unicast de l'équipement. Les objets de cette table sont des structures `inet6_ifaddr`, que nous reprenons ci-après :

```
struct inet6_ifaddr
{
        struct in6_addr         addr;
        __u32                   prefix_len;

        __u32                   valid_lft;
        __u32                   prefered_lft;
        unsigned long           cstamp; /* created timestamp */
        unsigned long           tstamp; /* updated timestamp */
        atomic_t                refcnt;
        spinlock_t              lock;

        __u8                    probes;
        __u8                    flags;

        __u16                   scope;

        struct timer_list       timer;

        struct inet6_dev        *idev;
        struct rt6_info         *rt;

        struct inet6_ifaddr     *lst_next;      /* next addr in addr_lst */
        struct inet6_ifaddr     *if_next;       /* next addr in inet6_dev */
```

```
#ifdef CONFIG_IPV6_PRIVACY
        struct inet6_ifaddr     *tmp_next;      /* next addr in
                                                   tempaddr_lst */
        struct inet6_ifaddr     *ifpub;
        int                     regen_count;
#endif

        int                     dead;
};
```

Il est intéressant de voir que cette structure regroupe plusieurs des concepts de programmation noyau développés à la section 2.1. On y trouve notamment un verrou de type spinlock ; un compteur de références (ici, implémenté manuellement) et plusieurs listes simplement chaînées, aussi implémentées manuellement. En particulier `lst_next` est la liste de collisions de la table de hashage. La fonction de hashage utilisée est `ipv6_addr_hash(struct in6_addr*)`.

On trouve dans cette structure les champs `valid_lft` et `prefered_lft`[1], qui correspondent aux deux durées de vie d'une adresse.

Le champ `dead` est mis à un lorsqu'une adresse est supprimée de la table de hashage (par un appel à `ipv6_del_addr`). L'usage de ce champ est lié à celui du compteur de références : si quelqu'un utilise l'adresse au moment de sa suppression, alors le compteur de références est seulement descendu, et le drapeau `dead` mis à un. Par la suite, les fonctions qui se servaient de l'adresse 'mise à mort' cessent de suite de l'utiliser.

Les drapeaux (champ `flags`) indiquent l'état d'une adresse. Ils peuvent prendre les valeurs suivantes :

- `IFA_F_SECONDARY` ou `IFA_F_TEMPORARY` : l'adresse IPv6 est temporaire (privacy extensions).
- `IFA_F_DEPRECATED` : lorsque l'âge d'une adresse dépasse `prefered_lft`, celle-ci reçoit ce drapeau. Une adresse périmée ne devrait si possible pas être utilisée.
- `IFA_F_TENTATIVE` :ce drapeau est désactivé lorsque l'algorithme DAD se termine avec succès. Il est activé pour n'importe quelle adresse insérée dans la table (`ipv6_add_addr`). Par conséquent toute adresse démarre sa vie dans la table de hashage en tant que 'tentative address'.
- `IFA_F_PERMANENT` : ce drapeau est activé pour une adresse configurée manuellement.

La suppression ou l'ajout d'une adresse IPv6 suscite l'appel de la fonction

`ipv6_ifa_notify(int event,struct inet6_ifaddr *ifp),`

où `event` peut être `RTM_NEWADDR` ou `RTM_DELADDR`. Le second argument est l'adresse concernée. Cette fonction exécute les opérations rendues nécessaires par l'ajout/suppression d'une adresse.

2.5 Conclusion

Dans ce chapitre nous avons vu diverses techniques utilisées dans le noyau Linux ainsi que certains aspects liés plus spécifiquement à l'architecture réseau.

Ainsi, nous avons vu que toute structure partagée doit être protégée adéquatement, à la fois par des verrous et des compteurs de référence, l'un ne dispensant pas de l'autre. De plus, à chaque usage de la structure, il faut réfléchir si la portion de code dans laquelle on travaille est un contexte utilisateur, une interruption software ou hardware. Le fruit de cette réflexion est un choix approprié du type de verrou :

[1] la faute d'orthographe provient du code, il est donc de bon usage d'adopter un comportement de moine copiste, par respect pour l'auteur, ou plus concrètement pour permettre la compilation du noyau...

- un sémaphore si la section critique n'est manipulée qu'à l'intérieur d'un contexte utilisateur ;
- des verrous `spin_lock_bh`, `spin_unlock_bh` si la section critique peut être exécutée aussi bien en contexte utilisateur qu'en contexte d'interruption. La variante `_bh` des spinlocks désactive les interruptions software durant l'accès à la section critique ;
- des verrous `spin_lock` si l'on travaille uniquement en interruption software.

Les méthodes de verrouillages en contexte d'interruption hardware n'ont pas été vues ici car `shim6` n'y intervient pas.

Dans le cadre de l'architecture réseau, on retiendra qu'un envoi de paquet se fait généralement en contexte utilisateur, tandis qu'une réception ou un forwarding se font en contexte d'interruption software.

Finalement, l'architecture réseau a été brièvement décrite. On y a vu le système d'autoconfiguration d'adresses, dont les structures de stockage des adresses s'avéreront d'un intérêt fondamental pour la méthode de gestion de localisateurs que nous proposons au chapitre suivant. Par ailleurs, précisons d'ores et déjà que le schéma de traversée des paquets à travers la couche IPv6 (fig. 2.6) nous servira de base pour décider où greffer les composantes de shim6.

Chapitre 3

Shim6 : détails et implémentation du protocole

Dans les premier et second chapitre, nous avons vu respectivement le protocole `shim6` et les méthodes de programmation dans le noyau Linux.

Dans ce chapitre nous associons les deux, en décrivant comment nous avons implémenté `shim6` dans le noyau. Ce sera aussi l'occasion d'étudier le protocole `shim6` plus en détails, tel qu'il est défini maintenant dans [32].

Alors que le chapitre 1 laissait certaines questions ouvertes, celui-ci s'attache à une description beaucoup plus concrète de la solution finale choisie à l'IETF, évolution qui reflète aussi le chemin parcouru par le groupe de travail `shim6` parallèlement à la réalisation de ce mémoire.

Du point de vue de l'implémentation, comme expliqué dans l'introduction, le but de ce mémoire était de réaliser un sous-ensemble fonctionnel des éléments principaux de `shim6`, sans supporter toutes les options. En particulier, la principale nouveauté du protocole réside dans le remplacement des localisateurs d'un hôte suite à une panne, sans bloquer les échanges en cours. C'est donc là-dessus que nous nous focalisons, ainsi que sur la méthode de détection de joignabilité et le choix d'une nouvelle paire de localisateurs, qui sera décrite au chapitre suivant.

A cause du volume du document définissant le protocole, ainsi que sa définition progressive au fil de cette année, une implémentation complète du protocole sortait du cadre d'un mémoire. Par ailleurs d'autres travaux ont été déjà réalisés, qu'il est inutile de dupliquer. Ainsi, nous avons choisi de ne pas implémenter la sécurisation des localisateurs par HBA, ce travail ayant déjà été fait par F. Dupont (disponible sur *http ://ops.ietf.org/multi6/francis-hba.tar.gz*).

Considérant tout ceci, nous avons décidé d'implémenter les fonctions suivantes du protocole :

- Déclenchement d'un échange d'initialisation dès le premier envoi de paquet vers un hôte pour lequel on ne possède pas encore de contexte (cette partie est décrite à la section 3.1.1).
- L'échange d'initialisation est du type simultané (les deux hôtes commencent à s'échanger des paquets en même temps, c'est le cas des échanges client/serveur). L'autre type non implémenté s'appliquerait au cas où seul l'un des deux hôtes impliqués souhaite initialiser l'échange, par exemple parce que sa politique de déclenchement diffère. C'est le sujet de la section 3.1.3.
- La détection de joignabilité et procédure d'exploration de paires de localisateurs fait l'objet du chapitre 4. Ceci est défini dans un autre document [7], et est complètement supporté dans notre implémentation.
- Une fois qu'une nouvelle paire de localisateurs a été trouvée, il faut effectuer une traduction de chaque paquet sortant à destination du correspondant, ou reçu de la part de celui-ci. Cette

traduction est supportée, car il s'agit là du point central du protocole `shim6`. La procédure de traduction d'adresses après rehoming est décrite à la section 3.3.

D'autre part, nous avons choisi de ne pas supporter dans cette première implémentation les options suivantes :

- L'émission d'une mise à jour de la liste de localisateurs, lorsqu'une nouvelle adresse IPv6 devient disponible, ou au contraire devient invalide. Ces événements sont rares en pratique et dans la plupart des cas, il est suffisant d'attendre l'établissement de session `shim6` suivant pour obtenir une liste de localisateurs à jour.
- Les méthodes de récupération de contexte, dans le cas où seul l'un des deux correspondants a supprimé son contexte, l'autre essayant toujours de l'utiliser : cette fonction rend le protocole plus robuste, mais s'avérera rarement utile, car un contexte est supprimé (dans notre implémentation) après 10 minutes d'absence de trafic, et ce chez chaque hôte. Ils supprimeront donc leurs contexte de manière simultanée ou presque.
- Les fonctions de sécurité ne sont pas non plus supportées. La nouveauté principale dans ce domaine est le mécanisme HBA, qui a déjà été testé par F. Dupont. Les autres mécanismes (initialisation avec nonces) sont d'usage courant en réseau et ne nécessitent pas de test particulier. Par contre il est évident que tous les mécanismes de sécurité devront être implémentés plus tard si l'on souhaite utiliser ce prototype en dehors d'un laboratoire.
- Le FII (forked instance identifier) : le protocole prévoit de pouvoir créer un contexte à partir d'un autre déjà existant (forking), en lui donnant les mêmes ULIDs, mais des localisateurs différents. Le système de forking doit être contrôlé par les applications, ce qui n'est pas possible actuellement car il n'existe pas d'API `shim6`.

Finalement, les hypothèses suivantes ont été faites :

- Les messages ICMP sont simplement ignorés par `shim6`. La plupart du temps c'est le comportement correct, car ces messages véhiculent de l'information sur les localisateurs, et ne doivent pas être transmis aux couches supérieures. Mais si un message ICMP devait être transmis aux couches supérieures après un rehoming, il ne serait pas traduit en ULIDs. Nous supposons que ce cas ne se produit pas (section 3.7).
- `shim6` est la seule extension d'en-tête IPv6 utilisée. La conception du prototype a été faite en tenant compte des extensions d'en-tête, mais son usage superposé à l'emploi de telles extensions n'a pas été testé. C'est pourquoi nous ne pouvons fournir de garanties à ce sujet.

La structure du chapitre est articulée autour du déroulement d'une connexion TCP, depuis l'émission d'un paquet de couche transport, déclenchant la création d'un contexte initial, jusqu'à l'effacement d'un contexte devenu inutilisé. A chaque étape, la théorie associée au protocole sera présentée, suivie d'un paragraphe ou d'une section décrivant comment cette partie précise est implémentée.

3.1 L'initialisation d'un contexte shim6

3.1.1 Déclenchement d'un échange d'initialisation

La première question à se poser dans la mise en oeuvre du mécanisme de multihoming est celle du moment où l'on décide de l'activer. L'implémentation que nous proposons déclenche l'initialisation dès le démarrage d'un échange avec un nouveau correspondant. Par 'nouveau', on entend un correspondant identifié par une adresse (ULID) qui ne correspond pas encore à un contexte. A ce stade il faut se rappeler que la couche `shim6` intercepte tous les paquets en provenance des couches supérieures sans les rattacher aucunement à une application ni même un protocole transport. Observons sur un exemple le comportement de `shim6` vis à vis des débuts de connexion :

– Un hôte démarre une connexion TCP avec le couple d'identifiants <A,B>. Le premier de ces paquets provoque chez l'émetteur la création d'un contexte `shim6` associé à ce couple, ainsi que le démarrage de l'échange d'initialisation du protocole. Concrètement, le premier segment TCP d'un échange et le premier paquet `shim6` d'initialisation de contexte partent en même temps. Plus précisément, le paquet d'initialisation de `shim6` part juste *après* le premier segment TCP, ceci afin de favoriser le mode d'initialisation simultanée, comme nous le verrons à la section 3.1.3.
– Les paquets suivants de cette même connexion seront par la suite rattachés et traités par le contexte `shim6` nouvellement créé. Celui-ci n'aura cependant aucun effet sur les échanges tant que son initialisation n'est pas terminée.
– Ensuite ce même hôte démarre un échange UDP avec le même couple <A,B>. `Shim6` ne voit pas la différence avec les paquets TCP et rattache les segments UDP au contexte déjà existant.
– Maintenant un troisième échange prend place, avec cette fois le couple <__,B> (adresse source indéterminée). Cet échange est encore attaché au même contexte, avec comme particularité le fait que `shim6` attribue lui-même l'ULID source (plutôt que le mécanisme RFC3484 [20]). Cet ULID sera donc A.

Le dernier point de cet exemple mérite quelques commentaires, puisqu'il s'agit d'un choix de design que nous avons fait pour le prototype implémenté. En l'absence de `shim6`, une application, souvent plus intéressée par l'adresse destination que par celle de la source, peut déléguer à la couche IP le choix de cette dernière. Un algorithme par défaut a été proposé pour cela dans [20]. Nous avons choisi dans l'implémentation du prototype de permettre à `shim6` d'attribuer lui-même une adresse source, en lui donnant priorité sur le mécanisme RFC3484. Le principe est de vérifier s'il existe un contexte ayant B comme ULID destination, auquel cas on impose l'identifiant source de ce contexte comme adresse source du paquet (vu par l'application). Si un tel contexte ne peut être trouvé, alors on laisse travailler le mécanisme habituel, et un nouveau contexte est créé.

Cette manière de faire peut néanmoins engendrer une situation particulière : imaginons en effet que deux contextes existent, l'un pour le couple <A,B>, l'autre pour le couple <C,B>. Alors `shim6` choisit le premier qui se présente. Si par après le contexte identifié par <A,B> n'est plus utilisé pendant un certain temps (10 minutes dans notre implémentation), il est alors supprimé. Si après cela l'échange veut reprendre, *et laisse toujours l'adresse source indéterminée*, il se verra alors attribuer comme source l'adresse C.

La situation parfaite pour rompre une connexion TCP ? Non. Il est vrai qu'un échange TCP s'interrompt si l'une de ses adresses change. Mais puisque la constance des adresses est nécessaire pour lui, il est de son ressort de mémoriser l'adresse source qui lui a été attribuée lors du premier échange, pour ensuite *l'imposer*. On a donc la situation suivante : un premier segment TCP est envoyé avec <__,B>. Shim6 attribue l'adresse A, qui est mémorisée dans le socket TCP. Par la suite TCP envoie tous ses segments en imposant le couple <A,B>. De manière plus générale, ce comportement sera celui de n'importe quel socket en mode connecté. Il est par exemple possible de rendre 'connecté' un socket UDP, ce qui consiste justement à enregistrer dans le socket les adresses source et destination, ainsi bien sûr que d'autres informations (ports).

Reprenons le cas d'une absence d'échange pendant 10 minutes, avec disparition du contexte <A,B>. Cette fois l'échange ne sera plus attaché au contexte <C,B> puisque TCP recommence à envoyer des segments identifiés par <A,B>. La réaction de `shim6` est de recréer le contexte <A,B>, si bien que l'échange TCP peut continuer sans problème.

Dans ce qui a été dit ici, la solution que nous proposons n'a de nouveau que le principe d'imposition d'une adresse source par la couche `shim6`. La mémorisation de l'adresse source dans les sockets TCP est un élément déjà existant, dont nous nous contentons de tirer avantage dans la réalisation du

prototype.

L'avantage de l'approche proposée ici est de fournir une bonne indépendance entre les couches supérieures et `shim6` (rattachement automatique de plusieurs flux à un même contexte), et, par là même, de permettre d'éviter une occupation de mémoire par des contextes inutiles.

Dans [32], d'autres approches sont proposées (qui ne sont pas implémentées ici). Par exemple on se rend compte qu'il n'est probablement pas judicieux de déclencher un échange d'initialisation `shim6` pour une simple requête DNS. De manière plus générale, `shim6` sera surtout utile pour des connexions longues. On pourrait donc utiliser une heuristique qui serait par exemple d'activer `shim6` seulement après l'échange de 50 paquets. On peut aussi proposer une interface aux couches supérieures, leur permettant de décider si un flux précis doit bénéficier ou non du support de `shim6`. Aujourd'hui, il n'existe pas encore d'API `shim6`.

Bien sûr, un contexte `shim6` peut être aussi créé sur sollicitation par un correspondant. Puisque le contexte doit impérativement exister aux deux extrémités d'une communication, un hôte qui crée un contexte doit solliciter une création chez son correspondant. Le cas d'un serveur est particulier à ce niveau : un serveur n'a pas nécessairement d'intérêt à ce qu'une communication soit résistante aux pannes, par contre un client peut être intéressé par cette fonctionnalité. Ainsi, il serait intéressant de fournir une interface qui permette de désactiver 'partiellement' `shim6`, c'est-à-dire désactiver le déclenchement automatique (sur base d'une quelconque heuristique), mais accepter de créer un contexte sur sollicitation d'un client (intéressant par exemple pour un serveur FTP).

Nous proposons ceci comme élément possible d'une API `shim6`. Comme celle-ci n'a pas encore été définie à l'IETF, cet élément n'a pas été implémenté.

L'implémentation : Dans le noyau, il existe deux points d'entrée dans la couche `shim6`, pour les paquets sortants :

```
shim6_translate_route()
shim6_translate_output()
```

La première est moins complexe et sera décrite à la section 3.2.1. La seconde est appelée au moment où les paquets s'apprêtent à quitter la couche IP (voir figure 3.1). Tous les paquets sortants passent par cette fonction (`shim6_translate_output`), qui a le comportement suivant :

- Si `shim6` est désactivé au niveau du système, la fonction se termine immédiatement. La désactivation de `shim6` au niveau du système est implémentée par sysctl (dont le fonctionnement est décrit à la section 2.1.7). La variable contrôlée par sysctl est `sysctl_shim6_enabled`; elle est vérifiée par chacune des fonctions d'interface de `shim6` (fig. 3.1), qui sortent toutes immédiatement en cas de valeur nulle. La commande de (dés)activation est donc :
  `echo 1|0 > /proc/sys/net/ipv6/shim6/enabled`
  Cette désactivation a bien sûr pour conséquence de perdre toutes les connexions qui avaient subi un rehoming. Néanmoins, les contextes `shim6` déjà existants restent en mémoire (bien qu'ils ne servent plus) jusqu'à être libérés par le garbage collector, de telle manière que l'effet d'une désactivation peut être annulé simplement en réactivant `shim6`.
- Si le paquet sortant est aussi un paquet entrant (*forwarding*), alors la fonction se termine aussi directement : `shim6` n'est pas concerné par les paquets en transit. On voit qu'un paquet ne provient pas de l'hôte local en constatant que le champ `skb->input_dev`, qui décrit l'interface d'entrée du paquet, est différent de `NULL`.
- Si le paquet est un message de contrôle `shim6`, alors il ne nécessite aucun traitement supplémentaire, et la fonction se termine. Il en est de même pour les paquets ICMP(section 3.7).

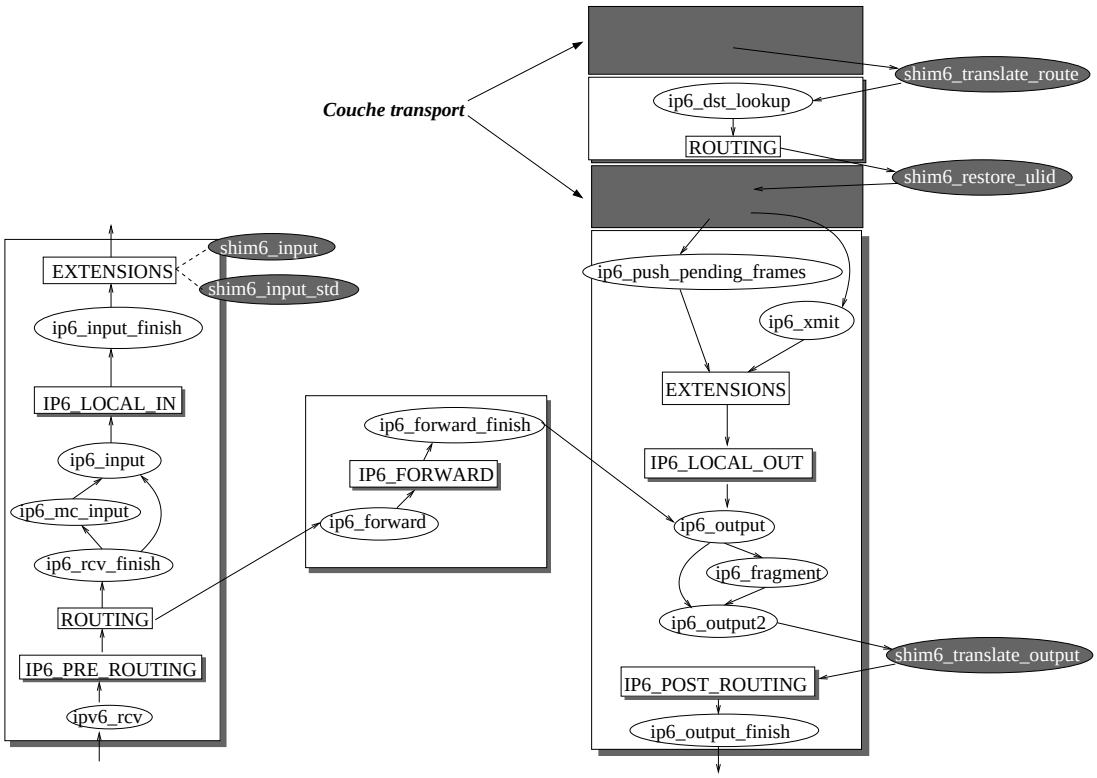


FIG. 3.1 – Insertion de `shim6` dans la couche IPv6

– Si aucune des conditions précédentes n'est vérifiée, alors il faut regarder si un contexte existe déjà pour les adresses du paquet, qui à cette étape sont encore des ULIDs. Si un tel contexte n'existe pas, alors il faudra le créer. C'est l'objet de la section suivante.

3.1.2 Création et initialisation d'un contexte

Un contexte est toujours créé par la fonction

```
struct shim6_ctx* shim6_lookup_and_create(struct sk_buff* skb)
```

Cette fonction va chercher les adresses du paquet `skb`, les comprend comme des ULIDs (`skb` doit être un paquet sortant) et essaye de trouver un contexte existant sur base de ceux-ci. Si aucun contexte n'est trouvé, la fonction en crée un nouveau. Cette opération doit se faire de manière atomique, sans quoi on risque de rencontrer la situation suivante :

– Un processus X envoie un paquet vers <ulid_a,ulid_b>. La table de contextes est consultée, aucun contexte n'est trouvé. On conclut donc qu'il faut en créer un nouveau.
– Un processus Y envoie un paquet vers le même couple d'adresses, en parallèle. La table de contextes est consultée, aucun contexte n'est trouvé. La même conclusion est faite, et un second contexte est créé pour la même paire d'ULIDs.

L'atomicité est assurée par un mutex, puisque l'on travaille dans un contexte utilisateur (variable `new_ctx_mutex` dans le code source).

La structure représentant un contexte `shim6` est représentée à la figure 3.2. Les divers éléments de ce contexte seront décrits au fur et à mesure de ce texte. Son contenu est visible à tout instant pendant l'exécution du noyau via le répertoire `/proc/net/shim6`. Un fichier est créé dans ce répertoire à

```
struct shim6_ctx {

  /*Collision resolution for the two hashtables*/
  struct list_head     collide_ulid;
  struct list_head     collide_ct;
  /*List of initializing contexts*/
  struct list_head     init_list;

  unsigned short       state;
  spinlock_t           lock;

  struct kref          kref;

  struct in6_addr      ulid_peer;
  struct in6_addr      ulid_local;

  __u32                fii; /*Forked Instance Identifier;
                              ignored for now*/

/*Locators info for the peer*/
  shim6_loc_p*         ls_peerp; /*Array of all locators*/
  int                  nb_peer_locs;
  unsigned int         gen_nb_peer; /*Locators list generation number*/
  shim6_loc_p*         lp_peer; /*Preferred loc, used as destination*/

/*Locators info for the host*/
  shim6_loc_l*         lp_local;

/*Context tags : 64 bits, but only 47 low order bits are used*/
  u64                  ct_peer;
  u64                  ct_local;

/*Initiator nonce*/
  u32                  init_nonce;

/*Timer information*/
  int                  lastuse; /*in jiffies, info for garbage coll.*/
  struct timer_list    retransmit_timer;
  int                  cur_timeout_val; /*for exp. backoff*/
  int                  nb_retries; /*maximum SHIM6_I1_RETRIES_MAX*/
  spinlock_t           stop_retr_lock; /* For protection of the
                                        * stop_retr_timer flag
                                        * (when doing the
                                        *  del_timer_sync)
                                        */
/*Flags.*/
  int                  stop_retr_timer:1,
                       translate:1, /* Translation activated*/
                       dead:1; /*If the context is about to
                                 be killed by the gc*/
  struct reap_ctx      reap;
/*procfs information*/
  char                 procfs_name[9];
};
```

FIG. 3.2 – Sructure d'un contexte `shim6`

chaque initialisation de contexte (et supprimé lorsque le contexte est détruit). Il fallait un nom unique pour chaque contexte, nous avons donc choisi par simplicité l'adresse mémoire du contexte en hexadécimal. Deux contextes ne peuvent en effet se trouver à la même adresse. Dans `__init_new_ctx()`, l'adresse du contexte est convertie en chaîne et stockée dans le champ `procfs_name`.

Insertion dans les tables de contextes : Il existe trois tables de contextes :

- `struct list_head ulid_hashtable[SHIM6_HASH_SIZE]` : c'est une table de hashage de taille 16 (`SHIM6_HASH_SIZE`). La clé est l'ULID *du correspondant* et le hash de celle-ci est calculé par la macro `hash_ulid(ulid_peer)`, qui fait appel à l'interface `jhash` (voir section 2.1.5).
  Intuitivement, il semble plus logique d'utiliser comme clé la paire d'identifiants, cependant le choix qui a été fait est d'accepter d'augmenter légèrement le taux de collisions (bien que très peu, puisque les adresses de pairs sont beaucoup plus variées que les adresses locales), au profit de l'attribution automatique d'adresse source par `shim6` : en cas d'adresse source indéterminée, `shim6_translate_route()` fait une recherche d'un contexte ayant comme *peer ULID* l'adresse destination du paquet à émettre. Cette recherche est rendue possible par le fait que la clé est l'adresse destination. Ensuite le premier contexte de la liste de collisions est choisi, et son adresse source est imposée. Ceci peut impliquer un changement d'*ULID* source en cours d'échange si le socket le laisse toujours indéterminé, mais ce n'est pas un problème puisque chaque fois qu'une paire d'adresses doit rester constante (TCP par ex.), le socket l'impose et le mécanisme d'attribution automatique ne sert pas, ou seulement la première fois.
  La liste de collisions de cette table correspond au champ `collide_ulid` de la structure `shim6_ctx` (fig. 3.2).
- `struct list_head ct_hashtable[SHIM6_HASH_SIZE]` : Le concept est complètement identique à la table précédente. La clé est le context tag, dont nous verrons le rôle plus loin.
- `struct list_head init_list` : C'est une simple liste, qui contient l'ensemble des contextes en cours d'initialisation. Ils sont retirés de la liste au moment de passer à l'état `established`. Cette liste sert à associer correctement à un contexte les paquets d'initialisation `shim6`, dont certains ne sont pas rattachables à un contexte sur base des ULIDs ou d'un context tag. Le choix d'une liste plutôt qu'une table de hashage provient du fait que l'initialisation ne dure qu'un court instant. De ce fait la liste sera toujours très petite (voire vide) ; une table de hashage ajouterait une complexité inutile sans gain d'efficacité.

Ces tables doivent être protégées contre les problèmes de concurrence : en effet, sans verrou, l'insertion d'un noeud dans une liste peut provoquer l'écrasement d'un autre qui serait inséré en même temps. Le verrou adéquat ici est un spinlock de type lecture/écriture. Les tables seront consultées pour chaque paquet entrant ou sortant, mais ne seront modifiées que lorsqu'un contexte est créé ou supprimé. Par ailleurs le même verrou peut servir pour les trois listes, car l'insertion d'un contexte se fait dans les trois listes en même temps. Quant à la lecture, elle ne réduit pas l'efficacité de par le fait que plusieurs lecteurs peuvent utiliser le verrou en même temps.

Stockage des localisateurs locaux : Dans le draft [32], la section 'modèle conceptuel d'un hôte' (section 6) propose de stocker dans le contexte `shim6` la liste des localisateurs locaux ainsi que la liste des localisateurs du pair.

C'est nécessaire pour les localisateurs du pair, par contre ceux de l'hôte local seront de cette manière recopiés beaucoup de fois, alors que, si l'on suppose que la plupart des connexions utilisent

les paramètres `shim6` par défaut (utilisation de tous les localisateurs disponibles), tous les contextes disposeront de la même information.

Il est donc plus efficace de ne rien garder des localisateurs locaux dans les contextes, mais de le faire de manière centralisée au niveau du système.

Les adresses unicast d'un équipement sont stockées dans une table de hashage au sein du module de gestion des adresses IPv6.

La table contient des structures `inet6_ifaddr` (étudiées à la section 2.4). Pour le support de `shim6`, des champs ont été ajoutés à cette structure.

L'implémentation prévoit donc que les localisateurs de l'équipement se trouvent sous la forme d'une `struct inet6_ifaddr` complétée avec les champs nécessaires à `shim6`. Par contre ceux du pair sont stockés sous la forme d'une structure `struct shim6_loc_p`, qui ne contient *que* des informations relatives à `shim6`. Afin de pouvoir faire abstraction de la structure `inet6_ifaddr` au sein de `shim6`, et de traiter de manière similaire les localisateurs du pair et ceux de l'équipement, on définit les deux structures suivantes, respectivement pour les localisateurs du pair (p) et ceux qui sont locaux (l) :

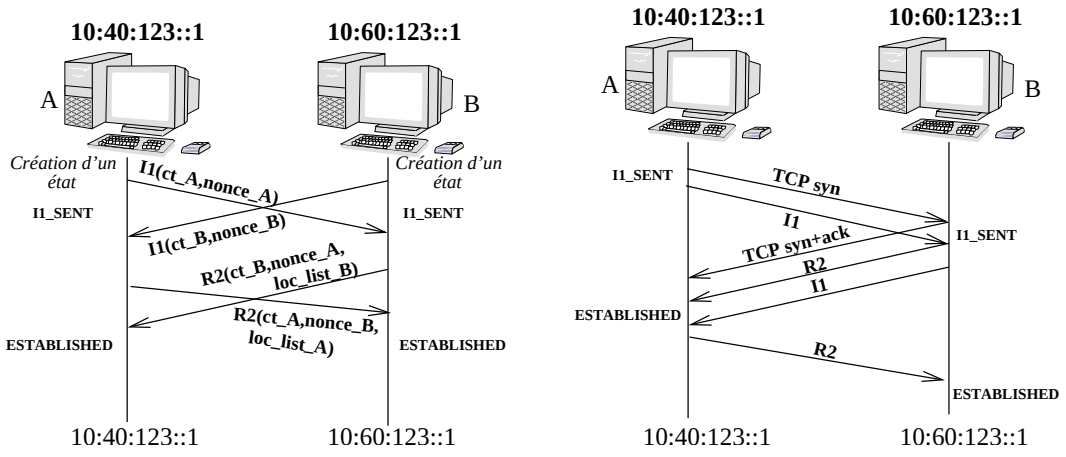
```
/*Locator structure : One structure per locator*/
struct shim6_loc_p {
  struct in6_addr   addr;
  __u8              valid_method; /*validation method*/
  __u8              valid_done:1,
                    probe_done:1;
};

/* shim6_loc_p and shim6_loc_l correspond to different structures,
 * but every field disponible in struct shim6_loc_p is also disponible
 * in shim6_loc_l : So it may be used as if it were the same thing. The only
 * thing we need to care about is to use shim6_loc_p when working with peer
 * locators, and shim6_loc_l when working with local locators.
 */
typedef struct shim6_loc_p shim6_loc_p;
typedef struct inet6_ifaddr shim6_loc_l;
```

Les champs de `shim6_loc_p` se retrouvent tels quels dans la structure `inet6_ifaddr` (mais pas au même endroit, raison pour laquelle on ne peut faire un casting).

Cette manière de faire permet de cacher aux fonctions de manipulations des localisateurs le choix d'implémentation qui a été fait, consistant en l'utilisation des adresses du module d'autoconfiguration d'adresses. Cela permettra aussi éventuellement dans le futur de facilement passer à un autre choix de design. Par exemple on pourrait, par des options fournies dans une API, permettre à certains contextes de gérer leur propre liste de localisateurs locaux.

Afin que l'implémentation de `shim6` puisse tenir compte des modifications dans les adresses disponibles localement, la fonction `ipv6_ifa_notify()` (définie dans *addrconf.c*) effectue un appel à `shim6_add_glob_locator()` ou `shim6_del_glob_locator()` selon qu'une adresse est ajoutée ou supprimée.

Puisque l'on n'est pas intéressé dans `shim6` par toutes les adresses IPv6 du système (l'adresse loopback ne peut pas être envoyée comme localisateur), une liste est maintenue par ces deux fonctions, `glob_locs_list`. Celle-ci doit aussi être protégée par un verrou de type lecture/écriture. Elle est modifiée sur notification du module d'autoconfiguration d'adresses, et est consultée lors de l'envoi de la liste des localisateurs au pair.



(a) cas général (b) Application à un échange client/serveur

FIG. 3.3 – Etablissement croisé

3.1.3 L'échange d'initialisation

Une fois le déclenchement de `shim6` décidé par un hôte, un échange d'initialisation prend place. Comme indiqué dans l'introduction de ce chapitre, le mode d'initialisation supporté dans le prototype est le mode simultané. L'échange de messages lié à ce mode d'initialisation se trouve à la figure 3.3.

Ce processus d'initialisation s'inscrit dans le comportement global d'un contexte `shim6`, illustré sous forme d'une machine à états à la figure 3.4.

On observe à la figure 3.3(b) qu'il n'y a pas réellement de croisement de messages. Il convient dès lors de définir plus clairement l'établissement croisé :

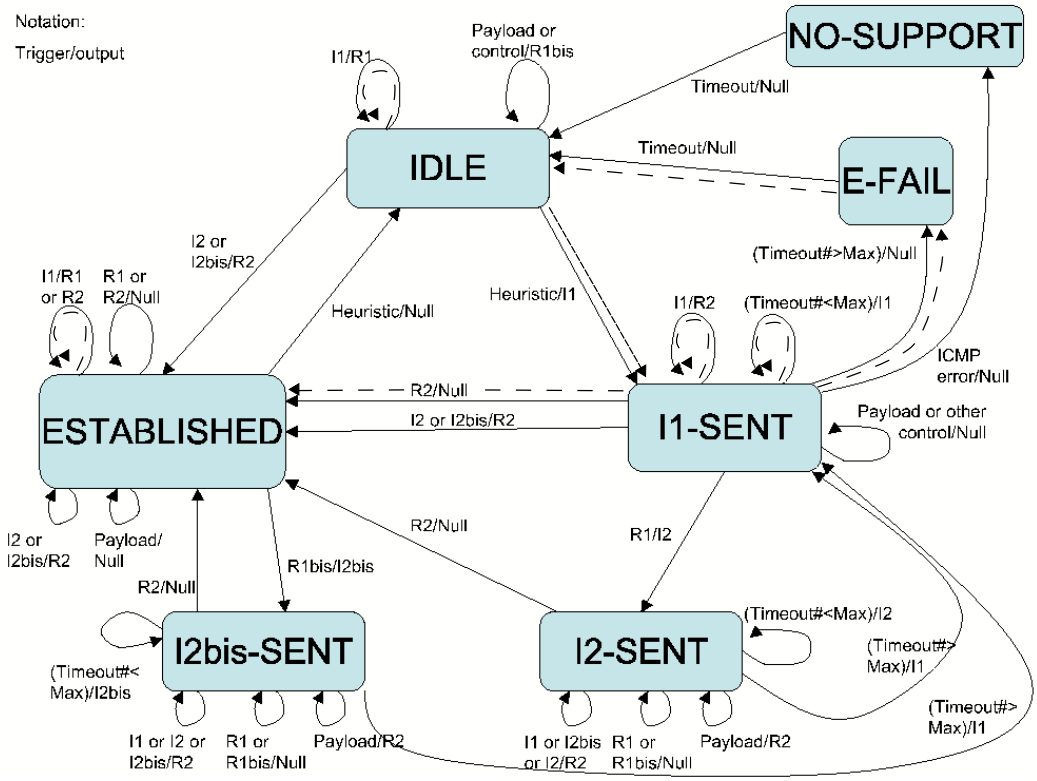
> Un échange d'initialisation `shim6` est dit *croisé* si chaque machine se trouve en l'état `I1_SENT` au moment de la réception d'un message I1.

Cette définition implique que tout établissement croisé se fait dans un contexte où *les deux correspondants* ont décidé d'établir une session `shim6` avec l'autre. Dans cette situation, aucune précaution particulière ne doit être prise, puisque les deux machines souhaitent se connecter. L'échange est donc simplifié et plus rapide : Dès réception du message I1 du correspondant, l'hôte apprend qu'il se trouve en établissement croisé et peut directement envoyer ses localisateurs (message R2).

A cause de la plus grande simplicité de cette méthode, et aussi sa plus grande rapidité, il est intéressant de favoriser cette situation. On peut le faire pour n'importe quel échange client/serveur en s'assurant que le message I1 parte *après* le paquet qui a déclenché son envoi (fig. 3.3(b)). Si ce n'était pas le cas, le receveur ne se trouverait pas en état `I1-SENT` au moment de sa réception.

Observons ce qui se produit dans le cas où A souhaite établir une communication SSH avec B (pour rappel, dans cette implémentation, l'établissement de contexte `shim6` est lancé dès l'envoi du premier segment TCP) :

– A envoie son premier segment TCP/SSH à B, ce qui déclenche la création d'un contexte et l'envoi d'un message `shim6` I1. Plus exactement, nous devons forcer I1 à partir après le segment TCP. Dans l'implémentation, il faut accepter de sortir de la couche `shim6` sans envoyer I1 pour que le segment TCP parte en premier. La solution à ce problème est d'utiliser le temporisateur de retransmission (`ctx->retransmit_timer`), qui ne sert pas à ce moment, pour programmer l'envoi le plus vite possible. En plaçant le champ `expires` du temporisateur à `jiffies` (qui signifie : expirer maintenant) ou une valeur inférieure, le noyau exécutera la



Représentation par machine à états du comportement d'un contexte Shim6, M. Bagnulo
disponible sur http://www.it.uc3m.es/marcelo/state_machine.pdf

Nous avons ajouté au dessin des flèches en traits discontinus, qui représentent les parties implémentées du protocole d'initialisation

FIG. 3.4 – Machine à états de `shim6`

fonction associée (`first_i1_transmit`) dès la prochaine interruption de temporisateur.
- B reçoit le segment SSH et y répond. La réponse déclenche la création d'un contexte et l'envoi d'un message I1 (encore une fois, après la réponse SSH).
- B reçoit I1, s'aperçoit qu'il avait lui-même déjà envoyé un paquet I1, et répond à A par R2. Le message I1 avait servi à vérifier le support de `shim6` chez le corresondant, ainsi qu'à prendre connaissance du mode d'établissement (croisé ou non). Dans le message R2, B envoie les localisateurs que A pourra utiliser pour le joindre. Il envoie aussi son context tag, qui permettra d'identifier le contexte après une opération de rehoming (la communication après rehoming est étudiée dans la section 3.3).
- A reçoit I1, se comporte exactement comme B et envoie un message R2.
- A reçoit R2, enregistre les localisateurs contenus dans celui-ci, et passe en état `established`.
- B reçoit R2, copie les localisateurs et passe en état `established`.

La figure 3.3(b) indique que ce mode d'échange implique un envoi simultané des messages I1 et R2 par B. En pratique, ce n'est pas possible, mais la conséquence est que l'on ne peut prévoir lequel sortira en premier. Ce n'est pas un problème : la seule différence est que si R2 arrive avant I1, alors A recevra I1 en ayant déjà l'état `established`. Or la machine à états (figure 3.4) spécifie que là aussi il faut répondre par un message R2 (L'autre situation spécifiée dans le schéma, où l'hôte répond par un message R1, est un cas particulier qui se produit si le message I1 n'est pas cohérent avec l'état déjà existant, ce cas n'est pas supporté).

Rôle du nonce de l'initiateur : Un nonce est utilisé par A pour retrouver le contexte qu'il avait créé au moment de l'envoi de I1. Ce nonce est répété par le correspondant dans le message R2, ce qui permet à A de retrouver le contexte correct. C'est dans ce cas qu'est utilisée la liste `init_list`. Au moment de la réception d'un message R2, cette liste est parcourue à la recherche d'un contexte correspondant au nonce joint au message R2.

Le nonce de l'initiateur doit donc être stocké dans le contexte. Dans la structure de la figure 3.2, c'est le champ `ctx->init_nonce`. C'est le cas aussi des contexts tags annoncés dans les messages R2 : `ctx->ct_local` et `ctx->ct_peer`.

L'emploi d'un nonce permet aussi d'obtenir une certaine protection contre des envois de paquets ayant pour but d'interférer avec l'établissement ([32], section 17). Par exemple, si A n'utilisait pas de nonce, n'importe qui (sans nécessairement être sur le chemin) pourrait lui envoyer un message R2 spoofé.

Le context tag : On avait vu au chapitre 1 que l'emploi d'un context tag pour retrouver un contexte `shim6` après rehoming était l'une des solutions envisagées. Aujourd'hui c'est cette solution qui a été retenue à l'IETF, ainsi qu'il est décrit dans le draft [32].

C'est un nombre de 47 bits, stocké par commodité sur 64 bits dans l'hôte. Pour chaque nouveau contexte créé, une valeur de context tag est choisie en prenant la valeur courante de la variable globale `cur_local_ct`. Puisqu'il s'agit d'un identifiant unique (au niveau de l'hôte), la valeur est lue, puis la variable incrémentée. La lecture et l'incrémentation doivent se faire de manière atomique, se que l'on voudrait réaliser avec l'interface `atomic.h` (section 2.1.2). Malheureusement cette interface ne travaille qu'avec des nombres de 32 bits, et nous utilisons donc un spinlock (`ct_lock`).

Lorsque `cur_local_ct` arrive à 2^{47}, elle est ramenée à 0. Pour cette raison, il faut toujours vérifier qu'aucun état ne possède déjà le même identifiant de contexte. On peut le faire par simple consultation de la table de hashage `ct_hashtable` sur base de ce nouvel identifiant. Tant qu'un identifiant libre n'a pas été trouvé, la variable `cur_local_ct` est incrémentée.

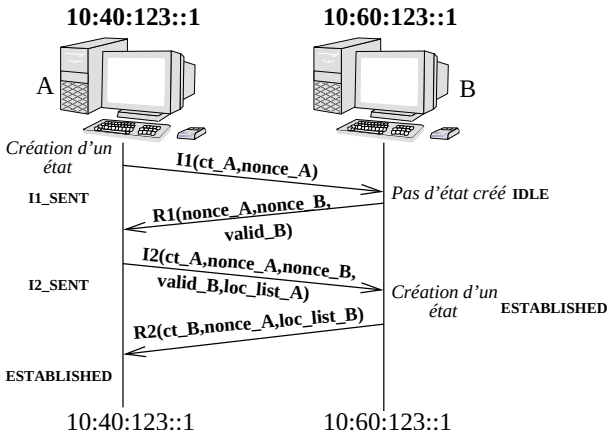


FIG. 3.5 – Mode d'échange avec nonces

Aspects de sécurité de l'échange : Une attaque possible sur l'échange tel qu'il est décrit est d'inonder un serveur de paquets munis d'adresses sources différentes. Dans l'exemple ci-dessus, si les paquets d'attaques sont des TCP SYN, le serveur répondra par TCP SYN+ACK, et créera un contexte pour chaque paquet reçu. Voilà donc une autre raison qui justifierait la création d'une option de désactivation partielle de `shim6`, telle que décrite à la section 3.1.1.

Dans ce cas l'établissement simultané n'est plus possible, et il devient nécessaire de faire appel à une méthode qui protège le serveur des attaques DoS. Cette solution correspond au second mode d'initialisation (non implémenté actuellement), qui met en oeuvre deux messages supplémentaires, R1 et I2 (fig. 3.5) : Sur réception d'un message I1, le serveur répond par un message R1, et ne crée pas d'état. Au contraire, il fournit un nonce au demandeur d'établissement, que celui-ci devra répéter dans un message I2. Dans ce cas les localisateurs et le context tag peuvent déjà être joints au message I2, ce qui permet de terminer l'échange avec la réponse R2 du serveur, qui cette fois crée un état.

Le nonce du répondeur (B) est différent de celui de l'initiateur. C'est un nombre incrémenté à chaque réception de message I1.

Par ailleurs B utilise un secret S pour calculer un hash (one-way function) de celui-ci, du nonce et des informations contenues dans I1. Le nombre secret permet d'assurer qu'un utilisateur malveillant ne puisse générer le hash. Ce hash est envoyé dans R1, et simplement recopié par A dans I2.

B sait maintenant qu'il peut créer un contexte, car il a pu recalculer le hash sur base du nonce, des informations contenues dans I1 (qui doivent donc êtres recopiées dans I2), et de son secret S qui reste constant. Ceci prouve que A a bien reçu R1.

Une description complète de l'établissement de session `shim6` se trouve dans la section 7 de [32].

3.1.4 Les localisateurs

Dans les messages I2 et R2, les hôtes s'échangent leurs localisateurs (loc_list_A et loc_list_B dans la figure 3.3). Comme on l'a vu à la section 1.3.3, cet échange doit être sécurisé afin d'éviter qu'un attaquant ne puisse injecter l'une de ses adresses dans les messages I2 ou R2. Pour cette raison, ces messages contiennent également toutes les informations de vérification pour chacun des localisateurs (qui peuvent être du type HBA ou CGA).

[32] indique que la vérification d'authenticité HBA ou CGA est obligatoire. Néanmoins nous ne l'avons pas implémenté dans notre prototype, parce que l'implémentation de cette fonction relève de la sécurité et non du bon fonctionnement du protocole. Le programmeur intéressé par l'ajout de

ces fonctionnalités de sécurité pourra utilement s'inspirer de l'implémentation d'HBA réalisée par F. Dupont pour OpenSSL (*http ://ops.ietf.org/multi6/francis-hba.tar.gz*).

Notion de groupe de localisateurs : Le protocole `shim6` prévoit que les localisateurs soient communiqués par groupes non modifiables. Cela signifie qu'un groupe de localisateurs est émis initialement dans un message I2 ou R2. Ensuite si un hôte souhaite annoncer à son correspondant qu'il a un nouveau localisateur, ou que l'un de ceux qu'il a envoyé n'est plus valide, alors il devra envoyer un nouveau groupe complet de localisateurs.

A chaque groupe est associé un numéro qui l'identifie, appelé *nombre de génération* de la liste de localisateurs. Ce numéro est joint à tout message contenant une liste de localisateurs, et conservé dans le champ `ctx->gen_nb_peer`. Le nombre de génération de l'hôte est une variable globale `glob_gen_nb`, à cause du fait que les adresses locales sont gérées de manière commune pour tous les contextes. Cette variable est incrémentée à chaque notification du système d'auto-configuration d'adresses (`shim6_add_glob_locator` ou `shim_del_glob_locator`).

De par le fait que le nombre de génération de liste identifie de manière unique une liste, sa modification ou sa consultation doit se faire à l'intérieur de la zone critique de lecture/écriture dans la liste de localisateurs locaux (protégée par `glob_locs_lock`).

3.2 La communication

Durant le processus d'initialisation, la liste des localisateurs de chaque hôte a été communiquée, ainsi qu'un context tag.

Tant qu'il n'y a pas de panne, la communication continue comme si la couche `shim6` n'existait pas : il faut cependant informer le contexte nouvellement créé du passage des paquets, pour les besoins du protocole REAP (chapitre 4). Chaque paquet entrant ou sortant est donc rattaché à son contexte sur base de ses ULIDs local et distant, ainsi que, optionnellement, sur base d'un FII.

Dans notre situation, où le FII n'est pas utilisé, et tant que les localisateurs sont identiques aux ULIDs, tous les paquets (entrants et sortants) sont rattachés à un contexte sur base des identifiants (toujours présents dans les champs source et destination d'un paquet IP, puisque les localisateurs *sont* à ce moment les identifiants). Le travail de `shim6` une fois un paquet attaché à un contexte consiste seulement en une tâche de comptabilité de paquets, gérée par le protocole REAP, décrit au chapitre suivant.

Du point de vue de l'implémentation, il faut décider à quel endroit intervient `shim6` dans la pile IPv6. Les points d'insertion de `shim6` sont indiqués à la figure 3.1.

3.2.1 Traitement des paquets sortants

Pour les paquets sortants, le routage est effectué lorsque l'on se trouve encore dans la couche transport. Or ceci doit absolument se faire en utilisant les localisateurs. Par contre lorsque la couche transport reprend le contrôle pour terminer ses tâches, elle doit retrouver ses ULIDs. La solution à ce problème est de pratiquer deux passages dans `shim6`. Lors du premier passage, par la fonction `shim6_translate_route`, `shim6` tente de trouver un contexte correspondant aux ULIDs. Si l'adresse source est indéterminée, il recherche une correspondance seulement pour l'adresse destination, et en cas de succès attribue lui-même l'adresse source. Si aucun contexte n'est trouvé, la fonction `shim6_translate_route` se termine. Il faut noter que l'on ne crée jamais de contexte à ce stade : ceci est laissé à la fonction `shim6_translate_output`.

L'appel à `shim6_translate_route` est effectué en couche transport, juste avant l'appel à `ip6_dst_lookup`. Il pourrait dès lors sembler plus adéquat de placer l'appel à `shim6` à l'intérieur de `ip6_dst_lookup`. Mais ceci serait incorrect vis-à-vis de l'option de source routing : lorsque cette option est utilisée, la couche transport remplace l'adresse destination par le premier saut, et la restaure après le routage. Ce comportement est en fait similaire à ce que nous faisons pous `shim6`. Mais à cause de cela, nous devons faire la traduction `shim6` avant la traduction source routing. Illustrons ceci sur un exemple :

- Un paquet possède les ULIDs source et destination <A,B>, et une option source routing munie d'un premier saut <X> ;
- `shim6` traduit les ULIDs en localisateurs, suite à quoi le paquet possède les adresses <C,D> ;
- Le traitement source routing garde D dans un tampon, et remplace D par X. Le paquet a maintenant les adresses <C,X> ;
- La fonction `ip6_dst_lookup` est appelée, elle effectue le routage pour le couple <C,X>, ce qui est correct : Les deux sont des localisateurs, et X est le prochain saut ;
- L'adresse D est restaurée, le paquet possède les adresses <C,D> ;
- `shim6` restore les ULIDs, dans `shim6_restore_ulid`. Le paquet revient alors aux ULIDs <A,B>, et le traitement de couche transport peut continuer.

Cette manipulation suppose qu'un tampon supplémentaire soit utilisé pour stocker les ULIDs pendant le routage, exactement comme le fait l'option de source routing.

Ensuite, pour l'appel final à `shim6`, nous attendons le dernier moment avant de quitter la couche IPv6 : en particulier [32] spécifie que le traitement de la fragmentation doit se faire sur base des ULIDs (rappelons que la fragmentation dans IPv6 est une tâche réservée aux extrémités d'une communication). C'est pourquoi la fonction `shim6_translate_output` prend place après l'appel à `ip6_fragment`.

Par contre nous avons choisi de placer la traduction avant le dernier passage dans netfilter, de telle manière que le filtrage se fasse sur base des localisateurs.

3.2.2 Traitement des paquets entrants

Il est beaucoup plus simple : les paquets entrants passent soient dans la fonction `shim6_input`, soit dans `shim6_input_std` (fig. 3.1, selon qu'il sont ou non munis d'une extension d'en-tête `shim6`.

Les paquets ne possédant pas d'extension `shim6` sont traités par `shim6_input_std`. Le fait qu'un paquet n'ait pas l'extension `shim6` signifie qu'il n'existe pas de contexte pour ce paquet, ou bien qu'il existe un contexte mais qu'actuellement les ULIDs sont identiques au localisateurs. Dans le premier cas rien n'est effectué. Par contre si un contexte existe, il faut lui notifier l'arrivée d'un paquet lui appartenant pour les besoins du protocole REAP (chapitre 4). On met également à jour la variable `ctx->lastuse` (fig. 3.2) avec la valeur courante de `jiffies`. Ce champ est utilisé par le garbage collector pour détecter qu'un contexte n'a pas été utilisé pendant 10 minutes. Le mécanisme du garbage collector sera décrit à la section 3.5.

Dans le cas où l'extension d'en-tête `shim6` est présente, cela signifie que les localisateurs sont différents des ULIDs, et qu'il y a donc eu rehoming. Ce cas est traité à la section suivante.

Pour ce qui est du placement des appels à `shim6` pour les paquets entrants, cela se fait dans `ip6_input_finish`. En cet endroit, seuls les paquets à destination de l'hôte arrivent. Plus précisément, l'appel à `shim6` est placé entre le traitement hop by hop et celui des autres extensions d'en-tête, ce qui est le comportement prévu par [32].

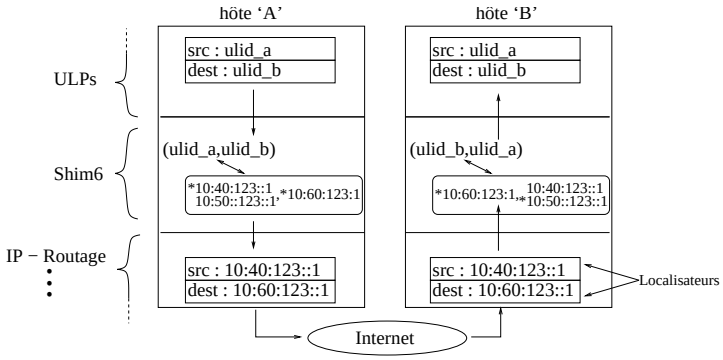


FIG. 3.6 – Modification des paquets dans la couche Shim (communication en cours)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   | hdr ext lgth  |P|                                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                                  |
|                       Receiver Context Tag                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FIG. 3.7 – Extension d'en-tête `shim6` utilisée après rehoming

3.3 Communication après rehoming

Grâce au protocole REAP (chapitre 4), les pannes sont détectées et une nouvelle paire de localisateurs est attribuée. Les localisateurs courants sont stockés dans les champs `ctx->lp_local` et `ctx->lp_peer`. Après rehoming, au moins l'un de ces champs devient différent des ULIDs (`ctx->ulid_local` et `ctx->ulid_peer`), et le drapeau `ctx->translate` est activé.

La couche `shim6` devra désormais effectuer une traduction des ULIDs en localisateurs et inversément. Ce processus est illustré à la figure 1.9, que nous reprenons ici (figure 3.6) par commodité. Le principe de traduction des adresses par la couche `shim6` a déjà été décrit à la section 1.3.2. Nous y disions que les ULIDs servaient de clés pour retrouver le contexte `shim6` et réaliser la traduction d'adresses. Nous y avons vu aussi qu'il fallait trouver un autre moyen d'associer le bon contexte aux paquets *entrants*. Ceci est dû au fait que les ULIDs ne sont pas transportés dans ces paquets, et que les localisateurs n'identifient pas un contexte de manière unique. Au moment de la rédaction du premier chapitre, la résolution de ce problème était en cours de discussion à l'IETF, et nous avons choisi de le laisser tel quel, puisque cela constitue un exposé intéressant des alternatives proposées au cours du temps.

Au moment de la rédaction de ce chapitre, un nouveau draft est apparu, [32], qui résout le problème de l'association à un contexte pour les paquets entrants par l'insertion d'un context tag dans une extension d'en-tête IPv6. Ce context tag, d'une longueur de 47 bits, identifie sans ambiguïté un contexte, et est annoncé au pair lors de l'échange d'initialisation (figure 3.3, `ct_A` et `ct_B`). Par exemple, A annonce à B son context tag, `ct_A`, qu'il a choisi unique. Ensuite tous les paquets envoyés par B et nécessitant une traduction sont munis d'une extension d'en-tête qui contient `ct_A`. Cette extension d'en-tête IPv6 est présentée à la figure 3.7 (Une introduction au principe des exten-

sions d'en-tête IPv6 peut être trouvée dans [3]). Les champs ont la signification suivante :

- `Next header` : code de l'en-tête suivante, par exemple 6 s'il s'agit de TCP. Notons que l'en-tête précédente (donc l'en-tête IPv6 ou l'extension d'en-tête placée avant celle-ci) doit comprendre dans son champ Next Header le code de `shim6`. Ce code n'a pas encore été attribué officiellement par IANA. Afin de pouvoir mener à bonne fin le prototype, nous avons choisi un code non encore utilisé (à notre connaissance) : 61. Bien sûr, cette valeur devrait être mise à jour dès que le code officiel sera attribué par IANA.
- `Hdr ext length` : ce champ indique la longueur de l'en-tête en unités de 8 octets, sans compter les 8 premiers octets. Donc pour l'en-tête de payload décrite ici, ce champ vaut zéro.
- `P` : un bit pour permettre à `shim6` de distinguer les paquets de payload (1), c'est-à-dire qui contiennent des données à transmettre aux couches supérieures. Si ce bit vaut 0, alors le paquet est un paquet de contrôle (par exemple les messages d'initialisation). Ici ce bit est à 1.
- `Context tag` : sur 47 bits, c'est l'identifiant qui permet au receveur de retrouver le contexte auquel appartient le paquet.

Si un paquet entrant possède une extension d'en-tête `shim6`, le context tag en est extrait. Le contexte associé est retrouvé, ce qui permet de restaurer les ULIDs. De plus, l'extension d'en-tête est supprimée pour permettre un traitement transparent dans les couches supérieures.

Dans l'implémentation, l'effet de 'suppression' est obtenu simplement par décalage du pointeur `skb->h.raw` (fig. 2.2), de telle manière qu'il pointe vers l'en-tête suivante. Le paquet traduit et libéré de son en-tête `shim6` peut alors continuer son chemin vers les couches supérieures, qui verront dès lors les ULIDs.

3.3.1 Problème du checksum TCP

Dans IPv4, l'en-tête des paquets contenait un checksum, modifié à chaque routeur (à cause de la décrémentation du champ TTL). L'un des buts d'IPv6 étant d'augmenter la rapidité du forwarding des paquets, beaucoup d'options d'en-tête ont été soit supprimées, soit remplacées par des extensions.

Dans le cas du checksum il a été décidé que ce rôle relèverait désormais uniquement de la responsabilité de la couche transport. C'est pourquoi dans IPv6, l'en-tête ne contient plus de champ checksum. Celui-ci devient par contre obligatoire pour tous les protocoles de couche transport, et doit englober les adresses IP source et destination [3]. La conséquence au niveau du routage est qu'un routeur ne pourra plus voir si une adresse a subi une modification. Dans un tel cas, le paquet sera routé vers une mauvaise destination qui, elle, vérifiera le checksum au niveau transport et jettera le paquet à ce moment.

Ces checksums, pour UDP et TCP, sont calculés sur base d'une pseudo-en-tête contenant les adresses IP (voir figure 3.8).

Ceci pose donc question au niveau de `shim6`, à cause de la modification d'adresses. [32] définit l'ULID comme l'adresse servant au calcul de checksum. Shim6 ne le modifie donc pas lorsqu'il remplace les adresses de l'en-tête IPv6.

Il faut noter cependant que pour cette raison, seules les extrémités de la connexion peuvent calculer/modifier le checksum. Ainsi, si l'on souhaite qu'un équipement intermédiaire du réseau soit capable de le manipuler, il faudrait qu'il maintienne un état associé à chaque connexion `shim6`, afin de garder une trace des ULIDs. Il semble que les NATs soient les seuls dispositifs affectés par cela.

Ils devraient comme en IPv4 remplacer le localisateur du paquet par un autre, mais ne pourraient plus modifier le checksum. Par ailleurs ils devraient traduire également les localisateurs contenus dans les échange d'initialisation `shim6`.

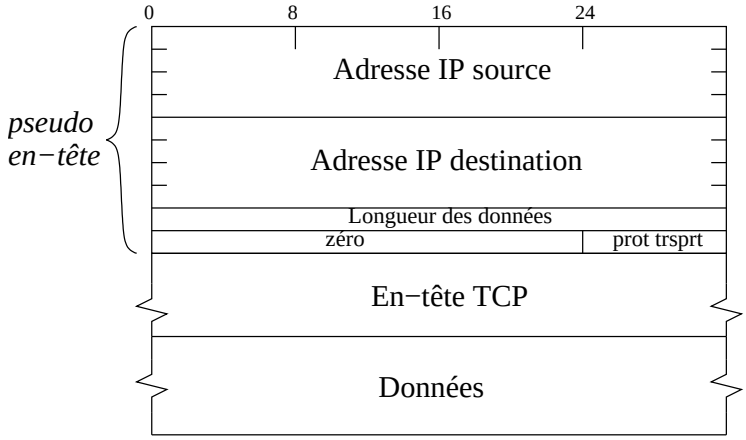


FIG. 3.8 – Données utilisées pour le calcul du checksum TCP

L'emploi du système NAT ne respecte pas le *end-to-end principle* d'Internet, qui consiste à refuser autant que possible le maintien d'états (coûteux en traitement et mémoire) dans les équipements de l'Internet [14, 15].

Il y a pourtant des discussions sur de la réécriture d'information `shim6` dans le réseau, mais ce n'est pas stable aujourd'hui.

3.4 Mise à jour des listes de localisateurs

Il est possible que des localisateurs apparaissent ou disparaissent après qu'un contexte `shim6` ait été établi. Le protocole défini dans [32] prévoit pour cela un message de mise à jour de la liste de localisateurs. Ce message peut soit contenir une nouvelle liste, soit de nouvelles préférences (pour le tri des localisateurs au moment de l'envoi des sondes), soit les deux. Dans tous les cas, l'option de signature HBA/CGA doit être jointe au message.

Pour permettre de modifier les préférences sans renvoyer la liste de localisateurs, le nombre de génération de la liste est utilisé (section 3.1.4). Ainsi, si l'on veut seulement modifier les préférences des localisateurs, l'envoi du numéro de liste permet d'assurer que l'on se réfère à la même liste. Si le receveur détecte une différence dans le numéro de liste, il répond par un message ICMP d'erreur.

La fonction de mise à jour des localisateurs n'est pas implémentée dans le prototype. Cette fonction peut être laissée comme extension car normalement la liste de localisateurs ne change pas souvent. Les conséquences de ne pas implémenter cette fonction sont :

- De ne pas pouvoir utiliser les nouveaux localisateurs rendus disponibles après établissement de la session `shim6` ;
- D'envoyer des sondes vers des localisateurs périmés, en cas d'exploration. Cela ne fait que ralentir le processus de récupération en cas de panne.

3.5 Suppression des anciens contextes

Lorsqu'il n'y a plus d'échange entre deux hôtes liés par une session `shim6`, le contexte peut être éliminé de la mémoire. Cela se fait après 10 minutes d'inactivité *des couches supérieures* (les messages de contrôle `shim6` ne sont pas considérés comme une activité).

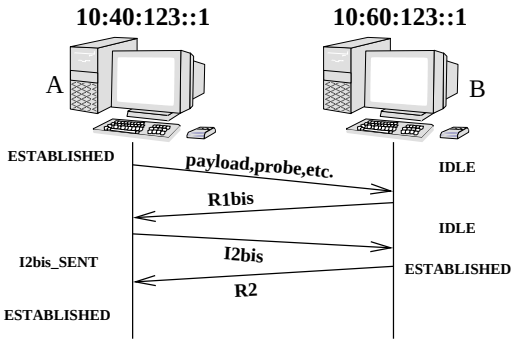


FIG. 3.9 – Rétablissement d'un contexte

Lorsqu'un contexte est éliminé, il l'est directement, sans avertir le correspondant. Ce fait peut mener à la situation où l'un des hôtes a éliminé un contexte mais pas l'autre. Il peut donc arriver qu'un hôte reçoive un paquet muni d'un context tag qu'il est incapable d'associer à un contexte existant parce qu'il a déjà été supprimé. Il devra en informer son correspondant pour démarrer un rétablissement de contexte. Ce processus est illustré à la figure 3.9 (inspirée de la figure 27 de [32]). Le comportement exact est défini par la machine à états de la figure 3.4.

Sur la figure 3.9, B reçoit un paquet `shim6` qu'il ne parvient pas à associer à un contexte. Rappelons qu'un contexte inexistant est toujours considéré comme étant à l'état `idle`. Comme ce paquet peut être une tentative d'attaque par inondation de paquets (flooding, section 3.1.3), B ne peut pas recréer l'état directement, mais plutôt envoyer un paquet R1bis, qui comprend un nonce et un hash comme le paquet R1 d'initialisation. Lorsque A reçoit le paquet R1bis, il comprend que B a éliminé son contexte et lui répond par un message I2bis, contenant les informations nécessaires à la recréation du contexte. B peut alors reconstruire son contexte et passer directement à l'état `established`. B devra aussi choisir un nouveau *context tag* pour son contexte, qu'il annoncera à A dans un message R2.

Les deux hôtes sont de nouveau en état `established` et peuvent continuer l'échange.

La récupération de contexte n'est pas supportée actuellement dans notre implémentation : lorsqu'un hôte ne parvient pas à associer un paquet entrant à un contexte, il le jette tout simplement. Dans le cadre d'expérimentations, cela ne constitue pas une limitation, puisque l'on peut toujours maintenir un contexte en vie en s'assurant que des paquets y passent régulièrement.

Dans un usage réel, c'est une fonction utile, mais pas suffisante pour garantir une bonne robustesse de `shim6`. Il faudra également, et surtout, utiliser d'autres critères que l'arrêt d'activité pour l'élimination d'un contexte. Par exemple, l'existence d'un socket TCP ayant une connexion établie pour une paire donnée d'ULIDs devrait être connue de `shim6` pour qu'il ne supprime pas son contexte tant que la liaison TCP existe. Cela non plus n'a pas été réalisé, de nouveau parce qu'on peut éviter les problèmes (en laboratoire) en maintenant le trafic entre les hôtes aussi longtemps que nécessaire.

3.6 Le garbage collector de shim6

Il a été décidé de supprimer un contexte lorsqu'il ne voit plus passer de paquets pendant au moins 10 minutes. Dans l'implémentation, on pourrait ajouter un temporisateur à chaque contexte, qui serait réinitialisé à chaque passage de paquet. En réalité, il est suffisant d'utiliser un seul temporisateur pour tout le système, qui regarde l'âge de tous les contextes et supprime ceux qui sont âgés de plus de 10 minutes. L'âge d'un contexte se calcule à partir de la valeur actuelle de `jiffies`, et le champ

`ctx->lastuse`. Ce champ est mis à jour chaque fois qu'un paquet *de données* traverse la couche `shim6`, de manière à refléter son usage par les couches supérieures. Il est à noter que ce mécanisme implique que la durée de vie d'un contexte `shim6` inactif peut varier entre 10 et 20 minutes (car le garbage collector s'exécute toutes les 10 minutes).

Pour itérer sur tous les contextes, il suffit de parcourir la table `ulid_hashtable` (`ct_table` aurait bien sûr convenu aussi), par index, et pour chaque index itérer sur la liste de collisions. Pour chaque contexte ainsi trouvé, l'âge est calculé et le contexte est supprimé si nécessaire.

La suppression de contexte est une opération délicate, à cause du fait qu'il peut avoir des temporisateurs en attente, ou même être utilisé à ce moment par une fonction (cela peut se produire si les couches supérieures décident justement de se réactiver au moment où le contexte allait être supprimé).

3.6.1 Arrêt des temporisateurs

La fonction `del_timer` permet de désactiver un temporisateur. Mais s'il vient de se déclencher, `del_timer` ne fait rien, et la routine de traitement est en cours d'exécution. Il existe aussi `del_timer_sync`, qui se synchronise avec la routine de traitement, de telle manière que le contrôle n'est rendu que lorsque la routine de traitement est terminée. Mais les spécifications de cette fonction (*kernel/timer.c*) précisent que le temporisateur ne peut être relancé à la fin de sa routine de traitement. Malheureusement, c'est exactement ce qui se passe pour le temporisateur de retransmission des messages d'initialisation I1 : sur expiration, un message I1 est renvoyé, puis le délai d'expiration est augmenté en respectant un backoff exponentiel, et le temporisateur est réarmé.

[2] suggère, dans cette situation, de faire appel à un drapeau "stop timer", qui serait consulté par la routine d'interruption pour savoir si elle peut réarmer le temporisateur. Nous plaçons donc ce drapeau dans le contexte : `ctx->stop_retr_timer`.

Mais le problème n'est pas encore résolu, car des conditions d'exécution parallèle peuvent, même dans ce cas, mener la routine d'interruption à travailler sur un contexte supprimé. La situation suivante peut en effet se produire :

```
if (!ctx->stop_retr_timer)
                                        ctx->stop_retr_timer=1;
                                        del_timer_sync(&ctx->retransmit_timer);
add_timer(&ctx->retransmit_timer);
```

La colonne de gauche est la portion de code de la routine de traitement du temporisateur. La colonne de droite est une portion de code du garbage collector. Dans le cas d'exécution parallèle présenté, on finit par armer le temporisateur malgré le fait que la fonction `del_timer_sync` est en cours d'exécution. Nous ajouterons donc un verrou pour protéger cette zone de code :

```
                                        ctx->stop_retr_timer=1;
                                        spin_lock(&ctx->stop_retr_lock);
                                        del_timer_sync(&ctx->retransmit_timer);
spin_lock(&ctx->stop_retr_lock);        /*DEADLOCK*/
if (!ctx->stop_retr_timer)
  add_timer(&ctx->retransmit_timer);
spin_unlock(&ctx->stop_retr_lock);
```

Cette fois, un problème de deadlock apparaît :

- La routine de traitement du temporisateur commence à s'exécuter.
- Le garbage collector obtient le verrou, et `del_timer_sync` se bloque en attendant que la routine de traitement se termine.
- Cette routine ne se terminera jamais, car elle attend que le verrou se libère.

La solution finale est la suivante, où nous utilisons un verrou non bloquant dans la routine du temporisateur :

```
lock_hold=
  spin_trylock(&ctx->stop_retr_lock);
if (!ctx->stop_retr_timer)
  add_timer(&ctx->retransmit_timer);
if (lock_hold)
  spin_unlock(ctx->stop_retr_lock);
```

```
ctx->stop_retr_timer=1;
spin_lock(&ctx->stop_retr_lock);
del_timer_sync(
  &ctx->retransmit_timer);
spin_unlock(&ctx->stop_retr_lock);
```

Ici le deadlock ne peut plus se produire car la routine ne bloque pas sur le spinlock. Un verrou non bloquant est suffisant, en effet :

- Si la routine parvient à acquérir le verrou, alors le garbage collector devra attendre que celui-ci soit libéré pour exécuter `del_timer_sync`. Entretemps, le temporisateur aura été réarmé complètement, si bien que `del_timer_sync` n'aura plus qu'à le désactiver(On évite la situation intermédiaire ambiguë, où le temporisateur est en cours de réarmement) ;
- Si la routine ne parvient pas à acquérir le verrou, alors il a été acquis par le garbage collector, mais étant donné l'ordre des instructions, cela signifie aussi que `ctx->stop_retr_timer` vaut 1. Le temporisateur ne sera pas réarmé dans ce cas.

En résumé, ce mécanisme final permet d'assurer que le temporisateur ne soit jamais réarmé après un appel à `del_timer_sync`. Cette fonction effectuera dès lors correctement son travail, qui est de garantir qu'aucun code de la routine de temporisateur ne s'exécute ni ne s'exécutera lorsque le contrôle est rendu à l'appelant de la fonction.

3.6.2 Manipulation des compteurs de références

L'interface *kref* est utilisée pour la gestion des compteurs de références (champ `ctx->kref` dans la figure 3.2).

Le compteur `kref` est initialisé dans `__init_new_ctx()` :

```
kref_init(&ctx->kref);
```

A ce moment le compteur de références est initialisé à 1. Ensuite il faut l'incrémenter chaque fois qu'une nouvelle référence vers le contexte est obtenue, c'est-à-dire dans `shim6_lookup_ulid()`, `shim6_lookup_ct()` et lorsque le pointeur est extrait de la liste `init_list`.

A la section 2.1.3, nous insistions sur la nécessité de protéger l'incrémentation du compteur de références par un verrou. C'est bien le cas ici, nous travaillons sous la protection du `hash_lock`, puisqu'il y a consultation ou modification (pour le garbage collector) de la liste d'initialisation ou des tables de hashage.

Chaque fois qu'une fonction cesse d'utiliser le pointeur, elle appelle la fonction `kref_put`, avec comme conséquence potentielle de supprimer le contexte (si cette fonction est la dernière à y faire référence).

Lorsque le compteur arrive à zéro, la fonction `ctx_release` est appelée. On y libère la mémoire qui avait été allouée pour stocker les localisateurs du pair (c'est désormais sans risque, puisque le contexte était le seul à utiliser ce tableau), ainsi que la mémoire occupée par le contexte lui-même (`struct shim6_ctx`).

Un compteur de références est aussi associé à chaque localisateur de l'hôte. Rappelons que ceux-ci sont gérés de manière globale et que leurs pointeurs se trouvent sous la responsabilité du module `addrconf`. Nous ne devons donc pas nous préoccuper de leur suppression, mais bien mettre à jour leur compteur de références de manière appropriée. *addrconf* définit (*include/net/addrconf.h*) :

```
in6_ifa_put(struct inet6_ifaddr*)
in6_ifa_hold(struct inet6_ifaddr*)
```

Ici aussi, nous souhaitons garder l'abstraction par rapport au module `addrconf` et définissons donc les macros suivantes :

```
#define loc_l_put(loc_l) in6_ifa_put((struct inet6_ifaddr*)loc_l)
#define loc_l_hold(loc_l) in6_ifa_hold((struct inet6_ifaddr*)loc_l)
```

L'argument `loc_l` est de type `shim6_loc_l` (voir section 3.1.2).

La liste globale de localisateurs de l'hôte n'est pas considérée comme une une référence supplémentaire vers les adresses, car celles-ci sont instantanément insérées ou retirées de la liste en même temps qu'elle le sont du module `addrconf` (au moment de la notification). `addrconf` ne libère une adresse qu'après la notification, si bien que nous pouvons la supprimer de notre liste de localisateurs de manière sûre.

Par contre lorsqu'un localisateur est choisi comme localisateur courant, une référence vers celui-ci est stockée dans le champ `ctx->lp_local`. Dans `__init_new_ctx`, les ULIDs sont pris comme localisateurs. La référence vers le localisateur correspondant à l'ULID est obtenue par

```
lookup_loc_l(&ctx->ulid_local);
```

Une fois de plus, il s'agit d'une macro qui cache l'accès au module `addr_conf`. Cette macro incrémente le compteur de références, ce qui assure que l'adresse ne disparaîtra pas tant qu'elle est utilisée par le contexte. Le garbage collector utilise donc `loc_l_put` pour libérer l'adresse.

La réalité est en fait un peu plus compliquée, car il peut se produire des conditions de parallélisme avec le protocole REAP, susceptible de remplacer le localisateur courant. Ce problème sera étudié au chapitre 4.

3.6.3 Autres opérations de suppression

Les autres opérations de suppression sont plus simples :

- Elimination du contexte dans les tables de hashage et, le cas échéant, dans la liste `init_list`. Puisque l'on supprime le contexte de la table `ulid_hashtable`, que l'on est en train de parcourir, il faut utiliser la version adaptée de la macro d'itération (section 2.1.5) :

  ```
  list_for_each_entry_safe()
  ```

- Suppression de l'entrée dans le répertoire `/proc/net/shim6` :

  ```
  remove_proc_entry(ctx->procfs_name, shim6_dir);
  ```

- Le protocole REAP manipule lui aussi des temporisateurs et diverses structures. Ce protocole est implémenté de manière indépendante, dans un fichier séparé (*reap.c*), et nous supprimons ses données simplement par un appel à

  ```
  del_reap_ctx(&ctx->reap)
  ```

  Cette fonction sera décrite au chapitre 4.

3.7 ICMPv6

Les messages ICMPv6 sont la plupart du temps destinés à véhiculer de l'information sur les localisateurs plutôt que sur les ULIDs. Pensons par exemple aux messages *Neighbor Discovery* qui remplacent le protocole ARP actuel. Un autre exemple est la notification d'indisponibilité d'un port (`port unreachable`). Ces paquets là ne pourront pas recevoir de modification de la part de shim, même si un état correspond au couple source-destination du paquet icmp. Par exemple, si une connexion possède les ULIDs <`1000:10:123::1, 2000:10:123::1`> et les localisateurs

<1000:20:123::1, 2000:20:123::1>, alors un paquet ICMP muni de la paire d'adresses <1000:10:123::1, 2000:10:123::1> devra partir tel quel sur le réseau, sans traduction avant la sortie.

On peut maintenant s'interroger sur la situation inverse : un paquet ICMP ayant les adresses <1000:20:123::1, 2000:20:123::1> devra-t-il être traduit en ULIDs à l'arrivée ? Il le faudra seulement si une notification doit être faite à la couche transport. Dans ce cas, la modification s'avérera plus complexe que dans le cas habituel : les paquets ICMPv6 de notification d'erreur reprennent une partie du paquet en erreur ; si ce paquet a été traduit par shim6 ou contient une extension d'en-tête shim6, il faudra le modifier de telle manière que la couche shim reste invisible pour les couches supérieures[32].

Dans l'implémentation, *tous* les paquets ICMP traversent la couche shim6 sans modification, et sans que celle-ci n'en tienne compte. Le support complet d'ICMP est donc laissé comme 'future work'.

3.8 Conclusion

Alors que conceptuellement les couches réseau et transport sont complètement distinctes, force est de constater que dans la réalité elles se chevauchent, à cause de l'accès aux fonctions de routage par la couche transport. Pour cette raison nous avons dû définir deux passages à travers la couche shim6.

Un échange d'initialisation shim6 est déclenché par l'envoi d'un paquet *de données* sur le réseau. Un contexte est alors créé, et un échange d'initialisation prend place. Notre implémentation prend en charge l'échange simultané, laissant la version sécurisée pour une version ultérieure. Cet échange permet à chaque équipement de connaître la liste de localisateurs de son correspondant.

Par la suite, la communication se poursuit normalement jusqu'à ce qu'une panne se produise, auquel cas une procédure d'exploration est lancée, ce qui est étudié au chapitre suivant. La communication après rehoming suppose une traduction des ULIDs en localisateurs, réalisée par la couche shim6. Finalement, nous avons vu que la suppression d'un contexte inutilisé demande des précautions particulières en matière de gestion de mémoire, en particulier pour ce qui est de l'arrêt des temporisateurs.

Chapitre 4

Le protocole REAP

Le chapitre précédent laissait en suspens le problème de la détection et récupération en cas de panne. La raison est que ceci est réalisé par un sous-protocole de `shim6`, appelé REAP (REAchability Protocol). Défini dans un draft séparé, [7], ce mécanisme était initialement conçu dans le but espéré de pouvoir être imbriqué dans d'autres protocoles. Bien que cette caractéristique soit actuellement encore sujette à discussions, il n'en reste pas moins qu'il s'agit là d'une partie relativement indépendante de `shim6`. Nous avons donc défini REAP dans des fichiers séparés, *reap.c* et *reap.h*.

Du point de vue de l'implémentation, l'entièreté de la machine à états du protocole (figure 4.1) a été implémentée. Il est cependant encore possible d'améliorer la vitesse de récupération en cas de panne par un choix mieux adapté des paires d'adresses utilisées lors de l'exploration, car il n'a pas encore été défini d'algorithme de gestion des sondes d'exploration. Le protocole actuel précise seulement à quel moment ces sondes devraient être émises.

L'implémentation proposée ici se limite donc à un algorithme simple d'exploration, qui sera décrit dans ce chapitre, et s'est avéré produire des résultats satisfaisants.

4.1 Détection de perte de joignabilité et récupération

On a vu à la section 3.1.3 que chaque hôte envoyait à son correspondant la liste de ses localisateurs. Tout au long de son fonctionnement, un contexte `shim6` gardera donc une liste de localisateurs propres (locaux), et une autre de localisateurs distants. En même temps, un contexte conserve les ULIDs actuellement utilisés, ainsi que les localisateurs courants, identiques aux ULIDs en début de connexion.

Le protocole REAP associe une machine à états à chaque paire de localisateurs utilisés à un instant donné (c'est-à-dire la paire de localisateurs courants). On a donc, à l'intérieur d'un contexte `shim6` *à l'état* `established` (Le protocole REAP n'est utilisable que lorsque l'initialisation `shim6` est terminée), un contexte REAP, qui possède son propre état. Les états REAP possibles sont les suivants :

- `operational` : la paire de localisateurs courante est opérationnelle, c'est-à-dire que le trafic émis par l'hôte en faisant usage de cette paire est bien reçu par son correspondant. Insistons sur le fait que les états REAP doivent être compris dans un sens uni-directionnel. Il est par exemple envisageable que les deux parties d'une communication utilisent chacune des paires différentes. Ceci peut être la conséquence de la présence de firewalls par exemple, qui rendent la communication possible dans un sens uniquement, obligeant à l'emploi d'un autre chemin, donc d'une autre paire de localisateurs pour le sens de communication inverse.

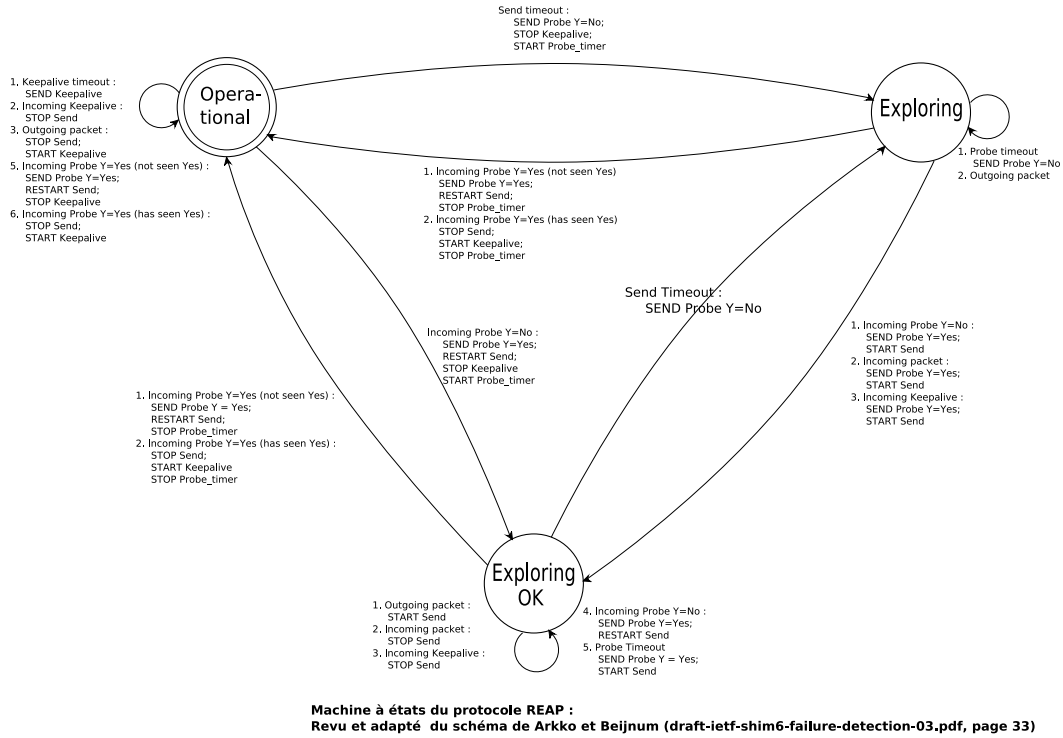


FIG. 4.1 – Machine à états de `REAP`

Chaque hôte traite donc uniquement la paire de localisateurs source et destination qu'il utilise pour l'envoi de paquets, et ne sait même pas quels sont ceux qui ont été choisis par le pair.
Ce n'est pas un problème car les paquets reçus sont rattachés à leur contexte au moyen de leur context tag, et non de leurs localisateurs (section 3.3).

- `exploring` : une perte de contact a été détectée (l'hôte ne voit plus de trafic entrant de la part de son correspondant). Dans cet état des sondes sont envoyées en utilisant diverses combinaisons d'adresses, dans l'espoir d'atteindre le correspondant avec l'une d'elles.
- `exploring_ok` : dans cet état du trafic est reçu de la part du correspondant, mais celui-ci ne voit pas les paquets émis par l'hôte. Comme dans l'état `exploring`, des sondes doivent être envoyées pour trouver une paire de localisateurs qui permette de joindre le correspondant.

La machine à états complète est présentée à la figure 4.1.

L'implémentation : Comme on l'a vu à la figure 3.2 la structure `shim6_ctx` comprend un champ `struct reap_ctx reap`. Cette structure, équivalent de `shim6_ctx` pour le protocole reap, est définie dans *reap.h* comme indiqué à la figure 4.2 :

Le contexte REAP est initialisé (`init_reap_ctx`) au moment où le contexte `shim6` passe à l'état `established`, car [7] précise (section 5.6) que le protocole ne peut fonctionner que dans ce cas.

L'état est stocké dans le champ `state` et vaut initialement `REAP_OPERATIONAL`, car le succès de l'échange d'initialisation `shim6` implique que la paire courante de localisateurs est opérationnelle.

La fonction `init_reap_ctx` se contente d'initialiser les variables d'état. REAP reste inactif jusqu'à ce que des paquets soient échangés.

```
struct reap_ctx {
        int                 state;
        struct timer_list   keepalive_timer;
        struct timer_list   send_timer;
        struct timer_list   probe_timer; /*For exploration*/

        int                 probe_ival; /*Time interval between probes*/

        spinlock_t          stop_send_lock;
        spinlock_t          stop_probe_lock;


        /*we add each incoming probe to rcvd_probes[rp_index],
          then rp_index=(rp_index+1)%NB_PROBES_RECVD, so that the oldest ones
          are replaced by the newest ones.*/
        struct rcvd_probe   rcvd_probes[NB_PROBES_RECVD];
        short               rp_index; /*This points to the place where we must
                                        add the next probe received, in term of
                                        index inside rcvd_probes*/
        /*sent probes with Y=yes*/
        struct sent_probe   sent_probes_y[NB_PROBES_SENT_Y];
        short               spy_index; /*idx in sent_probes_y*/
        /*sent probes with Y=no*/
        struct sent_probe   sent_probes_n[NB_PROBES_SENT_N];
        short               spn_index; /*idx in sent_probes_n*/

        int                 stop_timers:1; /*Flags.*/
};
```

FIG. 4.2 – Structure d'un contexte REAP

4.1.1 Détection de joignabilité

La technique employée est FBD (Forced Bidirectional Detection). Le principe est de vérifier que pour tout paquet émis dans une direction, un autre est émis dans l'autre direction. Naturellement, les protocoles des couches supérieures ne peuvent être forcés à continuellement échanger du trafic bidirectionnel. Pour cette raison, lorsqu'un hôte reçoit un paquet, REAP déclenche un temporisateur (*keepalive_timer*). Si à l'expiration de celui-ci (3 secondes), l'hôte n'a pas encore répondu par un autre paquet, alors un message de contrôle (*keepalive*) est envoyé. Le principe est illustré à la figure 4.3. Il faut noter que la charge supplémentaire impliquée par l'envoi des keepalives est très faible, car on n'en envoie que lorsque la communication cesse durant un moment (3 secondes). Dans ce cas, le dernier des deux intervenants à avoir reçu un paquet émet un keepalive, suite à quoi plus rien n'est envoyé (jusqu'à ce que les couches supérieures se remettent en activité). Cela nécessite que la réception d'un keepalive *ne déclenche pas* le temporisateur d'envoi d'un autre keepalive.

Maintenant que l'on est assuré qu'à chaque paquet sortant doit correspondre un paquet entrant ou un keepalive, nous sommes en mesure de détecter les pannes. En effet, si après un temps supérieur au *keepalive timeout* (3 secondes) additionné au délai de transmission dans le réseau, on ne reçoit rien de son correspondant, alors la procédure d'exploration peut être lancée. Le délai de déclenchement de la procédure d'exploration a donc été fixé à 10 secondes. Passé ce délai, le contexte REAP bascule à l'état `exploring`.

Ce basculement est assuré par un second temporisateur `send_timer`, armé chaque fois qu'un paquet est émis. Il est arrêté lorsqu'un message (paquet quelconque ou keepalive) est reçu du pair, sinon il expire après dix secondes et provoque le passage à la procédure d'exploration.

La gestion des temporisateurs `keepalive` et `send` suppose que le protocole REAP soit notifié des entrées et sorties de paquets appartenant au contexte `shim6` correspondant. Nous avons vu au

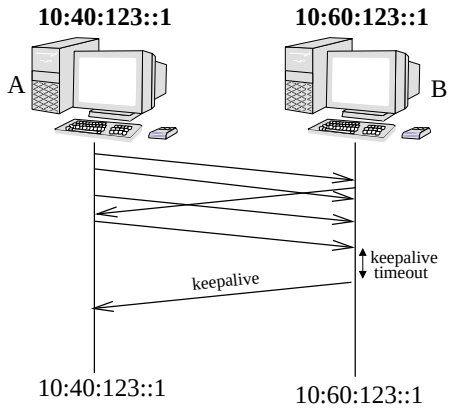


FIG. 4.3 – Communication bidirectionnelle forcée

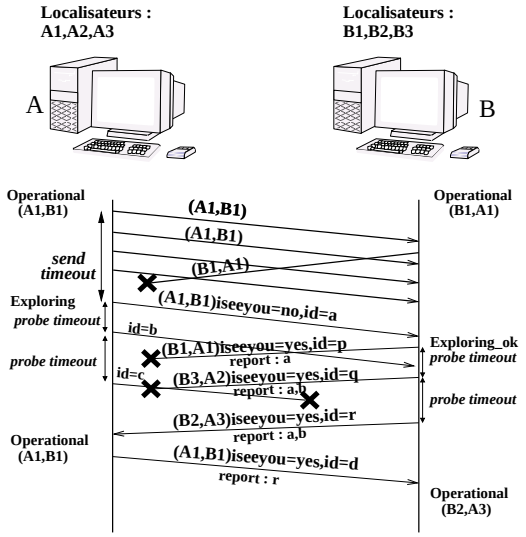


FIG. 4.4 – Processus d'exploration

chapitre précédent que `shim6` rattachait tous les paquets entrants et sortants à un contexte (si un tel contexte existe). Une fois ce contexte `shim6` identifié, le sous-contexte REAP est notifié par la fonction `reap_notify_in()` ou `reap_notify_out()`.

4.1.2 Procédure d'exploration

Un exemple de fonctionnement du processus d'exploration se trouve à la figure 4.4. Cet exemple se veut le plus général possible afin de permettre une compréhension intuitive du protocole dans son ensemble. Le lecteur intéressé par le comportement du protocole dans une situation précise peut consulter la machine à états de la figure 4.1. Il est également possible de faire des expériences en faisant usage du CD fourni avec ce mémoire.

Dans la figure 4.4, on observe une situation initiale où les deux hôtes A et B possédaient une paire de localisateurs opérationnels. On fait bien sûr l'hypothèse ici qu'un contexte `shim6` a été préalablement établi, sans quoi, comme indiqué plus haut, REAP ne peut être utilisé.

Tout à coup les paquets de B cessent d'arriver à A. Par contre la communication inverse continue

à fonctionner. Nous avons choisi de présenter le cas d'une coupure unidirectionnelle pour la clarté du schéma. Le cas bidirectionnel est cependant très semblable.

Puisque A ne reçoit plus rien pendant 10 secondes (*send timeout*), il passe en état exploring, et envoie donc une série de sondes (*probes*). Une sonde contient :

- Un identifiant sur 28 bits : cet identifiant est ensuite recopié par le receveur dans un 'rapport de réception', qui atteste la bonne réception de cette sonde. Il n'est en effet pas suffisant d'indiquer que l'on a reçu une sonde : il faut également préciser laquelle, de telle manière que l'émetteur (explorateur) soit capable de retrouver quelle paire d'adresses lui a permis de joindre son correspondant. L'utilisation de ce rapport de réception est décrite plus tard dans cet exemple.
- Un drapeau *I see you*. Il est mis à un lorsque l'émetteur 'voit' son correspondant, c'est-à-dire qu'il reçoit des messages de lui. En accord avec la définition des états donnée ci-dessus, ceci implique que le drapeau est à un lorsque REAP est à l'état `exploring_ok`, et à 0 lorsqu'il est à l'état `exploring`. Dans l'état `operational` aucune sonde n'est envoyée.
- Un rapport de réception indiquant les numéros de sondes qui ont été reçues. Ce rapport ne contient par contre aucune autre information sur les sondes reçues. C'est donc à l'émetteur de chaque sonde de garder en mémoire une table de ses sondes envoyées récemment, qu'il pourra consulter sur base des rapports.

Pour des raisons d'espace graphique, les données des sondes b et c ne sont pas indiquées sur le schéma. Mais puisque l'état de A est à ce moment `exploring`, on sait que le drapeau *I see you* est à zéro. Quant à la paire de localisateurs, elle est choisie différente à chaque sonde envoyée, le principe étant d'essayer de passer par autant de chemins différents que possible.

Le délai séparant l'émission de sondes n'est pas spécifié dans le draft [7], qui se limite à imposer un comportement de type 'exponential back-off'. Le choix de l'ordre des paires de localisateurs explorées (critique pour la vitesse de récupération en cas de panne) ne reçoit pas non plus d'indication précise. Certaines idées sont fournies à la section 5.3 du draft mais sont actuellement critiquées sur la mailing list du groupe de travail `shim6`.

Face à ce manque de précisions techniques, nous avons opté pour une solution partiellement inspirée de [40] (ce draft étant par ailleurs périmé et ne tenait pas compte de l'existence du protocole REAP), qui permet d'obtenir des résultats raisonnables au moyen d'une implémentation relativement simple : les premières sondes sont envoyées avec un délai initial de 200 ms, chaque fois multiplié par 1.25, pour respecter l'exigence d'exponential backoff. Lorsque le délai ainsi augmenté atteint 5 secondes, il cesse de croître, et une sonde est envoyée alors toutes les 5 secondes.

Nous choisissons la paire de localisateurs source et destination d'une sonde en prenant une adresse destination au hasard dans la liste de localisateurs distants. Ensuite l'adresse source est choisie en appliquant l'algorithme par défaut de sélection d'adresse source (RFC3484,[20]). Cet algorithme (RFC3484) était déjà implémenté dans le noyau. Le choix d'élire une destination au hasard est motivé par le fait que c'est un bon moyen d'essayer des adresses très différentes lors de l'envoi de sondes consécutives. Idéalement, il faudrait également s'assurer que le choix aléatoire soit complètement sans répétition (éviter de réessayer un couple d'adresses avant d'avoir essayé tous les autres). Ceci n'est pas garanti dans l'implémentation actuelle.

Une technique plus avancée serait d'imposer un ordre dans les couples testés sur base de préférences locales et de préférences annoncées par le pair lors de l'annonce des localisateurs. Une option est prévue par le protocole `shim6` pour associer des préférences aux localisateurs annoncés dans les messages d'initialisation et de mise à jour, mais elle n'est pas supportée dans notre implémentation.

Revenons donc à la figure 4.4. A était passé en état `exploring` sur expiration du temporisateur *Send* (selon le comportement imposé par la machine à états, fig. 4.1). B continue à croire que tout va bien car il continue à recevoir des paquets de A. Par contre lorsque A lui envoie sa première sonde,

il sait alors que son correspondant l'a perdu de vue, et qu'il devrait essayer de le joindre par d'autres chemins. Il passe donc à l'état `exploring_ok` (car lui voit encore A) et émet les sondes p, q et r en rafale (comme décrit ci-dessus). Les deux premières se perdent, mais la dernière (r) arrive à destination.

Sur réception de la sonde r, A apprend que B a bien reçu ses sondes a et b. Sur base des identifiants a et b, A peut retrouver les informations correspondant à chacune des sondes, en particulier les localisateurs utilisés pour ces sondes, dont il sait maintenant qu'ils sont joignables. Dans l'illustration ce sont les localisateurs de la sonde a qui sont choisis, donc en fait ceux d'origine. A peut revenir en état opérationnel.

Notons que les rapports de joignabilité joints aux sondes pourraient être utilisés dans l'élaboration d'un tri de paires d'adresses, pour une exploration future. Cependant le prototype ne le fait pas, puisqu'il ne possède pas de mécanisme de tri.

Notre prototype n'est donc intéressé que par un rapport, car ça lui suffit à choisir une quelconque paire de localisateurs opérationnels, et donc revenir à l'état `operational`. Cependant il devra examiner chaque rapport pour essayer d'en trouver un correspondant à "I see you = Yes". Ce n'est le cas d'aucune des sondes a et b. A en déduit que B n'a encore reçu aucune sonde de sa part avec "I see you = Yes" et croit donc encore que B ne le voit pas.

Sur cette conclusion, A émet une sonde "I see you = Yes" à B, lui permettant de repasser à son tour à l'état `operational`. B sait maintenant qu'il est capable de joindre A, mais doit encore trouver avec quelle paire de localisateurs. Or à la lecture de la sonde d, il apprend que la sonde r a été bien reçue par A. Cette sonde avait été envoyée avec les localisateurs (B2,A3) qui deviendront donc les localisateurs opérationnels courants. B observe ensuite également le drapeau "I see you" de la sonde r qu'il avait envoyée. Comme cette fois le drapeau vaut *Yes*, B sait que A est déjà repassé en état opérationnel.

Le mécanisme de récupération de panne est maintenant terminé : les deux hôtes sont revenus en état `operational`. A n'a pas été affecté par cet évènement (il utilise la même paire de localisateurs), mais B possède maintenant des localisateurs différents des ULIDs. La couche `shim6` devra désormais effectuer une modification des paquets passant par cette couche.

4.2 Gestion des sondes

Les informations sur les sondes sont stockées dans trois tampons circulaires (fig. 4.2) :

- `rcvd_probes` : ce tampon mémorise les numéros des sondes reçues. Ces numéros sont recopiés dans les rapports de réception joints aux sondes envoyées par l'hôte. Il y a cependant une mesure de précaution à prendre : si deux pannes successives se produisent, il ne faut plus annoncer au pair que l'on a bien reçu les sondes qui datent de l'exploration précédente. Pour cette raison, la structure `struct rcvd_probe` contient un identifiant de sonde, mais aussi la valeur qu'avait la variable `jiffies` au moment de la réception de la sonde. En accord avec [7], les sondes âgées de plus de `REAP_SEND_TIMEOUT` (actuellement 10 secondes) ne sont pas prises en compte dans les rapports de réception envoyés.
- `sent_probes_y` : y sont gardées les sondes émises avec le drapeau 'I see you=1'.
- `sent_probes_n` : Identique au tampon précédent, mais on y garde les sondes ayant le drapeau 'I see you=0'.

Les deux tampons de sondes émises contiennent des structures `struct sent_probe`, qui reprennent l'identifiant de sonde et la paire de localisateurs utilisée pour celle-ci. Si une sonde est reçue pendant que REAP se trouve en l'état `exploring` ou `exploring_ok`, il est possible de retrouver

les localisateurs qui ont servi à la construire. Ces localisateurs deviennent les localisateurs courants et l'on peut revenir à l'état `operational`.

Lorsqu'une entrée de tampon n'est pas utilisée, elle est placée à -1, car c'est une valeur qu'un identifiant ne peut jamais prendre (il n'occupe que 28 bits, et l'implémentation assure que les 4 bits de poids faible sont à zéro pour un identifiant valide).

La taille des tampons doit être choisie en fonction du taux d'envoi de sondes. Par exemple si l'on envoie 10 sondes en rafale, et que les tampons sont de taille 5, alors 5 sondes sont perdues : en effet si l'émetteur accuse réception de la première sonde envoyée, le receveur ne pourra retrouver l'information correspondante, qui aura été écrasée par celle d'une sonde émise plus récemment. Notre implémentation prévoit des tampons de taille 5. Ceci s'est avéré suffisant pour les expérimentations effectuées. Une taille si petite se justifie par le fait que les sondes doivent être émises en respectant un backoff exponentiel, pour éviter une "tempête de sondes" sur le réseau.

4.3 Compteurs de références pour les localisateurs locaux

Comme on l'a vu au chapitre précédent, les localisateurs de l'hôte sont des adresses gérées par le module *addrconf*. Nous avons vu également (section 3.6.2) que le nombre de références vers une adresse doit être contrôlé par les deux macros

```
loc_l_put(loc_l)
loc_l_hold(loc_l)
```

Par ailleurs, la structure `sent_probe`, utilisée pour mettre en tampon les données des sondes envoyées, utilise des pointeurs vers les adresses :

```
/* We use directly the pointers to addresses to avoid unnecessary lookup
 * However, it demands of course extra care...(refcnt usage)*/
struct sent_probe {
        int id;
        shim6_loc_l* laddr;
        shim6_loc_p* paddr;
};
```

Le pointeur vers le localisateur du pair ne demande pas de protection particulière parce qu'il référence une adresse qui appartient au contexte `shim6`. Actuellement, cette adresse reste valide aussi longtemps que le contexte existe. Néanmoins, si l'on souhaite implémenter l'option de mise à jour des localisateurs du pairs (sur base d'un message reçu de lui), alors le comportement correct sera d'effacer la mémoire de sondes envoyées, puisqu'elles deviendront toutes incorrectes. Donc même dans ce cas la situation est simple.

Le pointeur vers l'adresse locale est obtenu dans `probe_handler` (routine de traitement du temporisateur d'envoi de sondes), par la macro `lookup_loc_l` (décrite à la section 3.6.2). Nous avons vu que cette macro se charge d'incrémenter le compteur de référence de l'adresse. Il faudra dès lors appeler `loc_l_put` soit lorsqu'un problème fait que la sonde n'est finalement pas envoyée (débranchement soudain du fil par exemple), soit lorsque l'entrée du tampon est remplacée par une autre (car c'est à ce moment que la référence est effacée), soit finalement au moment de l'élimination du contexte `shim6` par le garbage collector.

Le premier cas est le plus simple : Si un problème se produit lors de l'envoi d'une sonde (donc à l'intérieur de `send_probe`), il est fait un saut vers la partie "gestion d'erreurs" de la fonction (`goto error`), et c'est là que `loc_l_put` est appelé.

Les second et troisième cas (remplacement et élimination de contexte) se font concurrence : une entrée dans un tampon peut être libérée par écrasement ou par élimination de contexte. Il y a donc

deux voies possibles pour libérer une même adresse, ce qui pourrait mener à la libérer deux fois ! C'est pour cette raison que le champ `dead` a été ajouté au contexte `shim6`. Dans le garbage collector de `shim6`, on trouve les instructions suivantes :

```
spin_lock(ctx->lock);
ctx->dead=1;
spin_unlock(ctx->lock);
loc_l_put(ctx->lp_local);
/*Deleting reap related elements*/
if (ctx->state==SHIM6_ESTABLISHED)
        del_reap_ctx(&ctx->reap);
```

(La fonction `del_reap_ctx` appelle `loc_l_put` pour toutes les adresses stockées dans les tampons de sondes émises)

Et dans la fonction `send_probe` :

```
spin_lock(&ctx->lock);
/*Before to replace old probe info, we must free the
  memory used by the local address : If the context is dead,
  previous address has already been freed (or will be soon), but not
  the new one*/
if (!ctx->dead && rctx->sent_probes_y[rctx->spy_index].id!=-1)
        loc_l_put(rctx->sent_probes_y[rctx->spy_index].laddr);
else if (ctx->dead) {
        loc_l_put(src);
        spin_unlock(&ctx->lock);
        return;
}
spin_unlock(&ctx->lock);
```

Deux situations sont possibles, du point de vue de REAP :

- Le contexte n'est pas 'mort' et l'emplacement que l'on s'apprête à occuper n'est pas vide (si il était vide il n'y aurait pas d'écrasement, donc rien à libérer) : alors il suffit de libérer l'ancienne adresse puis l'écraser. Dans ce premier cas, le garbage collector n'a pas pu s'exécuter avant, sans quoi `dead` vaudrait 1. Il peut par contre s'exécuter juste après, mais alors c'est la nouvelle adresse qui sera libérée, ce qui est correct. S'il s'exécute en même temps, il se bloquera sur le verrou, ce qui nous laissera le temps d'effectuer la manoeuvre d'écrasement proprement.
- Le contexte est mort (`dead==1`), mais alors on sait que le garbage collector a déjà libéré l'adresse que l'on s'apprête à écraser, ou est en train de le faire. Nous ne devons donc plus le faire. Par contre, nous nous apprêtions à envoyer une sonde pour laquelle nous avions réservé une référence vers l'adresse source (`src`). Puisque nous apprenons que le contexte est mort, il devient inutile d'envoyer la sonde, il faut donc libérer cette fois `src`, et sortir directement de la fonction. Notons qu'il serait *incorrect* d'écraser l'ancienne adresse dans ce cas, car le garbage collector libère le verrou dès que `dead` vaut 1 (pour des raisons d'efficacité). L'effet de ceci est que si l'on faisait un écrasement, il pourrait se produire *avant* la libération par le garbage collector, provoquant une libération double pour une adresse, et une absence de libération pour l'autre.

4.4 Prise en compte de l'état d'une adresse

Dans l'implémentation actuelle, nous ne tenons pas compte de l'état des adresses gérées par le module `addrconf`. Ce n'est pas indispensable, car la seule chose qui soit vraiment nécessaire pour assurer la stabilité du système est de toujours prendre garde d'utiliser des pointeurs valides. Si une adresse devient inaccessible, ce fait est détecté par le protocole REAP, avec comme seul inconvénient que cela nécessite une expiration de temporisateur REAP (10 secondes).

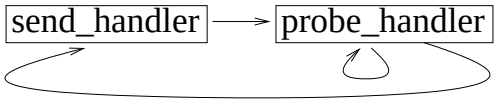


FIG. 4.5 – Déclenchement mutuel des temporisateurs REAP

Une possible amélioration future de l'implémentation serait donc d'observer l'état des adresses (stocké dans la structure `inet6_ifaddr`, voir section 2.4), de manière à déclencher directement un processus d'exploration si une adresse devient invalide. En réalité, il serait même bénéfique de déclencher une exploration dès qu'une adresse devient périmée (rappelons qu'une adresse, avant de ne plus pouvoir être utilisée, passe par une période où elle est 'périmée', voir figure 1.5, page 11). Cette manière de faire permettrait de continuer à utiliser l'adresse pendant le temps de l'exploration, et d'en obtenir une nouvelle sans même passer par une période de coupure. Le saut d'adresse serait complètement invisible pour les couches supérieures, car aucun paquet ne serait perdu.

4.5 Suppression d'un contexte REAP

Il a déjà été dit que la suppression se faisait au moment de supprimer un contexte `shim6`, par un appel à la fonction `del_reap_ctx`. Celle-ci, comme nous l'avons vu, libère les adresses pointées dans les tampons de sondes envoyées.

Une autre tâche de cette fonction est la suppression des temporisateurs, comme dans `shim6`, qui possède le même caractère délicat. Ici une difficulté supplémentaire est que deux temporisateurs s'arment mutuellement, comme illustré à la figure 4.5.

Si deux temporisateurs s'arment mutuellement, cela peut être un symptôme d'utilisation d'un temporisateur inutile : en effet, si les deux temporisateurs ne fonctionnent jamais en même temps, il est possible d'en utiliser un seul, en changeant éventuellement la routine de traitement du temporisateur si l'on veut simuler l'emploi de deux temporisateurs. La routine de traitement se change simplement en modifiant le pointeur `function` de la structure `timer_list`. C'est ce qui a été fait pour `shim6` (chapitre 3), où deux routines, `shim6_retransmit` et `shim6_hold_down_handler`, se partagent le même temporisateur.

Ici c'est impossible car les deux temporisateurs peuvent fonctionner en même temps dans l'état `exploring_ok` (voir la machine à états de la figure 4.1).

Si le graphe de déclenchement des temporisateurs (tel celui de la figure 4.5) ne contient pas de cycles, il est possible d'arrêter l'ensemble proprement en arrêtant d'abord le temporisateur le plus à gauche du graphe, puis respecter l'ordre d'arrêt indiqué par les flèches.

Encore une fois, pour notre infortune, nous voilà en présence d'un cycle. La solution que nous avons choisie est de généraliser la solution obtenue au chapitre précédent. Par commodité, nous reprenons ici la solution finale obtenue au chapitre précédent (donc pour les temporisateurs de `shim6`), où la portion de gauche est le code de la routine du temporisateur, et la portion de droite le code du garbage collector :

```
lock_hold=
  spin_trylock(ctx->stop_retr_lock);
if (!ctx->stop_retr_timer)
  add_timer(&ctx->retransmit_timer);
if (lock_hold)
  spin_unlock(ctx->stop_retr_lock);
```

```
ctx->stop_retr_timer=1;
spin_lock(ctx->stop_retr_lock);
del_timer_sync(
  &ctx->retransmit_timer);
spin_unlock(ctx->stop_retr_lock);
```

```
lock_hold=
  spin_trylock(&rctx->stop_probe_lock);
if (!ctx->stop_timers) {
  add_timer(&rctx->send_timer);
  add_timer(&rctx->probe_timer);
}
if (lock_hold)
  spin_unlock(&rctx->stop_probe_lock);
```

```
lock_hold=
  spin_trylock(&rctx->stop_send_lock);
if (!ctx->stop_timers)
  add_timer(&rctx->send_timer);
if (lock_hold)
  spin_unlock(&rctx->stop_send_lock);
```

FIG. 4.6 – Routines `probe_handler` (gauche) et `send_handler` (droite)

```
rctx->stop_timers=1;
spin_lock(&rctx->stop_send_lock);
spin_lock(&rctx->stop_probe_lock);
del_timer_sync(
  &rctx->probe_timer);
del_timer_sync(
  &rctx->send_timer);
spin_unlock(&rctx->stop_probe_lock);
spin_unlock(&rctx->stop_send_lock);
```

FIG. 4.7 – fonction `del_reap_ctx`

Pour rappel, le principe reposait sur l'emploi de la fonction `spin_trylock` par la routine du temporisateur, et le fait que l'échec dans l'obtention du verrou par la routine impliquait que c'était le garbage collector qui l'avait obtenu. Ce raisonnement suppose l'emploi d'un verrou spécifique réservé à cet usage.

Si l'on souhaite maintenant utiliser un seul verrou pour deux routines, l'échec d'acquisition de `spin_trylock` peut être dû à l'acquisition du verrou par le garbage collector, mais cela peut aussi être dû à une obtention par l'autre routine. Nous ferons donc appel cette fois à un verrou par routine. Par contre il est inutile d'utiliser un drapeau supplémentaire, puisque l'on souhaite arrêter les deux temporisateurs au même moment. La solution finale est présentée aux figures 4.6 et 4.7.

Pour ce qui est du temporisateur `probe_timer`, les difficultés liées au réarmement sont maîtrisées par la même technique que celle utilisée au chapitre précédent.

Voyons maintenant ce qui se produit si la routine `probe_handler` essaye de réarmer le temporisateur `send_timer` :

- Si la routine a pu acquérir le verrou `stop_probe_lock`, alors `del_reap_ctx` devra attendre qu'il se libère pour exécuter `del_timer_sync`. Entretemps, le temporisateur `send` aura été réarmé complètement, si bien que `del_timer_sync` n'aura plus qu'à le désactiver. Notons que c'est le verrou `stop_probe_lock` qui protège le temporisateur `send`.
- Si la routine ne parvient pas à acquérir le verrou, alors il a été acquis par `del_timer_sync`. Mais alors le drapeau d'arrêt des temporisateurs vaut 1 et aucun temporisateur ne sera réarmé.

Un raisonnement symmétrique nous conduit à demander les deux verrous dans `del_reap_ctx` avant d'arrêter les temporisateurs.

Dans la discussion ci-dessus, nous avons omis de parler du temporisateur `keepalive`. C'est parce qu'il s'agit d'un temporisateur indépendant, qui ne se réarme pas. On l'arrête par un appel à `del_timer_sync` sans artifice supplémentaire.

Un autre point qui a été omis est que les temporisateurs peuvent être armés, outre par eux-mêmes, par des fonctions du module, exécutées par exemple au moment de l'envoi ou de la réception d'un paquet (c'est le cas du temporisateur `keepalive`). Ces fonctions apprennent qu'elles ne doivent pas

relancer les temporisateurs grâce au drapeau `dead` du contexte `shim6` : Si une fonction arme un temporisateur avant que ce drapeau ne soit activé, il sera arrêté par le mécanisme décrit ci-dessus. Sinon elle verra que le drapeau est activé et n'exécutera pas la fonction.

Il faut assurer l'atomicité de la vérification du drapeau `dead`, ce qui se fait en se mettant sous la protection du verrou `ctx->lock`, sur lequel le garbage collector se synchronise. Par exemple, on trouve dans la fonction `reap_notify_out` :

```
spin_lock_bh(&ctx->lock);
if (!ctx->dead) add_timer(&rctx->send_timer);
spin_unlock_bh(&ctx->lock);
```

4.6 Conclusion

Le protocole REAP prend en charge ce qui concerne la détection d'une panne de lien et l'exploration de chemins alternatifs, au moyen d'un envoi de sondes munies de paires d'adresses différentes.

La détection de pannes est assurée par le mécanisme FBD (Forced Bidirectional Detection), dans lequel on garantit l'envoi d'une réponse à tout message entrant ou sortant, ce qui permet de déduire une panne de l'absence de réponse.

L'exploration d'adresse se fait au moyen d'un envoi de sondes vers diverses paires d'adresses. L'ordre d'exploration ainsi que l'intervalle entre l'envoi des sondes ne sont pas encore clairement définis dans les drafts, ce qui nous a menés à choisir une solution provisoire, qui produit cependant des résultats satisfaisants.

Comme nous l'avons déjà vu pour les contextes `shim6`, l'élimination d'un contexte `REAP` suppose également des précautions particulière, avec la difficulté supplémentaire présentée par la présence de temporisateurs qui s'arment mutuellement. Nous avons proposé dans ce chapitre une manière d'arrêter correctement ce type de temporisateurs, par l'usage de deux verrous non bloquants.

Conclusions

L'Internet se trouve actuellement dans une période de transition vers le protocole IPv6. Cette transition suppose de pouvoir fournir aux différents intervenants des domaines réseaux des solutions qui répondent à leurs besoins. Idéalement, il serait souhaitable de profiter de cette transition pour améliorer la qualité des services, l'efficacité et la sécurité globales de l'Internet.

L'une des manières d'améliorer la fiabilité des réseaux est de pratiquer le multihoming, d'une manière qui permette aux connexions établies de survivre aux pannes de lien. `Shim6` est l'un des moyens qui répond à cette exigence, à travers la dissociation des sémantiques d'identifiant et de localisateur d'une adresse IPv6. De plus, `shim6` a l'avantage de ne pas nécessiter de modifications dans les protocoles transport.

L'implémentation réalisée dans le cadre de ce mémoire nous permet de conclure qu'il est effectivement possible de voir une connexion survivre à travers une panne de lien, en ne modifiant pas les protocoles transport. Ceci est réalisé concrètement à travers un double passage dans la couche shim6, une première fois pour le routage, la seconde pour la traduction d'adresses si nécessaire. Le prototype dans sa version actuelle permet déjà de nombreuses expérimentations. Il a notamment été possible d'observer le rétablissement d'une connexion ssh après désactivation de l'interface utilisée. L'usage du système de fichiers `procfs` nous permet de visualiser les résultats des expériences menées, en combinaison avec l'usage d'un sniffeur de paquets (un exemple, pour l'expérience faite avec ssh, se trouve en annexe A).

Contributions de ce mémoire : Le but de ce mémoire était d'examiner les aspects d'implémentation liés à l'insertion de la couche `shim6` dans un Kernel Linux. Afin d'éviter de se disperser dans des aspects non spécifiques à shim6, telle la sécurité, nous avons fait certaines hypothèses (présentées au chapitre 3) permettant d'examiner précisément le principe même du multihoming par shim6. L'implémentation fournie (disponible sur le CD joint au mémoire) réalise donc une version fonctionnelle de shim6, laissant cependant les fonctions de sécurité comme partie modulaire à joindre dans une version ultérieure.

A travers la réalisation de l'implémentation, nous avons été amenés à proposer diverses solutions concrètes pour rendre efficace la gestion de `shim6` dans le noyau :

- La version actuelle de la définition de `shim6` propose de gérer une liste de localisateurs pour chaque connexion shim6 avec un pair. Notre travail suggère de faire appel au module d'autoconfiguration d'adresses déjà existant dans le noyau comme source d'informations sur les adresses disponibles localement. Nous considérons que le fait de garder une liste pour chaque connexion devrait plutôt être une option (non implémentée), tandis que par défaut les localisateurs locaux sont gérés de manière globale (c'est ce qui a été implémenté). Cela permet une économie de mémoire RAM.
- Au moment d'émettre un paquet sur le réseau, la couche application ne spécifie habituellement que l'adresse destination. Dans ce cas, la couche IPv6 se charge d'attribuer l'adresse source.

Cela pose un problème pour la nouvelle couche shim6, qui voit arriver des paquets ayant une adresse source indéfinie. Ce mémoire propose un mécanisme par lequel la couche shim6 essaye d'imposer elle-même l'adresse source si possible, dans l'objectif de réutiliser autant que possible des états précédemment créés. Cette solution est exposée à la section 3.1.1.

– Un noyau Linux doit normalement pouvoir être compilé pour diverses plateformes, en particulier des multiprocesseurs. Il faut donc toujours tenir compte de conditions d'exécution parallèle, c'est pourquoi nous discutons de manière étendue le problème du parallélisme, et décrivons les choix qui ont été faits de manière à assurer un bon fonctionnement de l'implémentation dans un environnement d'exécution parallèle. Il faut cependant préciser que les tests ont été faits sur un mono-processeur, et qu'il peut donc encore exister des bugs non détectés au niveau du parallélisme.

Possibilités de poursuite du travail effectué : Plusieurs hypothèses, décrites au début du troisième chapitre, ont été posées pour la réalisation de l'implémentation. Une suite possible du travail serait de supprimer ces diverses hypothèses, de manière à obtenir en définitive une implémentation utilisable par le public.

Si un tel but est poursuivi, l'une des priorités devrait être l'ajout des fonctions de *sécurité*, à savoir la méthode d'initialisation avec nonces et la vérification des groupes d'adresses par HBA/CGA.

La *robustesse* du protocole peut aussi être améliorée en supportant l'annonce de nouvelles listes de localisateurs à son correspondant, ou la récupération de contexte en cas d'élimination de celui-ci chez le pair.

D'un point de vue théorique cette fois, beaucoup d'avancées sont encore possibles dans le domaine de l'exploration de paires d'adresses. Nous avons vu que le document [7] définissant le protocole REAP n'était pas encore complet, raison pour laquelle notre travail propose une solution opérationnelle transitoire. Il serait donc intéressant :

– De réfléchir à une manière optimale de définir les *délais d'envoi des sondes*. Des sondes trop rapprochées chargeraient le réseau avec des messages de signalisation, ce qui n'est pas souhaitable. Par ailleurs si elles sont trop espacées, les protocoles de couche supérieures finissent par considérer que la connexion est perdue.
– D'imaginer un *ordre optimal d'essai des paires d'adresses*, par exemple sur base d'une modification de l'algorithme RFC3484[20]. Au moment d'écrire ces lignes, un draft vient de paraître à ce sujet, où M. Bagnulo propose un algorithme de sélection de paires d'adresses, dans le cadre d'une utilisation dans `shim6` [12].

Un autre terrain d'action est la *définition d'une API pour shim6*. Ce mémoire a mis en avant l'utilité de fournir une fonction permettant une désactivation partielle de shim6 par des sockets serveurs (section 3.1.1). [32] propose plusieurs autres possibilités.

Dans tous les cas, l'implémentation fournie en annexe peut constituer un terrain d'expérimentation, ou une base fonctionnelle que l'on peut améliorer.

Le futur de shim6 : L'IETF continue à travailler activement sur le protocole `shim6`. Alors qu'une tendance dans IPv6 serait d'éliminer l'usage des NATs, il est cependant question, pour shim6, de permettre à des routeurs du réseau de réécrire les localisateurs des paquets qui passent, à des fins de Traffic Enfineering [31]. Ce n'est cependant qu'une hypothèse actuellement.

Récemment, une décision de ARIN (American Registry for Internet Numbers) de commencer à distribuer des adresses de type PI a suscité de vives réactions sur la mailing list `shim6` : Les blocs d'adresses PI, Provider Independent, sont, par opposition aux blocs PA, attribués au demandeur sans

tenir compte du fournisseur. Un risque soulevé dans les discussions est l'augmentation de la taille des tables BGP dans l'Internet, ce que l'on cherchait précisément à éviter. Pour le mécanisme `shim6` aussi, les adresses PI poseraient des difficultés, puisque `shim6` tire parti du fait qu'un équipement dispose de plusieurs localisateurs, en particulier un par fournisseur. C'est ainsi que certains parlaient même de "la fin de shim6".

Dans un article sur l'état du multihoming, [22], écrit il y a un an, l'importance d'une évolution rapide de la recherche en matière de multihoming était soulignée. Les discussions actuelles et la récente décision d'ARIN semblent confirmer ce point de vue. Par ailleurs les professionnels du secteur souhaitent faire une transition harmonieuse et simple vers IPv6, ce qui motive certains à demander des adresses PI.

Nous espérons que les résultats proposés dans ce mémoire constituent un petit pas dans la direction d'une solution concrète à `shim6`, qui permette, à travers l'implémentation, de mener les expériences nécessaires à obtenir des résultats qui, mieux que relever de la conviction personnelle, montreraient clairement les forces et faiblesses de ce nouveau protocole.

Bibliographie

[1] *Understanding the linux kernel*. O'REILLY, 2000.

[2] *Linux Device Drivers, 2nd edition*. O'REILLY, 2001.

[3] *IPv6 Théorie et pratique, 4e édition*. O'REILLY, 2005.

[4] *The Linux Networking Architecture*. Prentice Hall, 2005.

[5] Arkko, Aura, Kempf, Mäntylä, Nikander, and Roe. Securing IPv6 Neighbor and Router Discovery. Atlanta, GA USA, septembre 2002. ACM Workshop on Wireless Security (WiSe 2002).

[6] J. Arkko. Failure Detection and Locator Pair Exploration Design for IPv6 Multihoming. Internet Draft, IETF, octobre 2005. <draft-ietf-shim6-failure-detection-01.txt>, expired.

[7] J. Arkko and I. Beijnum. Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. Internet Draft, IETF, décembre 2005. <draft-ietf-shim6-failure-detection-03.txt>, work in progress.

[8] J. Arkko, J. Kempf, B. Zill, and P. Nikander. Secure neighbor Discovery (SEND). RFC 3971, IETF, mars 2005.

[9] T. Aura. Cryptographically Generated Addresses (CGA). Bristol, UK, Octobre 2003. 6th Information Security Conference (ISC'03).

[10] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972, IETF, mars 2005.

[11] M. Bagnulo. Hashed Based Adresses (HBA). Internet Draft, IETF, octobre 2005. <draft-ietf-shim6-hba-01.txt>, expired.

[12] M. Bagnulo. Default Locator-pair selection algorithm for the SHIM6 protocol. Internet Draft, IETF, mai 2006. <draft-ietf-shim6-locator-pair-selection-00.txt>, work in progress.

[13] M. Bagnulo and J. Arkko. Functional decomposition of the multihoming protocol. Internet Draft, IETF, juillet 2005. <draft-ietf-shim6-functional-dec-00.txt>, expired.

[14] B.Carpenter. Internet Transparency. RFC 2775, IETF, février 2000.

[15] R. Bush and D. Meyer. Some Internet Architectural Guidelines and Philosophy. RFC 3439, IETF, décembre 2002.

[16] Jonathan Corbet. Driver Porting : The seq_file interface. *LWN.net*, 2003. http ://lwn.net/Articles/22355.

[17] M. Crawford. Transmission of IPv6 Paquets over Ethernet Networks. RFC 2464, IETF, décembre 1998.

[18] Cédric de Launois. *Unleashing Traffic Engineering for IPv6 Multihomed Sites*. PhD thesis, UCL, 2005.

[19] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, décembre 1998.

[20] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484, IETF, février 2003.

[21] Droms, Bound, Volz, Lemon, Perkins, and Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315, IETF, juillet 2003.

[22] Alan Ford. The Current State and Future Developments of IPv6 Multihoming. 2005. http ://www.ecs.soton.ac.uk/ ajf101/irp-ajf101-multihoming.pdf.

[23] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2893, IETF, août 2000.

[24] Kevin He. Byte and Bit Order Dissection.
Linux Journal, 2003. http ://www.linuxjournal.com/article/6788.

[25] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Adressing Architecture. RFC 3513, IETF, avril 2003.

[26] G. Kroah-Hartman. kobjects and krefs, lockless reference counting for kernel structures. In *Proceedings of the Linux Symposium, Volume Two*, Ottawa, Ontario, Juillet 2004.

[27] Erik Mouw. Linux Kernel Procfs Guide. Sources du noyau Linux, version 2.6.15, Documentation/DocBook/procfs-guide.tmpl, 2001.

[28] T. Narten and R. Draves. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 3041, IETF, janvier 2001.

[29] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP version 6 (IPv6). RFC 2461, IETF, décembre 1998.

[30] E. Nordmark. Shim6 Application Referral Issues. Internet Draft, IETF, juillet 2005. <draft-ietf-shim6-app-refer-00.txt>, expired.

[31] E. Nordmark. Extended Shim6 Design for ID/loc split and Traffic Engineering. Internet Draft, IETF, février 2006. <draft-nordmark-shim6-esd-00.txt>, work in progress.

[32] E. Nordmark and M. Bagnulo. Level 3 multihoming shim protocol. Internet Draft, IETF, septembre 2005. <draft-ietf-shim6-proto-03.txt>, expired.

[33] E. Nordmark and M. Bagnulo. Multihoming L3 Shim approach. Internet Draft, IETF, juillet 2005. <draft-ietf-shim6-l3shim-00.txt>, expired.

[34] Alessandro Rubini. The sysctl Interface.
Linux Journal, 1997. http ://www2.linuxjournal.com/article/2365.

[35] Rusty Russell. Unreliable Guide To Locking. Sources du noyau Linux, version 2.6.15, Documentation/DocBook/kernel-locking.tmpl, 2003.

[36] Rusty Russell. Unreliable Guide To Hacking The Linux Kernel. Sources du noyau Linux, version 2.6.15, Documentation/DocBook/kernel-hacking.tmpl, 2005.

[37] S. Thomson, C. Huitema, V. Ksinant, and M. Souissi. DNS Extensions to Support IP Version 6. RFC 3596, IETF, octobre 2003.

[38] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 2462, IETF, décembre 1998.

[39] Linus Torvalds and Amit Gud. Spinlocks. Sources du noyau Linux, version 2.6.15, Documentation/spinlocks.txt, 2005.

[40] I. van Beijnum. Shim6 Reachability Detection. Internet Draft, IETF, juillet 2005. <draft-ietf-shim6-reach-detect-01.txt>, expired.

Annexes

Annexe A

Contenu des fichiers de */proc/net/shim6*

```
guest1:~ # cat /proc/net/shim6/a4fc7200
State : established
Peer ulid : fec0:0:0:0:0:0:0:30
Local ulid : fec0:0:0:0:0:0:0:20
FII : 0
Peer locators list :
                fec0:0:0:0:0:0:0:30
                fe80:0:0:0:fcfd:ff:fe00:3
                fe80:0:0:0:fcfd:ff:fe00:2
                fec0:0:0:0:0:0:0:3
Local locators list :
                fe80:0:0:0:fcfd:ff:fe00:1       refcnt=2
                fe80:0:0:0:fcfd:ff:fe00:0       refcnt=5
                fec0:0:0:0:0:0:0:2      refcnt=2
                fec0:0:0:0:0:0:0:20     refcnt=5
Peer context tag : 00
Local context tag : 00
Init nonce : 8b5e4671
Age : 21 seconds
Preferred peer locator : fe80:0:0:0:fcfd:ff:fe00:2
Preferred local locator : fe80:0:0:0:fcfd:ff:fe00:0
refcnt : 1
REAP data :
         state : operational
         last probes recvd :
                 id = 60c38440, rcvd 59 seconds ago
                 id = c35b8630, rcvd 59 seconds ago
                 id = 748ad3c0, rcvd 22 seconds ago
         last probes sent (with iseeyou = yes):
                 id = 27906e30, fec0:0:0:0:0:0:0:20 -> fec0:0:0:0:0:0:0:30
                 id = 74521530, fe80:0:0:0:fcfd:ff:fe00:0 -> fe80:0:0:0:fcfd:ff:fe00:2
         last probes sent (with iseeyou = no):
                 id = fef4ed20, fec0:0:0:0:0:0:0:20 -> fec0:0:0:0:0:0:0:30
                 id = 99c66e50, fe80:0:0:0:fcfd:ff:fe00:0 -> fe80:0:0:0:fcfd:ff:fe00:2
                 id = f540ca30, fec0:0:0:0:0:0:0:20 -> fec0:0:0:0:0:0:0:30
```

FIG. A.1 – Etat shim6 pour l'hôte `guest1`, après rehoming

```
guest2:~ # cat /proc/net/shim6/a4d8e800
State : established
Peer ulid : fec0:0:0:0:0:0:0:20
Local ulid : fec0:0:0:0:0:0:0:30
FII : 0
Peer locators list :
                fe80:0:0:0:fcfd:ff:fe00:1
                fe80:0:0:0:fcfd:ff:fe00:0
                fec0:0:0:0:0:0:0:2
                fec0:0:0:0:0:0:0:20
Local locators list :
                fe80:0:0:0:fcfd:ff:fe00:2       refcnt=2
                fec0:0:0:0:0:0:0:3      refcnt=4
Peer context tag : 00
Local context tag : 00
Init nonce : 71e3088b
Age : 37 seconds
Preferred peer locator : fec0:0:0:0:0:0:0:2
Preferred local locator : fec0:0:0:0:0:0:0:3
refcnt : 1
REAP data :
         state : operational
         last probes recvd :
                 id = 27906e30, rcvd 75 seconds ago
                 id = 99c66e50, rcvd 38 seconds ago
                 id = 74521530, rcvd 38 seconds ago
         last probes sent (with iseeyou = yes):
                 id = c35b8630, fec0:0:0:0:0:0:0:30 -> fec0:0:0:0:0:0:0:20
                 id = 748ad3c0, fec0:0:0:0:0:0:0:3 -> fec0:0:0:0:0:0:0:2
         last probes sent (with iseeyou = no):
                 id = 60c38440, fec0:0:0:0:0:0:0:30 -> fec0:0:0:0:0:0:0:20
```

FIG. A.2 – Etat shim6 pour l'hôte `guest2`, après rehoming

Description des champs

Les divers champs d'un fichier de */proc/net/shim6* ont la signification suivante :

- `State` : l'état du contexte shim6. La plupart du temps c'est normalement `established`. Les états possibles d'un contexte `shim6` se trouvent à la figure 3.4 (page 54).
- `Peer/Local ulid` : adresse IPv6 utilisée comme identifiant respectivement pour le correspondant ou pour l'équipement local. Cette adresse est donc celle qui est vue par les couches supérieures (TCP par exemple).
- `FII` : Forked Instance Identifier. Cette valeur est présente en prévision d'une implémentation future du forking de contextes shim6. Puisque ceci n'est pas supporté dans l'implémentation actuelle, le FII vaut toujours 0.
- `Peer locators list` : c'est la liste de localisateurs du correspondant, reçue de lui au cours de l'initialisation, dans un message R2. Cette liste est propre à chaque contexte.
- `Local locators list` : liste des localisateurs locaux. Contrairement à ceux du correspondants, les localisateurs locaux sont mémorisés de manière centralisée au niveau du système. Plus encore, ils ne sont même pas propres à shim6, mais appartiennent au module d'autoconfiguration d'adresses. Pour cette raison (voir section 4.3), des compteurs de références sont utilisés, et leur valeur est affichée à des fins de debugging. Ils ont toujours une valeur minimale de deux (sauf si l'adresse est en cours de suppression), attribuée par le module `addrconf`. Ensuite `shim6` augmente cette valeur sur base du nombre de références qu'il garde vers une adresse. Par exemple dans la figure A.2, l'adresse `fec0::3` a une valeur de compteur de 4, ce

qui signifie que `shim6` possède deux références vers celle-ci. En effet, l'une d'elle se trouve dans le champ `Preferred local locator`, l'autre dans un tampon d'envoi de sondes.

- `Peer context tag` : le context tag qui sera inclus dans les en-têtes de paquets à destination du correspondant, après un rehoming (ce qui est le cas ici). Dans la version implémentée de l'échange d'initialisation, ce context tag est enregistré dans le contexte au moment de la réception d'un message I1.
- `Local context tag` : Attribué au moment de la création d'un nouveau contexte. Sa valeur est choisie à partir d'une variable globale initialisée à 0 (on voit donc que les deux contextes de l'exemple sont les premiers créés après le boot). Plus de détails sur l'attribution du context tag sont donnés à la section 3.1.3.
- `Init nonce` : Nonce d'initialisation, décrit à la section 3.1.3.
- `Age` : Temps écoulé depuis le dernier passage d'un paquet *de données* à travers le contexte. Cette information, calculée à partir du champ `lastuse` du contexte `shim6`, permet au garbage collector de choisir les contextes à supprimer (qui sont ceux âgés de plus de dix minutes).
- `Preferred peer/local locator` : Adresses IPv6 utilisées actuellement comme localisateurs. Ces adresses appartiennent nécessairement à la liste de localisateurs présentée plus haut. L'exemple illustre une situation après rehoming, c'est pourquoi les localisateurs sont différents des identifiants (ULIDs). Le contexte est donc en train de traduire tous les paquets qui sortent (et aussi ceux qui entrent car l'autre contexte possède également des identifiants différents des localisateurs). Il est important de voir qu'il n'est nullement nécessaire que les deux hôtes émettent leurs paquets avec la même paire de localisateurs. Au contraire, c'est justement ce qui permet d'utiliser des chemins différents dans les deux sens de communication (utile si l'une des voies est bloquée dans un sens, par un firewall par exemple).
- `refcnt` : C'est le compteur de références du contexte. Normalement il vaut toujours 1, et n'est incrémenté que lorsqu'il est utilisé par une fonction du module `shim6` (ce qui dure une fraction de seconde et ne pourrait donc être vu que par hasard).
- `REAP data` : REAP étant un sous-protocole de shim6, il possède un sous-contexte du contexte shim6. Les données affichées dans cette section sont manipulées dans le fichier *reap.c*.
- `state` : C'est l'état REAP, différent de l'état shim6, et correspondant à l'une des valeurs définies dans la machine à états de la figure 4.1 (page 68) : `operational`, `exploring` ou `exploring_ok`.
- `last probes ...` : Fournit le contenu des trois tampons de sondes. Ainsi que l'explique la section 4.2, les sondes reçues sont munies d'un timestamp, on observe ainsi dans la figure A.2 que la sonde d'identifiant `27906e3` appartenait à une autre procédure d'exploration que les deux suivantes. Les identifiants se terminent toujours par zéro car ce sont des nombres de 28 bits stockés sur les bits de poids fort d'un entier (32 bits). Pour les sondes envoyées, les localisateurs source et destination utilisés pour la sonde sont indiqués.