Responsable France Télécom R&D :
Olivier DUGEON
FTR&D/DAC/CPN
2, avenue Pierre Marzin
22307 Lannion

Correspondant ENSSAT :
Olivier BOEFFARD
ENSSAT
6, rue de Kérampont
22305 Lannion

Tutorat ingénieur

Étude du filtrage de paquets IP dans un routeur d'accès à un réseau MPLS

Study of IP packet filtering in a MPLS label edge router

Alexandre DAGAN

01/09/2000 - 31/07/2001

Logiciel et Système Informatique

Last compiled on 22 janvier 2002

Résumé

Résumé

Avec le développement croissant des technologies de l'information et de l'Internet en particulier, les grands opérateurs de télécommunications sont dans la nécessité de développer des solutions techniques visant à améliorer la qualité de services sur le réseau IP. C'est dans ce but, que le projet ASIA de France Télécom a vu le jour, et est maintenant relayé par un des projets européens du cadre IST : CADENUS. Le but de ces projets est de mettre en place la notion de qualité de service dans les réseaux IP. Une solution est d'utiliser le protocole MPLS afin d'accélérer le routage des paquets IP et de réaliser une gestion de trafic IP tout en assurant une qualité de service selon les besoins des différents utilisateurs.

Le but du projet est d'étudier, améliorer, modifier et implémenter la classification des paquets IP dans le routeur d'accès MPLS ainsi que la commande de cette classification. La plateforme de développement sera un PC sous environnement Linux.

Mots clés : MPLS, ATM, IP, TCP, UDP, noyau Linux, classification de paquets, FEC.

Abstract

To keep up with the fast evolution of the new information technologies, and in particular the Internet, the major telecommunication companies have to develop new technical solutions in order to increase and improve the quality of services on IP networks. The project of France Télécom named ASIA has been decided with this aim in view. It is now relayed by a European project : CADENUS.

The aim of both projects is to implement quality of service within the IP network. One solution is to use MPLS protocol in order to increase the speed of IP packet routing and make traffic engineering mechanisms efficient insuring the QoS required by the user's connection.

The aim of this project concerns the developement and the implementation of a label edge router on a PC Linux, and more particulary the study and the implementation of IP packet classification within the edge router as well as the control of this classification.

Keywords : MPLS, ATM, IP, TCP, UDP, Linux kernel, forwarding mechanism, labels.

Remerciements

Je tiens a remercier tout particulièrement Olivier DUGEON pour avoir fait de ce tutorat une expérience incomparable. Sa sympathie, sa disponibilité et ses compétences aussi vastes que variées ont été autant d'atouts pour mener à bien ce tutorat dans les meilleurs conditions. Je pensais connaître Linux, il m'en a appris encore plus. Merci.

Mes remerciements vont aussi aux membres de l'équipe CRP (Michel BOURBAO, Jaqueline et Pierre BOYER, Alain DUPUIS, Fabrice GUILLEMIN et Stéphane STATIOTIS) pour les moments de détente qui ont agrémenté ce tutorat.

Enfin je remercie Olivier BOEFFARD pour ses précieux conseils apportés à la rédaction de ce rapport ainsi qu'à l'intérêt qu'il a porté à mon travail.

Table des matières

Table des figures

Liste des tableaux

Glossaire

ATM : Asynchronous Tranfer Mode (Mode de transfert asynchrone)
CR-LDP : Constraint based Routed LDP
FEC : Forwarding Equivalence Class
FTR&D : France Télécom Recherche et Développement
IP : Internet Protocol
LDP Label Distribution Protocol
LEAP Layer Environment Accelerated Portability
LER Label Edge Router
LPE Label Process Engine
LSP Label Switched Path
LSR Label Switched Router
LTCS Label Trafic Control System
MPLS Multi-Protocol Label Switching
RFC Request For Comment
RNRT Réseau National de Recherche en Télécom
TCP Transfert Control Protocol
VC Virtual Channel
VCI Virtual Channel Identifier
VP Virtual Path
VPI Virtual Path Identifier

Introduction

Dans le cadre du tutorat de l'innovation, le centre de recherche de FTR&D à Lannion a accepté de m'accueillir au sein de la DAC (Direction de l'Architecture, de l'intégration et de la Commande de réseaux), dans le laboratoire CPN (Cœur de réseaux Paquets pour le NGN : **N**ew **G**eneration **N**etwork) au sein du groupe MCR (Machines et Commandes de Réseau) (cf. figure 1.). Le laboratoire CPN a

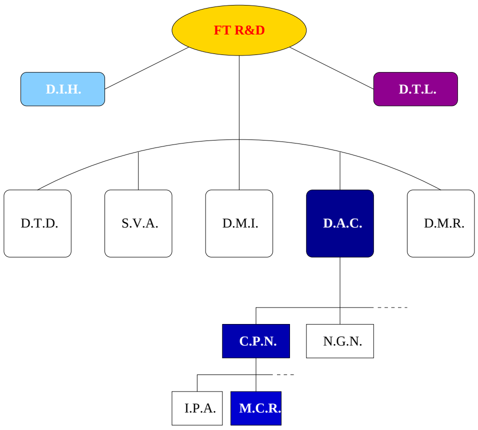

Fig. 1 – Situation du service DAC/CPN/MCR

pour mission d'étudier les techniques de transfert en mode paquet (ATM, IP, ...) et leur adéquation aux besoins de services au profit de France Télécom et de ses filiales. Il propose de nouvelles solutions, les implémente, les teste et contribue aussi à la normalisation.
Le sujet de ce tutorat porte sur un des éléments physiques qu'utilise une nouvelle technique de transfert garantissant une qualité de service sur IP : le routeur d'accès MPLS (Label Edge Router). Cette nouvelle génération de routeur s'appuie sur le protocole MPLS[1] et permettra à terme d'accélérer le routage et d'offrir des mécanismes de gestion de trafic. Le principe d'étiquetage des paquets à l'entrée d'un réseau MPLS repose sur un mécanisme important : la classification de paquets.
Le travail demandé consiste à étudier, améliorer, modifier et implémenter la classification des paquets IP dans le routeur d'accès MPLS ainsi que la commande de cette classification. Le travail se décompose en deux grandes étapes :

- Dans un premier temps, la réalisation d'une plateforme[2] de développement sous environnement Linux.
- Dans un deuxième temps, l'étude et l'implémentation des algorithmes de filtrage[3] de paquets existants en vue de leur amélioration et de leur adaptation pour la classification de paquets IP sous MPLS.

Ce document présentera dans un premier temps la branche recherche de France Télécom, puis l'intérêt de la qualité de service sur IP et la place que peut y jouer le protocole MPLS. Enfin pour terminer, il développe le travail réalisé et à venir.

[1]qui introduit la notion de qualité de service sur IP
[2]Routeur d'accès MLPS
[3]Ceux existant pour le routage classique, diffServ, firewall, proxy, ...

Chapitre 1

France Télécom R&D et la qualité de service

1.1 France Télécom R&D

Appelé autrefois CNET[1], France Télécom R&D est l'un des plus grands centres de recherche en télécommunications d'Europe. Il est implanté en France sur huit sites : Belfort, Caen, Grenoble, Issy-les-Moulineaux, Lannion[2], Rennes, Sophia Antipolis et La Turbie. La branche recherche de France Télécom possède aussi un laboratoire à l'étranger situé à Brisbane aux États Unis depuis quatre ans. Le site de Lannion a été créé en 1962 suite à une politique de décentralisation entamée a cette époque.

France Télécom R&D emploie environ 3900 personnes, dont près de 1500 travaillent sur le site de Lannion. Ses principales activités sont :

- le développement de nouveaux services
- le développement de services pour ses clients
- l'étude de nouveaux réseaux et de leur architecture

Parmi les dix DRD (Directions de Recherche et de Développement), neuf sont présentes sur le site Lannionais. On les retrouvent en détails dans le tableau 1.1.

1.2 Intérêt de la qualité de service sur IP

Depuis quelques années, l'essor de l'Internet suit une croissance exponentielle. Le trafic est en hausse constante et de nouveaux services voient sans cesse le jour. Mais toutes ces évolutions ont entraîné la nécessité de prendre en compte des contraintes de qualité[3] de service. Il devient donc primordial pour les opérateurs de télécommunications de disposer d'un réseau Internet capable de garantir des objectifs de qualité de service.

Plusieurs raisons les poussent :

- le marché florissant des applications grand public sur Internet,
- substituer les réseaux IP au réseau téléphonique classique,
- nécessité de rester compétitif sur les tarifs d'accès à l'Internet.

Le premier point concerne aussi bien les applications multimédia, qui exigent des temps de transfert de l'information constants et bornés, que les transmissions de données "classiques" qui, elles, ne tolèrent que très peu de pertes d'information.

Le second point, quant à lui, impose que les nouveaux services qui lui seront associés devront être de qualité au moins comparable à ceux du réseau téléphonique.

Enfin, le dernier point concerne plus les opérateur eux-mêmes, qui, au travers d'une meilleure maîtrise de certains paramètres[4] pourront ainsi rationaliser leur tarification, baisser les coûts de communication

[1] Centre National d'Étude des Télécommunications

[2] Le centre de Lannion fait partie des deux plus importants avec celui de Paris .

[3] Dans le cadre de la téléphonie sur Internet par exemple.

[4] qualité de service, volume des données échangées, multiplexage statistique

Direction de recherche et de Développement	Activités
Architecture, intégration et commande de réseau (DAC)	Commutation synchrone ATM, IP
Interactions humaines (DIH)	Technologies vocales Traitement de la parole et du son Dialogue intelligent
Services mobiles et systèmes radio (DMR)	Mobile UMTS
Services de diffusion multimédia (DMI)	Services Internet (ex : Voilà) Service Intranet
Services voix-données avancés (SVA)	Services vocaux Services du téléphone Services sur IP Télé-activités
Techniques logicielles (DTL)	Systèmes et réseaux de distribution Systèmes et réseaux de transmission optique Infrastructures des réseaux optiques Qualité fiabilité
Réseau Transport et Accès (RTA)	Support de distribution (Fibre optique, cuivre...)

TAB. 1.1 – Directions de recherche et développement sur le site de Lannion

et optimiser leur réseau.

L'un des rôles de DAC/CPN est justement d'étudier et de mettre en place de nouveaux mécanismes permettant d'introduire la notion de qualité de service sur le réseau IP qui sert de support à l'Internet.

Chapitre 2

Le problème à résoudre

2.1 Le but du tutorat

Mon tutorat est en fait la continuation d'un projet réalisé par un autre élève ingénieur de l'ENSSAT au cours des six mois précédant mon arrivée. Ce projet fait partie de deux autres projets distincts menés au sein de CPN.

Le premier se nomme ASIA pour "***A**ccelerated **S**ignalling for **I**P over **A**TM*". Il vise en priorité l'innovation suivante : constituer un réseau Internet haut débit capable de garantir des objectifs de qualité de service. Le réseau choisi est ATM enrichi d'une capacité de signalisation allégée capable d'établir, de renégocier la qualité de service et de rompre des connexions avec des temps de réponse très courts. C'est un projet RNRT qui s'est achevé en janvier 2001.

Le second est intitulé CADENUS pour ***C**reation **A**nd **D**eployment of **EN**d-**U**ser **S**ervices in Premium IP Networks.*
A terme le projet vise à automatiser la négociation de qualité de service entre usagers et réseau. Il est basé sur la notion de contrat[1].

Avant de rentrer plus en détail dans la définition de mon sujet de tutorat, il est nécessaire d'en expliquer le contexte et l'environnement en particulier la signalisation MPLS afin que le lecteur comprenne bien le travail qui m'a été demandé.

2.2 Le protocole MPLS : principes de fonctionnement

Le routage dans un réseau MPLS diffère d'avec un routage classique. Pour comprendre le fonctionnement du protocole MPLS, nous allons partir de l'exemple représenté à la figure 2.

Un paquet IP en provenance de l'utilisateur 1 arrive sur un des ports du routeur LER 1[2]. Pour arriver à destination, il doit transiter par le réseau MPLS auquel est connecté LER 1. LER 1 regarde dans sa table de routage (cf. table 2.) la correspondance entre l'adresse du réseau de destination et le label MPLS. Ici, il doit transmettre le paquet avec le label 3 sur son interface de sortie 1 (vers le routeur interne LSR 3[3]). Le paquet sera labellisé comme le montre la première étape de la figure 3.
Ensuite le paquet labellisé arrive sur le routeur LSR 3, qui regarde le label du paquet entrant et recherche dans sa table de labels la correspondance entre le label d'entrée et le label de sortie. Ensuite le paquet est relabellisé avec le nouveau label (ici 2) et renvoyé sur l'interface de sortie associée (ici 3).

[1]SLA : Service Level Agreement.
[2]LER 1 est un routeur d'accès au réseau MPLS.
[3]Le numéro de label est indépendant du numéro de routeur. Le fait qu'ils soient identiques dans l'exemple n'est que fortuit

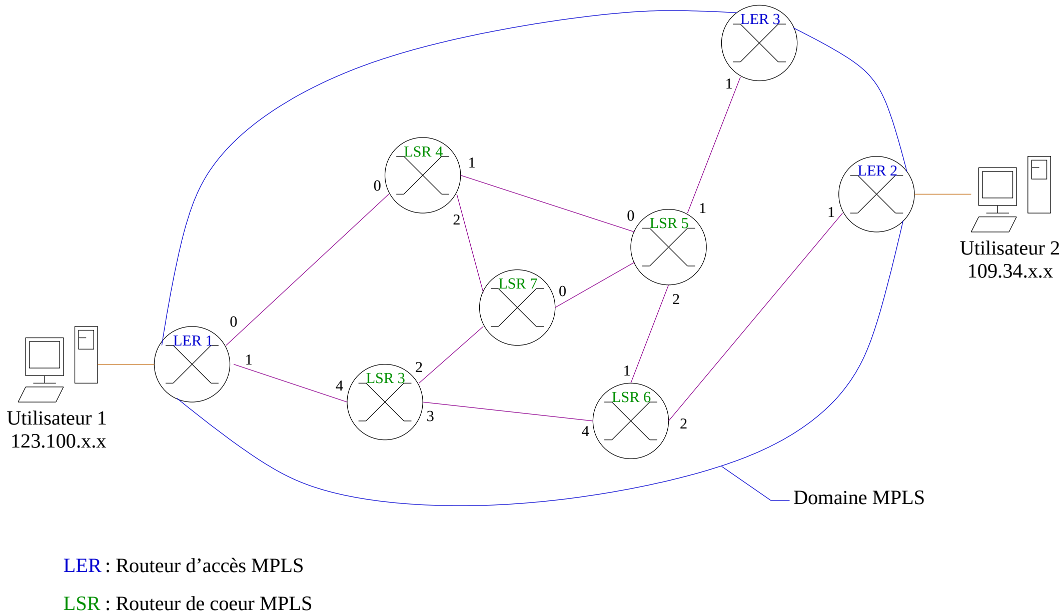

FIG. 2.1 – Schéma d'un réseau MPLS

Et ainsi de suite, jusqu'à atteindre le routeur de sortie, par exemple ici LER 2, où le label est supprimé avant que le paquet ne poursuive sa route jusqu'à l'utilisateur final.

Préfixe	Label	Port de sortie
109.34.x.x	3	1
160.55.x.x	4	0
123.97.x.x	5	1

Label in	If in	Label out	If out
3	4	2	3
2	4	4	2
1	4	3	2

TAB. 2.1 – Tables de routage du LER 1 (à gauche) et du LSR 3 (à droite)

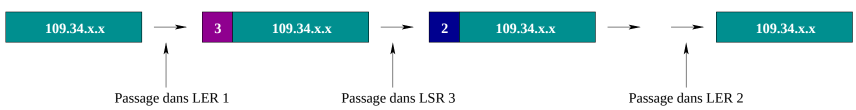

FIG. 2.2 – Évolution d'un paquet au sein du réseau MPLS

2.3 Routeur d'accès MPLS (Label Edge Router)

En routage classique (bond par bond), chaque routeur exécute l'algorithme dit du "plus long préfixe" sur sa table de routage afin de déterminer le prochain noeud (routeur) que doit emprunter le paquet arrivant. Cet algorithme est coûteux en temps CPU (il croit de façon exponentielle en fonction du nombre d'entrées dans la table de routage) et est exécuté à chaque paquet même si le résultat est identique pour tous les paquets d'une même connexion. Dans le cas de MPLS, un algorithme similaire est exécuté par le premier noeud i.e. le LER, les routeurs suivants, dit de cœur, se contentant d'effectuer de la commutation (plus simple et plus rapide).
Le rôle de ce type de routeur est donc de recevoir en entrée des paquets IP et de leur associer une route qui leur garantisse la classe de service qui doit leur être associée[4]. Chaque route est identifiée par une étiquette appelée *label*.

[4] La classe de service est préalablement négociée par l'utilisateur auprès de l'opérateur

De même, ce type de routeur assure la "délabellisation" des paquets MPLS qui quittent le réseau MPLS vers le réseau IP classique. En effet, si le routeur est un routeur d'accès, alors il est aussi l'extrémité du chemin pour tous les paquets labellisés en provenance du réseau MPLS. Il va supprimer le label du paquet et, toujours grâce à sa table de routage, le router classiquement vers le prochain nœud.

Le LER assigne un label à un paquet donné en fonction d'un certain nombre de critères. Par définition, tous les paquets qui respectent le même ensemble de critères se verront attribuer le même label. Un ensemble de critères de classifications est appelé une *FEC : Forwarding Equivalent Class* ou classe d'équivalence de transmission. La relation *FEC label* est injective i.e. à une FEC donnée il existe un et un seul label, par contre un label peut correspondre à plusieurs FEC.

Les critères contenus dans une FEC sont multiples. Leur nombre varie et ils vont d'une classification grossière (noeud de sortie du réseau MPLS, préfixe destination IP, adresse IP destination) à une classification très fine (adresse IP destination et source + port destination et source + protocole) permettant d'identifier un flot ou un micro-flot[5]. Une FEC est constituée de :

- adresse IP du noeud de sortie du réseau MPLS
- adresse IP destination ou préfixe adresse IP destination
- protocole
- adresse IP source + adresse IP destination
- adresse IP destination + numéro de Port destination
- ...
- adresse IP source et destination + numéro de Port source et destination + protocole

La qualité de service avec MPLS est assurée à deux niveaux :

1. le LSP est vu comme une connexion pour laquelle on garantit une qualité de service donnée (débit, délai, ...) par construction,
2. seuls les paquets IP appartenant à la FEC correspondante sont routés dans ce LSP.

Ainsi à la réception d'un paquet IP, si le paquet appartient à une FEC alors il est labellisé, sinon le routeur procède à la recherche du prochain noeud en routage classique. Les LSP sont créés selon deux modes possibles :

1. *Unsollicited* : chaque routeur demande des LSP à ses adjacents en fonction de sa table de routage. Une FEC est associée à un label par entrée dans la table de routage. Il est aussi possible de regrouper plusieurs entrées de la table de routage à un seul et même label, c'est ce que l'on nomme le *merging*.
2. *DOD (Downstream On Demand)* : dans ce cas, c'est l'opérateur ou un organe de commande qui sollicite un routeur d'accès (LER) pour établir un LSP pour une FEC donnée.

Le routeur de cœur a aussi un rôle dans le mécanisme de routage au sein même du réseau MPLS. Dans ce cas, il est appelé LCR, *Label Core Router*. A savoir, il ne reçoit que des paquets labellisés[6]. Si ce routeur est un *core router*[7] alors le paquet labellisé reçu est routé vers le prochain routeur du chemin. Le routeur suivant est déterminé grâce à la table des labels interne qui assure une correspondance entre label d'entrée et label de sortie : c'est de la commutation.

Si l'on résume, ce routeur doit être capable d'assurer les mécanismes de routage classiques sur un réseau IP, mais aussi celui, plus complexe, d'un réseau MPLS. Il assure donc une classification des paquets, afin de leur associer un label et pour différencier ceux qui emprunterons donc le réseau MPLS de ceux qui seront routés de manière "classique" vers un réseau non-MPLS.

[5] Un flot ou un micro-flot représente une connexion IP au sens connecté du terme de durée finie i.e. le rapatriement d'une page HTML

[6] Il reçoit aussi des paquets non labellisés, mais il n'appartiennent pas au réseau MPLS et ne sont pas concernés par le mécanismes MPLS

[7] Routeur de cœur interne au réseau MPLS

2.4 Définition du sujet de Tutorat

Les grandes lignes de mon tutorat était :

- Dans un premier temps, la réalisation d'une plateforme[8] de développement sous environnement Linux.
- Dans un deuxième temps, l'étude et l'implémentation des algorithmes de filtrage[9] de paquets existants en vue de leur amélioration et de leur adaptation pour la classification de paquets IP sous MPLS.

La première partie était quasi indispensable à la deuxième et était la suite du travail réalisé par un stagiaire ENSSAT juste avant mon arrivée (Un recouvrement de 15 jours des deux stages m'a permis de reprendre plus aisément la suite). Elle s'est avérée plus longue et plus complexe qu'il n'y paraissait au premier abord. L'ensemble de ce travail est décrit dans le chapitre suivant.

La deuxième partie, plus fondamentale, fût de ce fait plus courte que prévue mais menée à terme malgré tout. La partie filtrage est essentielle pour un LER. En effet, les premiers routeurs MPLS du commerce n'implémentent que la classification des paquets avec le grain le plus gros pour les FEC : préfixe IP destination voir adresse IP destination. Ceci correspond tout simplement à la partie de l'entête IP que les routeurs manipulent pour effectuer un routage classique bond par bond. Afin d'assurer de la qualité de service à un flot ou un micro-flot donné, il faut être en mesure de pouvoir réaliser une classification aussi fine que possible i.e. il faut que le LER puisse manipuler des FEC à la fois complexes et variées. Le travail demandé consiste précisément à chercher, améliorer et/ou adapter et réaliser un mécanisme de classification de paquet IP basé sur cette notion de FEC afin de pouvoir labellisé un flot ou un micro-flot donné. Enfin, il m'a été demandé d'étudier une commande de ce mécanisme pour que l'opérateur puisse programmer les FEC (les filtres).

La classification est compliquée due au fait que certains de ses paramètres (Numéros de port source UDP et TCP) sont calculés dynamiquement à l'établissement de la connexion (socket). La fragmentation IP qui supprime les informations des couches supérieures est un facteur supplémentaire de complication. La classification doit également tenir compte d'une certaine granularité de filtrage : d'un réseau distant jusqu'au *micro-flot* en passant par l'agrégation de flot[10]. L'ensemble de ce travail est décrit dans le "chapitre Filtrage".

2.5 Méthodologie de travail

Dans un premier temps, je me suis familiarisé avec les technologies utilisées dans le routage et dans les réseau haut débit :

- le fonctionnement du réseau ATM [6] ,
- le protocole MPLS et en particulier son implémentation sous Linux [8],
- les techniques de communication dans le noyau Linux [9], [10] et [11],
- l'implémentation commerciale de MPLS par Netplane [5] et [7],
- la signalisation accélérée sous ASIA [1] et [3].

Puis j'ai repris le travail précédemment réalisé et résumé dans un rapport [4] qui m'a servi de point de départ pour la suite.

Les seules contraintes du projet sont l'environnement Linux et l'utilisation d'une version commerciale de MPLS (la seule aussi complète à ce jour en notre possession). Une autre version non-commerciale de MPLS sous Linux est aussi en cours de développement à l'université Madison au Wisconsin. Elle nous a permis d'intégrer la version commerciale de MPLS au noyau Linux.

L'ensemble du développement se fait en langage C, langage avec lequel est écrit le noyau Linux et les deux versions de MPLS.

[8]Routeur d'accès MLPS
[9]Ceux existant pour le routage classique, diffServ, firewall, proxy, ...
[10]comprendre connexion

Chapitre 3

La plateforme de développement

3.1 Préambule

La réalisation et la mise au point d'une plateforme de développement est le premier point de ce travail. En effet la suite du projet va reposer sur cette architecture matérielle et logicielle. Elle combine deux choix logiciels : le noyau Linux et une solution commerciale pour la pile de signalisation MPLS.

Comme nous allons le voir par la suite, la démarche consiste dans un premier temps à adapter le noyau Linux, puis à installer et configurer la solution commerciale afin de comprendre le fonctionnement des deux parties indépendamment avant de déterminer les modifications à y apporter.

Le but final est donc de faire en sorte que les protocoles LDP, CR-LDP, RSVP-TE et le routage interagissent avec la partie du noyau correspondant à MPLS, à savoir celle qui s'occupe de la labellisation et de la transmission de paquets labellisés.

3.2 La partie noyau de MPLS

3.2.1 But à atteindre

La plateforme est constituée d'un PC sous Linux configuré pour fonctionner comme un routeur. C'est une fonction aisément réalisable sous Linux par une simple configuration du noyau. Mais dans notre cas, il a fallu aussi intégrer la contrainte de MPLS. Pour cela, la première étape a consisté à appliquer un patch[1] sur le noyau afin que celui-ci puisse prendre en compte MPLS[2]. Ainsi le noyau même du système d'exploitation s'en trouve modifié et intègre MPLS au niveau des tables de routage, de l'acheminement des paquets MPLS, de la labellisation, dé-labellisation, etc ...

3.2.2 Méthode de test

Ce patch est réalisé par Jim Leu, universitaire américain à l'origine du portage de MPLS sous Linux. *MPLS for Linux* est un projet ouvert qui adhère à la philosophie du logiciel libre, à savoir que quiconque peut accéder aux sources, les modifier[3] et participer à l'avancement du projet. Cette remarque a son importance car elle a pour conséquence une mise à jour et une évolution perpétuelle du code du fait principalement du nombres de contributeurs. Cela entraîne pour nous une nécessité de suivre en permanence les évolutions du code afin d'y adapter notre développement et soumettre nos modifications/amélioration.

[1]Fichier remplaçant (ou modifiant) tout ou partie d'un ou plusieurs fichiers
[2]Prise en compte effective après compilation du noyau ainsi modifié
[3]Généralement pour apporter de améliorations à l'existant

Une fois le noyau correctement patché et compilé, nous avons pu tester le fonctionnement des outils fournis par Jim Leu. Ce test va se faire au travers d'une démo proposée avec les utilitaires. Elle consiste à mettre en place un LSP entre deux machines Linux au travers d'une connexion Ethernet. Nous avons adapté le test pour que la connexion se fasse sur ATM.

3.2.3 Présentation des outils

L'outils principal de ce test est un programme qui permet d'interagir avec la partie routage MPLS du noyau Linux. L'étude de son fonctionnement a été un préambule a cette phase de test.

Le but de cet utilitaire est donc de pouvoir configurer les structures internes du noyau relatives à MPLS depuis l'espace utilisateur. Il s'appuie sur le principe des *sockets netlink* pour communiquer avec le noyau.

Les *netlink* sont utilisés pour transférer des informations entre les différents modules du noyau et les processus utilisateurs. Ils fournissent des liens bidirectionnels entre l'espace utilisateur et l'espace noyau. Du coté des processus utilisateurs, un *netlink* fournit une interface standard basée sur les sockets. Du coté des modules du noyau, c'est une simple API interne.

3.2.4 Le test proprement dit

La démarche est celle développée dans le tableau 3.1.

<table>
<tr><th colspan="2">Commandes pour chaque LSR</th></tr>
<tr><th>LER 1 hostname atm1</th><th>LER 2 hostname atm2</th></tr>
<tr><td><code>atmarp -d atm2</code></td><td><code>atmarp -d atm1</code></td></tr>
<tr><td><code>add route -host atm2 gw atm2</code></td><td><code>add route -host atm1 gw atm1</code></td></tr>
<tr><td colspan="2"><code>mplsadm -v -L atm0 :0/32</code></td></tr>
<tr><td><code>mplsadm -v -A -B -O atm :0/100 :atm0 -f atm2/32</code></td><td><code>mplasdm -v -A -I atm :0/100 :0/32</code></td></tr>
<tr><td><code>mplasdm -v -A -I atm :0/101 :0/32</code></td><td><code>mplsadm -v -A -B -O atm :0/101 :atm0 -f atm2/32</code></td></tr>
<tr><td colspan="2"><code>atmdiag -z</code></td></tr>
<tr><td><code>ping atm2</code></td><td><code>ping atm1</code></td></tr>
<tr><td colspan="2"><code>atmdiag</code></td></tr>
</table>

TAB. 3.1 – Test de mise en place d'un réseau MPLS

Il s'agit en fait d'établir une connexion ATM entre les deux machines sensées jouer le rôle de routeur. Ensuite on y fait transiter des données matérialisées par des messages ICMP produits par la commande *ping*.
La première commande détruit le *vc* par défaut[4] pour éviter toute communication intempestives en dehors du champ de notre expérience.
La seconde va nous permettre d'établir une route utilisant la connexion ATM entre les cartes réseaux de *LER 1* et *LER 2* d'adresses respectives `atm1` et `atm2`.
Ensuite on crée un LSP entre *LER 1* et *LER 2*, puis un autre entre *LER 2* et *LER 1*, pour enfin lancer la commande *ping* qui nous permet de vérifier si une connexion entre deux hôtes est effective.

[4]il est établie par les scripts de démarrage du noyau, il permet de vérifier que les piles IP et ATM sont correctement initialisées sur chaque machine

3.2.5 Test et résultats

Avant de tester, on réinitialise de part et d'autre les compteurs de cellules ATM à zéro[5] grâce à la commande `atmdiag -z`.
L'utilitaire *mplsadm* permet également d'activer le mode *debug* (déverminage) du patch MPLS. Dans ce cas, le noyau affiche sur la console et dans un fichier de log les messages relatifs à tous ce qui a trait au traitement des paquets IP en lien avec MPLS. C'est une option très utile pour vérifier et contrôler les évènements au niveau de la partie MPLS du noyau.

La commande UNIX `ping` nous indique que le nombre de messages ICMP envoyés est bien égal au nombre reçu. Ce nombre est le même affiché par le compteur de cellules ATM via la commande `atmdiag`. Sachant que la seule route empruntant l'ATM est celle utilisant MPLS, les résultats valident ce test d'un micro-réseau MPLS[6].

3.3 Le démon LDP de Netplane

3.3.1 But à atteindre

L'autre partie ce projet pour Linux comprend un volet relatif au routage proprement dit des paquets MPLS. Mais le démon chargé d'effectuer cette tâche ne prend pas encore en compte la variante du protocole de routage que nous comptons utiliser. En clair, le protocole LDP est supporté mais la notion de routage contraint (CR-LDP : Constraint Routing-Label Distribution Protocol) ne l'est pas encore.

Pour remédier à ce problème, le laboratoire a décider d'utiliser une pile de signalisation MPLS du commerce. Mais celle-ci est totalement indépendante de *MPLS for Linux* et fait en réalité partie d'une solution MPLS complète fournie par la société Netplane[7]. Elle n'interagit pas avec le noyau Linux, car se voulant portable, elle fonctionne comme une sur-couche logicielle. Cela veut dire par exemple qu'un changement dans le routage MPLS n'est pas répercuté dans les tables de routage du noyau Linux.

3.3.2 Méthode

Dans un premier temps, nous avons testé le démon LDP au travers une démonstration fournie par Netplane avant d'aller en étudier le code.

Ensuite il a fallu repérer les endroits à modifier pour permettre une communication avec le noyau et la table de routage. Le but final est de se servir de certaines parties du code de l'utilitaire de gestion de Jim Leu comme interface entre LDP et le noyau (i.e. via les netlink).

Une idée est aussi d'utiliser *Zebra* comme logiciel de routage global à notre plateforme. *Zebra* implémente tous les démons de routage classiques (OSPF, BGP4, RIP, etc), avec l'avantage d'être libre de droits.

Cela nous amène à présenter rapidement l'architecture de la solution logicielle de Netplane pour permettre de saisir l'environnement dans lequel nous évoluons. Elle se compose de deux parties : une couche intermédiaire et le produit mettant en œuvre les protocoles MPLS proprement dit.

La société Netplane propose un ensemble de produits multi-plateformes (Unix, Windows ou Solaris). Cette portabilité est assurée par une sous-couche dite d'abstraction qui leur permet de développer

[5]Dans le seul but de vérifier que le transfert se fait bien via les cartes ATM et donc notre réseau MPLS.
[6]Pour tester un autre cas, nous avons aussi réalisé un transfert de fichier via MPLS et FTP qui s'est avéré tout aussi concluant.
[7]Anciennement Harris & Jeffries

leurs applications sans se soucier de la machine cible. Cette couche baptisée LEAP pour *Layered Environment for Accelerated Portability*, contient un ensemble de fonctions assurant l'interface entre la plateforme d'exécution et les programmes : gestion de la mémoire, gestion des processus légers et du parallélisme et enfin encapsulation des fonctions réseau (cf. Figure 3.1).

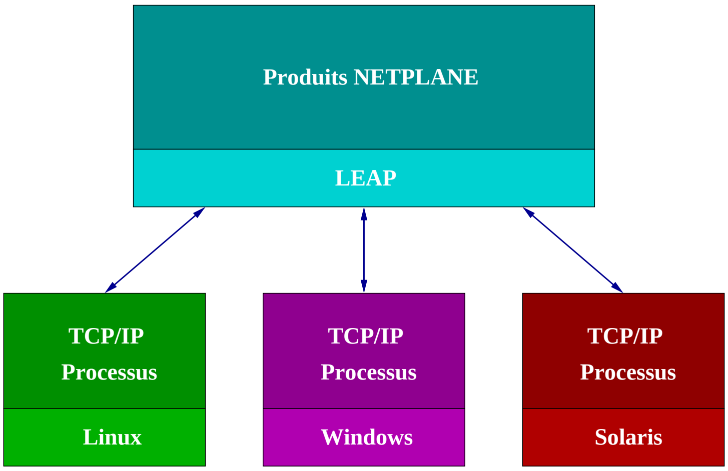

FIG. 3.1 – Architecture des produits Netplane

Mais chaque plateforme a ses caractéristiques propres concernant la mise en œuvre de la pile réseau, les communications avec le noyau ou encore la gestion des informations de routage. La tache qui nous incombe est donc, comme il a déjà été dit, d'établir les liens entre LEAP et les fonctions spécifiques au système, qui dans notre cas est Linux.

La seconde composante de la solution logicielle de Netplane qui met en oeuvre les protocoles MPLS s'appelle LTCS pour *Label Trafic Control System*. Elle comprend un ensemble de composantes permettant de programmer un démon LDP, CR-LDP ou RSVP-TE grâce à un ensemble de modules ayant chacun une tâche distincte :
- LMAN : ce module est chargé de gérer les labels, c'est-à-dire d'allouer et de désallouer les labels. Il est indépendant du protocole utilisé (LDP dans notre cas).
- LDP : ce module met en œuvre le protocole LDP permettant l'échange d'information sur les labels.
- LPE pour *Label Process Engine* : ce module gère les labels qui sont échangés par le module LDP. C'est ce module qui gère donc la table de labels.
- RSVP-TE : c'est un protocole alternatif à LDP qui permet d'une part l'échange d'information sur les labels à allouer, et d'autre part la réservation de bande passante donnée.
- CR-LDP : c'est le module qui implémente CR-LDP. Cependant comme le LER développé est assez simple concernant les fonctionnalités, ce module n'est pas utilisé. Il pourra être intégré dans une évolution du LER.

Tous ces modules viennent ensuite se raccrocher sur un méta-module appelé LMS qui assure la communication entre eux.

L'ensemble du produit Netplane que nous utilisons représente plusieurs dizaines de fichiers dont certains avoisinent les cinq mille lignes de code pour un total de quelques centaines de milliers. C'est dire que la phase de compréhension, d'assimilation et d'adaptation du code à nos besoins a représenté une part très importante de ce tutorat.

3.3.3 Adaptation de la démo de base

Au départ, notre étude s'est porté sur le fichier source de la démonstration fourni avec le produit. Celui ci nous a permis de mieux saisir le fonctionnement interne de LTCS ainsi que l'ordre d'appel des différentes fonctions menant à la création du démon LDP.

Il ressort de cette étude que deux parties de LTCS seront les points du programme où notre action sera la plus importante. Ces fichiers concernent la partie relative au traitement des labels[8] et le fichier contenant les sources de la démo.
La première est en effet l'endroit où toutes les modifications dues au traitement des labels devront être répercutées sur les structures internes du noyau Linux[9].
La seconde correspond à l'endroit où l'on initie le système et où l'on peut le configurer.

Le fichier source de la démo

Ce fichier nous donne l'exemple d'un programme qui puisse convenir aussi bien à un LER qu'à un LSR en jouant avec les options de la ligne de commande. L'option `n=x` permet de choisir sur quel type de routeur on se place. Par exemple la valeur `1` correspond à un LER. La différence va se faire sur le nombre de démons LDP à lancer. Un LER aura au moins deux interfaces, l'une étant connectée au réseau "classique", l'autre au réseau MPLS. Un LSR aura lui aussi au minimum deux interfaces, mais qui cette fois font toutes partie du réseau MPLS.
L'option `n=x` permet de lancer autant de démon LDP que le routeur possède d'interfaces MPLS.

Les modifications réalisées

La première modification concerne la récupération des informations de configuration du démon. Nous avons choisi d'utiliser un fichier de configuration externe et de récupérer ces informations à l'aide d'un "parser" plutôt que d'utiliser des options de la ligne de commande. Les données ainsi récupérées sont stockées en interne pour une utilisation ultérieure.
Le fichier de configuration s'organise comme le montre l'exemple 3.2.

```
# Fichier de configuration

ATM1 192.168.56.31
ATM2 192.168.56.32
# Fin fichier de configuration
```

TAB. 3.2 – Extrait du fichier de configuration de LDP

Ce fichier sera traité par un analyseur syntaxique qui compare les premiers termes de chaque ligne avec ceux qu'il connaît selon la méthode suivante.
Les lignes commençant par le caractère `#` sont considérées comme des commentaires et ignorées. De la même manière, on ne tient pas compte des lignes ne comportant que des espaces. Une ligne correcte comporte un mot clef, par exemple ici `ATM1`, et est suivi de sa valeur, toujours dans ce même cas : `192.168.56.31`. Le mot clef et sa valeur sont alors stockés dans un tableau pour une utilisation ultérieure.

Ensuite on met en place un masquage des interruptions destiné à empêcher certaines séquences de touches[10] d'influer sur le programme. A réception de certaines de ces interruptions, on va associer

[8] LPE pour *Label Processing Engine*
[9] Essentiellement sur celles relatives au routage
[10] La séquence CTRL+C par ex.

l'exécution d'une fonction qui nous permet de terminer correctement le programme[11]

La suite du lancement de LMS suit alors le schéma de la figure 3.2. Différents services vont être connectés au LMS dont le LPE. Ensuite, l'instance LMS sera identifiée par la variable `LsrId` qui a en fait comme valeur l'adresse IP de l'interface physique.

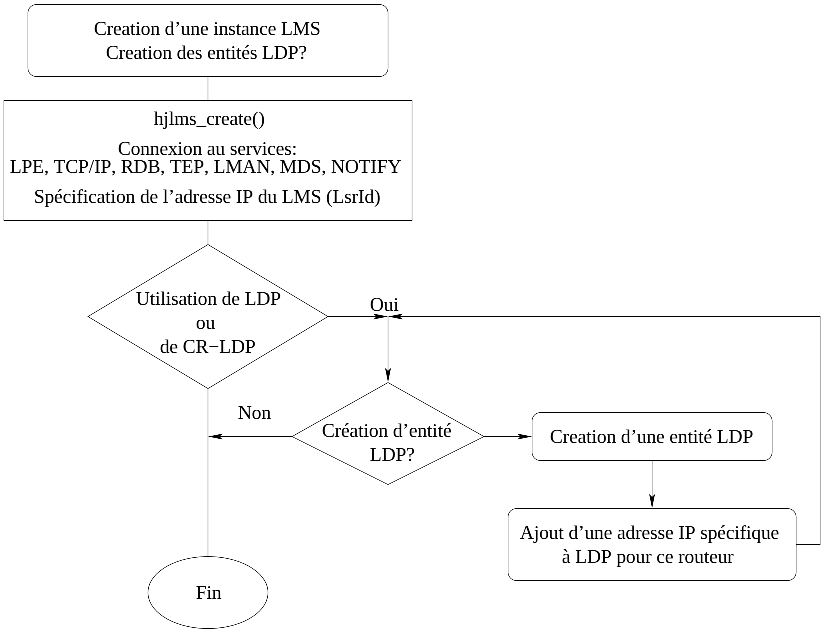

FIG. 3.2 – Création d'une instance LMS

Au cours de la mise en place du LMS, au moins une instance[12] LDP est créée et associée à ce LMS. Cette mise en place passe par un certain nombre d'opérations qui sont résumées à la figure 3.3. Dans le cas qui nous concerne, l'entité LDP est du type ATM. En effet, le protocole support de notre réseau MPLS est l'ATM[13].
Une des tâches principales lors de la création de l'entité LDP, est l'affectation d'un intervalle de label. Le démon ne pourra donc affecter qu'un label appartenant à cet ensemble de valeurs. L'ajout de cet intervalle est décrit dans le schéma de la figure 3.4.

Résultat

Cette première exploration du code source de Netplane m'a permis de mieux saisir le fonctionnement et l'organisation interne des fichiers et des fonctions. Le seul fichier de démo comporte un plus de cinq mille lignes de code qu'il a fallu analyser et décortiquer pour en saisir les fonctionnalités.

La récupération des informations à partir du fichier de configuration et leur prise en compte au niveau du programme sont réalisées et fonctionnent. Le masquage des interruptions pose plus de problèmes, car le programme lance plusieurs processus indépendant en parallèle dans lesquels nos modifications ne sont pas répercutées.
Une autre solution a été trouvée, comme nous le verrons par la suite, avec la mise en place d'un serveur

[11]Ceci est rendu nécessaire, car l'implémentation de NetPlane utilise les *threads*, de la mémoire partagée et des *mailbox*, donc des ressources systèmes qu'ils faut restituer correctement.
[12]En fait un démon associé
[13]Le programme réalisé fonctionne également avec les interfaces Ethernet, c'est le fichier de configuration qui va déterminer le type d'interfaces utilisées.

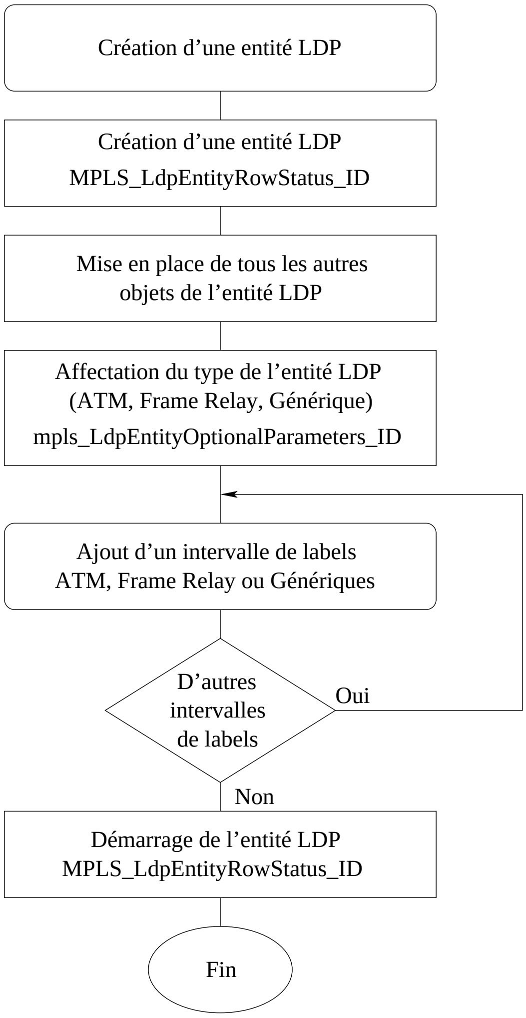

FIG. 3.3 – Création d'une instance LDP

de commande.

Mais la partie où le plus grand nombre de modifications ont été effectuées se situe au niveau de la partie *LPE*. Celle-ci est une composante de LTCS, chargée de la gestion des labels, qui est mise en place lors de la création d'une instance LMS. C'est donc cette partie qui va maintenant être développée.

3.3.4 La partie LPE

Le *Label Process Engine* se charge de gérer les labels au travers d'un arbre de Patricia. C'est donc un point névralgique dans notre programme. En effet, chaque action du LPE doit aussi se répercuter sur les structures de gestion des labels au niveau du noyau Linux. Mais les structures mises en place par la solution commerciale omettent certaines informations qui sont nécessaires pour faire le lien avec le noyau.

La première étape a donc été de mettre en place une structure indépendante de gestion des labels ainsi que les mécanismes de gestion qui lui sont associés.

Dans un second temps, il a fallu répertorier l'ensemble des fonctions composant les sources[14] du LPE qui agissaient sur les labels (création, modification ou encore destruction de labels). En effet, chacune de ces fonctions devait être modifiée pour permettre une répercution des actions sur les nouvelles

[14]plusieurs dizaines de milliers de lignes de codes

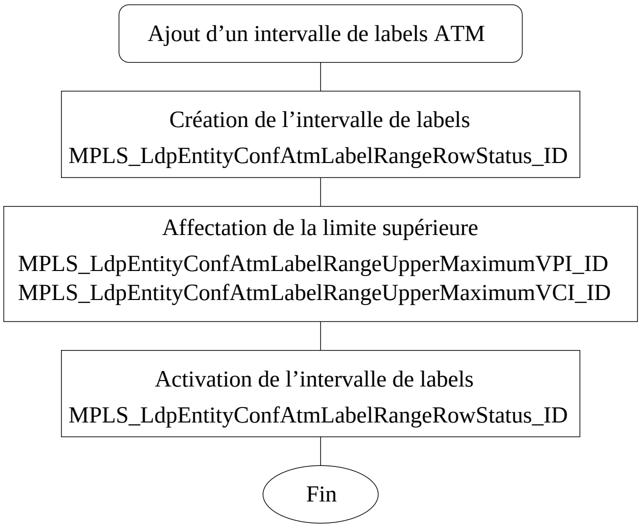

FIG. 3.4 – Ajout d'un intervalle de labels

structures mais aussi sur celles du noyau.

Nouvelle structure

La structure qui est mise en place est organisée autour d'un tableau de LSP indexé sur l'identifiant de LSP comme le montre la figure 3.5. Le champ important est `info` qui contient les identifiants des connexions[15] nécessaires pour toute communication avec le noyau. Ce sont ces informations qui nous manquaient dans la version de base du LPE.

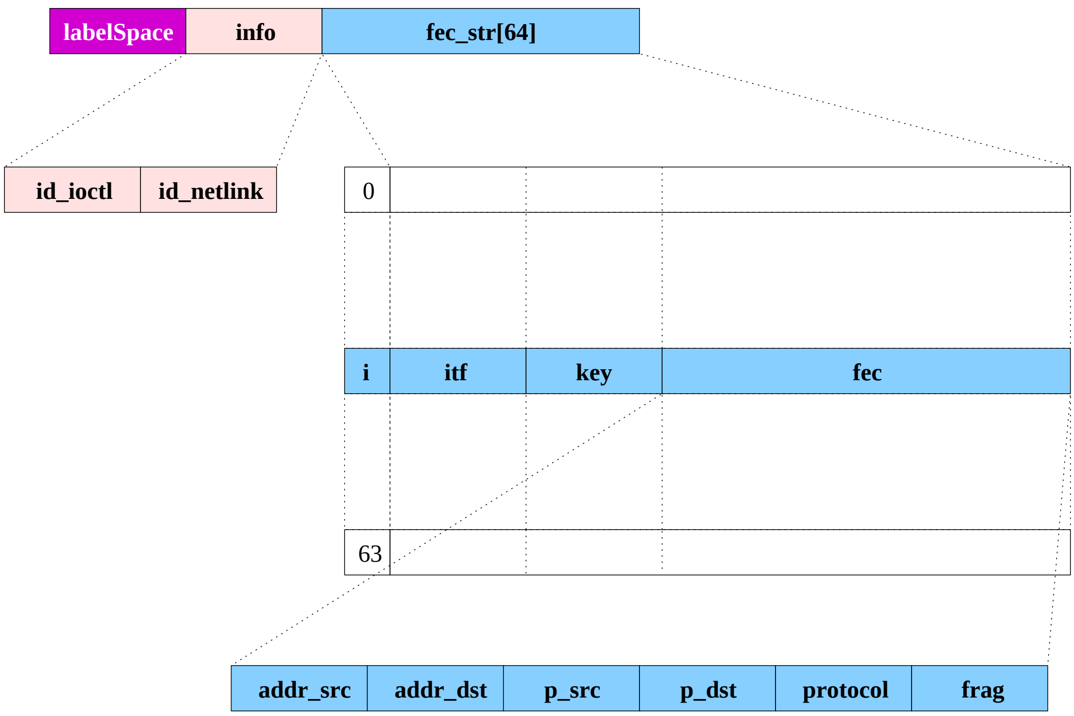

FIG. 3.5 – Nouvelle structure de gestion des labels

[15] ioctl et netlink

Pour chaque LSP, on associe l'interface à laquelle il se rattache, une clé qui permet de l'identifier dans les structures du noyau et une FEC. La clé est nécessaire parce qu'elle permet d'identifier de manière unique chaque label au niveau du noyau Linux. Enfin chaque FEC est composée de six champs (qui ne sont pas toujours tous initialisés) correspondant à :
- adresse IP source,
- adresse IP destination,
- port source,
- port destination,
- protocole,
- état de fragmentation (pour prendre en compte la fragmentation ou non).

Transmission de la structure

Au cours de l'étude du code de création de l'instance LPE, il s'est avéré que certaines informations nécessaires à l'interaction avec le noyau n'étaient pas disponibles. En effet, la création et l'activation de l'instance LPE sont faites au cours de la mise en place du LMS. Il s'agit en fait du simple remplissage d'une structure `lpe_bindery` dont certains champs sont des pointeurs vers des fonctions de création. A l'appel de chacune de ces fonctions, seuls les autres champs de la structure sont disponibles.
Il faut que notre structure de gestion de labels puisse être "visible" par toutes les fonctions agissant sur les labels[16] pour pouvoir la mettre à jour. Deux solutions peuvent permettre de répondre à cette contrainte : définir notre structure comme globale au programme ou modifier la structure `lpe_bindery` pour y insérer les informations qui nous manquent.
La première semble la plus simple, mais aussi la moins "esthétique". Je lui ai donc préféré la seconde qui répond plus précisément a nos besoins. J'ai donc rajouté un champ dans `lpe_bindery` qui est un simple pointeur sur notre structure. Ainsi elle est accessible et modifiable par toutes les fonctions de gestion des labels[17].

Intégration

Le champ `info` de notre structure permet l'accès au noyau via les netlinks. La deuxième étape consiste donc à modifier les fonctions de gestion des labels, pour répercuter leurs actions sur le noyau.

Ainsi, lors de la configuration des interfaces réseau par LDP, il convient de rajouter une fonctionnalité permettant de lui associer un pool de label MPLS au niveau du noyau. Pour cela, il faut envoyer au noyau un message particulier (`SIOCSLABELSPACEMPLS`) via un ioctl.
De même, lorsque l'interface doit être "dé-configurée", le pool de label qui lui est associé sera alors détruit selon le même principe.

Mais les fonctions de gestion des interfaces ne sont bien sûr pas les seules à avoir été modifiées. En fait, toutes fonctions agissant sur la gestion des labels s'est vu ajouter du code. Les nouvelles lignes respectent toutes plus ou moins le schéma :

1. Répercution des actions sur les labels dans notre structure de gestion,
2. Transmission des modifications, pour mise à jour des structures MPLS internes au noyau (Fait automatiquement par le noyau à réception des informations.).

La nouvelle version du LPE

Les modifications apportées à la version originale de LPE garantissent que la gestion des labels sera répercutée sur notre nouvelle structure et sur celles du noyau. Un premier lien entre la version "libre" et le produit commercial est réalisé.

[16]et en particulier par celles appelées au travers de `lpe_bindery`
[17]En effet `lpe_bindery` est le paramètre de chacune des fonctions de création

Cette phase de développement a duré plusieurs mois et à subi plusieurs modifications au cours de sa réalisation. En effet, durant les premiers mois de mon tutorat, la société Netplane a réalisé plusieurs versions successives de LTCS. L'une d'elle a vu une refonte profonde de la partie gestion des labels (LPE). Ainsi, une partie qui nous manquait et que j'étais en train de développer a été intégrée. Il a donc fallu que je réadapte mon code[18] aux nouvelles contraintes imposées par cette version.
De même, la taille du fichier regroupant les fonctions relatives à LPE s'est vue doublée pour atteindre un petit peu moins de onze mille lignes de code. Après avoir étudié plus attentivement le fichier, il s'est avéré que pour des raisons de compatibilité ascendante et de portabilité l'ancien code avait été conservé en parallèle avec le nouveau. Le choix de la version se fait à la compilation en affectant ou non une option du `Makefile`[19]. J'ai donc pu réduire mon champ d'étude à approximativement cinq mille lignes.

L'adaptation du LPE a donc demandé beaucoup de temps, principalement à cause des versions successives du code (trois au total). Le passage en paramètre de notre structure a nécessité une étude approfondie du code et des mécanismes qui géraient la création des différentes instances du LMS.
Un autre point noir a été la documentation fournie par Netplane [7] [5]. Volumineuse (mille pages environ au total), elle n'en reste pas moins plus un manuel de référence qu'une aide à la compréhension du fonctionnement de l'outil. A chaque fois que j'avais besoin d'utiliser une de leur structure ou fonction, j'avais en détail ce qu'elle faisait, les arguments qu'elle prenait, mais en aucun cas la méthode pour la mettre en œuvre. Et malgré la présence d'une assistance technique, il n'était pas toujours facile de trouver l'information qui nous manquait. Bien souvent la solution a été d'utiliser la méthode empirique qui consiste à écrire le code, tester et aviser en fonction du résultat.

3.3.5 Intégration de *Zebra*

Le contexte

Comme nous l'avons déjà vu, MPLS a besoin d'utiliser le routage classique bond par bond afin de déterminer le prochain nœud à qui adresser une demande de label en cours, découvrir ses voisins, établir une session LDP, CR-LDP et/ou RSVP-TE, ... Ces derniers sont transmis normalement à l'aide du routage classique et ne sont en aucun cas labellisés. La signalisation MPLS peut également utiliser la diffusion *multicast* pour déterminer ses adjacents (voisin) avec qui elle va établir des connexions.
Dans la version de base fournie par Netplane, la partie LDP comporte trois modules d'interactions avec le routage :
- une gestion interne du routage avec des tables de routages propres au processus LDP,
- une communication par *netlink* avec le noyau Linux afin d'interroger ses tables de routages,
- et une communication par *socket* avec le processus de routage *Zebra*.

L'utilisation de *Zebra* est intéressante à plusieurs point de vues :
- nous disposons avec *Zebra* de plusieurs protocoles de routage : *OSPF, BGP4* pour ne citer que les plus important,
- *Zebra* centralise l'ensemble des informations de routage de tous les protocoles et mets à jour en conséquence les tables de routages du noyau,
- il peut informer tous les processus qui le demandent de toutes les modifications de routages en temps réel,
- il offre un accès direct aux informations complètes de routage (en particulier la topologie complète) sans passer par le noyau.

Zebra en quelques mots

Zebra est un logiciel du domaine public sous licence GNU/GPL qui permet de gérer un ensemble de protocoles de routage basés sur TCP/IP. La particularité de ce logiciel est d'être basé sur la modularité (à chaque protocole de routage correspond un module), ce qui lui confère une grande souplesse

[18]En autre, la structure de gestion des labels.
[19]Makefile : fichier utilisé par la commande `make` pour compiler un ensemble de fichier de manière ordonnée.

d'utilisation et la possibilité d'être *multi-thread*. *Zebra* centralise les information en provenance et à destination des différents modules à la manière d'un micro-noyau. A l'aide de toutes ces informations il peut mettre à jour le routage effectif présent dans le noyau Linux. Ce module central offre, de plus, une connexion vers un programme utilisateur à l'aide d'une simple *socket*. Dans ce cas, le programme utilisateur est vu comme un module supplémentaire.

Zebra et LTCS

L'idée est donc d'opérer des modifications dans le code de RDB[20] pour le faire interagir avec *Zebra* selon le modèle présenté à la figure 3.6. Les informations de routage seront récupérées de *Zebra* par LDP via RDB.

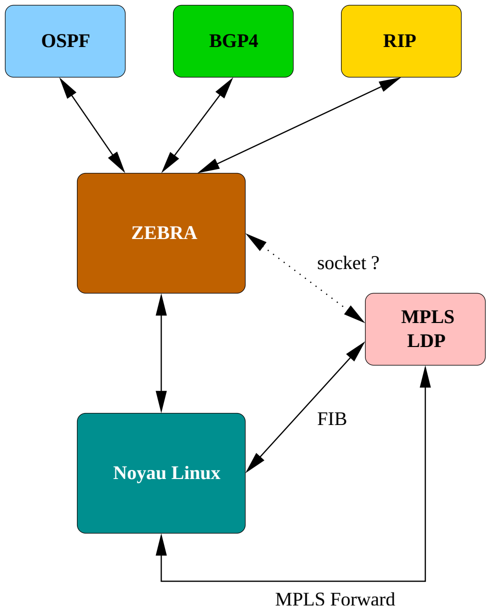

FIG. 3.6 – Interactions de Zebra

Le première chose a consisté a comprendre les mécanismes qui régissent l'initialisation et la création d'instances (RDB dans le cas qui nous concerne). Comme nous l'avons vu dans la section 3.3.4, la phase d'initialisation se fait par le remplissage des champs d'une structure commune à toutes les instances. Certains de ces champs sont en fait des pointeurs sur des fonctions chargées de lancer les instances.

Notre problème a donc été de remplacer la fonction initiale `rdb_layer_handle` utilisant la gestion classique via les *netlink* par celle utilisant *Zebra*. Après discussion avec l'assistance technique de Netplane, un des développeurs nous a fournit un package qu'il avait développé et permettant selon lui une intégration simplifié de *Zebra* dans RDB et le tout sous Linux.

L'enchantement a été de courte durée, sa version était inutilisable car développée sous un environnement spécifique à Netplane auquel il n'était pas possible d'avoir accès et lui n'avait pas assez de temps pour nous expliquer ce qu'il avait fait[21]. De plus il s'était basé sur *libnet*, une collection de routines assurant une interface de gestion réseau avec le noyau. Notre idée étant d'alléger cette gestion, une couche supplémentaire n'était pas le bon choix.

Modifier la phase d'initialisation nous a donc semblé une meilleure piste. Après quelques tâtonnements, une première version du programme intégrant *Zebra* a vu le jour. Mais un autre problème est apparu : une incompatibilité entre le module *Zebra* de Netplane et certaines versions de *Zebra*. Après s'être renseigné auprès de l'assistance technique pour des bugs que nous avions détectés, il nous a été

[20]RDB : **R**outing **D**ata**B**ase, démon Netplane chargé de gérer les bases de données de routage.
[21]Et surtout comment il avait procédé.

répondu que le module était juste fourni a titre d'exemple et sans aucune garantie.

Après plusieurs essais nous avons réussi à trouver la bonne version de *Zebra*, mais lors des tests[22] nos deux LER n'arrivaient pas à dialoguer via *Zebra*. La socket était bien initialisée, mais l'échange de messages entre *RDB* et *Zebra* ne fonctionnait pas. Apparemment le format des messages avait changé. Comme beaucoup de logiciels libres *Zebra* ne comporte peu ou pas de documentation sur son implémentation et encore moins sur une éventuelle interface de programmation. La seule documentation disponible est le code source. De même, le projet évolue assez vite et n'a pas encore sortie une version dite "stable". De ce fait, entre la version utilisable par LDP et la dernière version officielle sur le site de *Zebra*, il y avait eu une refonte complète des structures internes y compris des messages échangés entre les modules.

Le temps passé sur cette partie et le fait qu'elle n'était pas indispensable au fonctionnement de la plateforme mais juste un confort supplémentaire, nous a convaincu de la laisser de côté pour passer à la partie filtrage plus importante. Nous nous sommes repliés sur une utilisation du routage via la connexion avec le noyau par *netlink*. Les tables de routage du noyau étant quand à elles mises à jour par *Zebra*. La récupération des informations est alors indirecte.

3.4 Bilan

La réalisation de la plateforme de développement a monopolisé plusieurs mois (d'octobre à mi-avril pour être précis). C'est beaucoup plus que ce qui avait été initialement prévu. Les raisons de ce débordement sont multiples :

- les versions successives de la solution commerciale ont bien souvent remis en cause tout ou partie du travail déjà accompli, au point parfois de devoir reprendre l'étude de certaines parties à zéro,
- le code pour utiliser *Zebra* s'est avéré difficile à intégrer et aucune aide ne pouvait nous être fournie sur ce sujet par Netplane, et il n'y a pas de documentation de *Zebra* si ce n'est le code source,
- la documentation fournie par Netplane ressemblait bien plus à un manuel de référence plus qu'à un manuel du développeur. Bien souvent la compréhension du fonctionnement de certaines fonctions, par exemple, s'est faite par le décorticage du code existant et par de multiples essais successifs avant d'arriver au résultat escompté[23].

Mais ces phases de blocage on été utilisés pour avancer dans l'étude du filtrage, pour ainsi travailler sur l'aval du projet.

Au final, la plateforme fonctionne avec un lien indirect vers *Zebra*[24], mais sert de support à la partie filtrage que nous allons aborder maintenant.

[22] les mêmes que ceux décrits à la section 3.2.4

[23] Sans compter le nombre impressionnant de fausses pistes...

[24] emphZebra tourne dans son coin et met à jour les table de routage du noyau. emphLTCS fonctionne indépendamment de et récupère les informations de routage en lisant la table de routage du noyau

Chapitre 4

Le filtrage de paquets

La précédente réalisation de la plateforme avait pour but de nous fournir un support de développement. Ainsi nous disposons d'un outil pouvant fonctionner aussi bien comme un routeur d'accès à un réseau MPLS que comme un routeur de cœur. L'entière gestion des labels et des mécanismes de routage est assuré. Mais il reste un point en amont du routage : pouvoir faire l'association entre un paquet arrivant et la qualité de service qui lui est associée, entre un paquet et une *FEC*.

La résolution de ce dernier point nécessite de récupérer un certain nombre d'informations sur les paquets arrivant (en particulier les entêtes IP et TCP ou UDP) puis de les comparer avec d'autres que l'on aura nous même fixées pour déterminer le traitement à faire sur chaque paquet.

Dans un premier temps, il s'agira d'établir une *FEC*[1] à partir des informations récupérées du paquet. Ensuite il faudra faire l'association entre la *FEC* et le label afin que le paquet soit aiguillé vers le bon LSP.

L'appartenance d'un paquet à une *FEC* donnée va se faire grâce à une partie de filtrage et classification.

4.1 Le voyage d'un paquet MPLS au coeur du noyau Linux

Dans cette première étape de la partie filtrage, la phase de compréhension du fonctionnement de la pile réseau du noyau Linux a été la plus importante. Il fallait trouver la méthode la plus appropriée pour déterminer l'appartenance à une FEC pour ensuite assigner le label correspondant au paquet. Mais avant cela, il était nécessaire de connaître les caractéristiques du cheminement d'un paquet et les différences introduites dans le noyau par le patch MPLS. Non seulement pour comprendre les mécanismes de traitement des paquets dans la couche IP du noyau, mais surtout il nous fallait déterminer l'endroit le plus judicieux pour placer notre mécanisme de classification.

4.1.1 Méthode

Je me suis basé sur un document décrivant *"le voyage d'un paquet dans la couche réseau du noyau Linux version 2.4"* [14]. Ainsi j'ai pu tracer dans le code du noyau le trajet effectif d'un paquet IP. Mais ce document ne concerne que les paquets IP classiques et ne prend pas en compte MPLS. Afin de connaître les différences entre le traitement classique et le traitement qu'entraîne MPLS, j'ai tracé le trajet effectif d'un paquet dans les sources du noyau modifié par le patch MPLS[2].

C'est au cours de cette étude que j'ai corrigé et augmenté avec les informations concernant MPLS le document initial. Par la même occasion je l'ai traduit en français et soumis à l'auteur pour intégration.

[1] Forwarding Equivalent Class
[2] Soit une exploration dans un petit peu plus de 150000 lignes de code

Le résultat de ce travail est présenté en annexe A.

4.1.2 Résultats

Cette étude m'a permis de mieux comprendre les mécanismes qui régissent la pile réseau du noyau Linux. Plusieurs points importants ressortent :
- il s'avère que le parcours d'un paquet varie selon qu'il s'agisse d'un paquet classique ou d'un paquet MPLS,
- des notions de clé et de table de hashage sont introduites par MPLS, en plus de celles utilisées pour le routage classique bond par bond,
- chaque paquet entrant est filtré ; le traitement qu'il doit subir[3] est subordonné au résultat de ce filtrage,
- le mécanisme de filtrage dans le noyau utilise le système *netfilter* qui sera détaillé à la section 4.3.

Les différences de trajet vont être décrites maintenant tandis que les deux autres points seront développés dans les sections suivantes.

4.1.3 Différences dans le traitement des paquets IP

Les principales différences résident dans la manière dont est transmis l'information à l'entité physique. Dans le cas d'un routage classique, le paquet transite à travers le mécanisme de filtrage jusqu'à être transmis à la couche matérielle. Dans le cas de MPLS, l'information peut être sous forme labellisée[4]. Pour déterminer la suite du traitement, il faut savoir s'il existe une correspondance directe entre le label et une interface de sortie.

Le traitement des paquets dans le noyau est associé à la notion de *skbuff* pour socket buffer. C'est en réalité une structure, qui outre le paquet lui même, contient toutes les informations nécessaires au traitement du paquet depuis les entêtes jusqu'à la description de l'interface physique utilisée. Le *skbuff* contient cependant dans sa structure plusieurs pointeurs sur des fonctions. Elles s'apparentent aux méthodes de traitement associées à une donnée en programmation objet. Il a été, entre-autre, défini deux méthodes pour traiter le paquet en entrée et en sortie. La différence entre le traitement classique et MPLS réside en grande partie dans la modification de la méthode de sortie. Dans le cas de MPLS, si le paquet appartient à une FEC donnée, la méthode de sortie est positionnée à `mpls_output` au lieu de `ip_finish_output` et la clé correspondante à la FEC est positionnée dans la structure *skbuff*.

De même, si le paquet est le premier d'un flot destiné au réseau MPLS, on doit lui associer un label correspondant au LSP associé à la FEC. Toutes les informations nécessaires au routage MPLS sont contenues et organisées dans une table de hashage et indexées selon une clé, comme nous allons le voir dans la section 4.2.

D'autre part la détermination de la FEC ne se fait pas sur l'ensemble des informations recommandées par la norme IETF. Il est donc nécessaire d'améliorer encore le filtrage pour pouvoir déterminer avec plus de précisions les FEC. L'étude des documents rédigés par Rusty RUSSEL [13], [15] ainsi que le code du noyau ont servi de base pour la suite.

4.2 Clé et table de hashage

Une nouvelle implémentation de la pile IP, appelée *IProute2*, a été officiellement intégrée au noyau Linux depuis la version 2.4. Une des modifications apportées réside dans l'utilisation d'une table de "hash" pour accélérer le routage des paquets. Les informations nécessaires au routage sont contenues et organisées dans cette table de hashage et indexées selon une clé. La clé de hashage est calculée

[3]destruction, routage classique, routage MPLS, ...

[4]Encapsulation du paquet IP dans un paquet qui a comme entête un label.

sur les champs : adresse IP destination et source, le numéro de l'interface physique d'entrée (pour un paquet arrivant) ou de sortie (pour un paquet partant) et le champ TOS[5].

Lorsqu'un paquet arrive, on va rechercher dans la table si une clé correspond à celle calculée à partir des données contenues dans le *skbuff*. Deux cas peuvent se présenter :

– aucune correspondance ne peut être faite,
– la clé calculée à partir du *skbuff* correspond à une entrée dans la table.

Dans le cas de la première hypothèse, cela veut dire que le paquet est nouveau et ne fait partie d'aucun flot ou micro-flot de paquets déja existant. Dans ce cas, on va créer une nouvelle entrée dans la table de hashage qui correspondra à ce flot ou micro-flot de paquets. Tout ce travail est réalisé lors de l'appel aux fonctions `ip_route_input_slow()` et `rt_set_nexthop()` qui détermineront la suite du traitement à réaliser pour ce paquet. Les informations sur ce traitement seront elles aussi stockées dans la structure `rt_entry` indexée par la clé de "hash".

Dans le second cas, si une entrée existe, cela veut dire que le paquet fait partie d'un flot ou d'un micro-flot de paquets déja connu. Il n'y a pas besoin de créer une nouvelle entrée dans la table, il suffit juste de le traiter comme l'a été le premier paquet de ce flot. Les informations de traitement sont récupérées dans la structure `rt_entry` indexée par la clé de "hash". On s'affranchit donc de passer dans les fonctions `ip_route_input_slow()` et `rt_set_nexthop()` qui sont lentes (puisqu'elles mettent en œuvre l'algorithme dit du "plus long préfixe"). Dans les deux cas, le traitement se poursuit dans la fonction contenue dans `skb->dst->input`, i.e. `ip_forward()`.

Le mécanisme MPLS du noyau Linux est également basé sur l'utilisation d'une table de hashage appelé *RADIX_Tree*. Elle est initialisée en deux temps. Premiérement lorsque un LSP (plus exactement un label) de sortie est créé. La clé est formée du label, du numéro de l'interface de sortie et du type de protocole (ETHERNET, ATM, FRAME RELAY). Dans un deuxième temps, lorsque l'on associe une FEC à ce label, la clé du *RADIX_Tree* est rattachée à une entrée de la table de routage. Ceci limite la FEC au champ adresse IP destination. De plus, il faut que cette FEC existe dans la table de routage du noyau. D'où la nécessité de créer un routage explicite pour une machine donnée dans le cas de l'utilisation de l'adresse IP destination complète.

Lorsque le premier paquet que l'on veut labelliser va arriver, il va engendrer un nouveau calcul de clé dans la table de routage. C'est alors dans la fonction `rt_set_nexthop()` que la clé du *RADIX_Tree* est récupérée grâce à l'association entre la table de routage classique et la clé MPLS. Les paquets suivants utilisent les mêmes informations, donc héritent du même label.

Si l'on résume, le premier paquet d'un flot va créer une nouvelle entrée dans la table et ainsi tous les autres paquets de ce flot auront déja une entrée et en utiliseront les informations pour poursuivre leur chemin dans le noyau[6]. Cependant, à l'instar des implémentations commerciales, le mécanisme proposé par *MPLS for Linux*, n'utilise que la classification des paquets par FEC selon le grain le plus fort.

4.3 Le principe de *netfilter*

4.3.1 Le contexte

Lors de l'étude du trajet d'un paquet dans la pile réseau du noyau, il est apparu qu'un mécanisme appelé *netfilter* gère l'ensemble du filtrage.

4.3.2 Le fonctionnement

Tout d'abord, *netfilter* est un canevas pour modifier les paquets réseau en dehors de l'interface de sockets Berkeley. D'abord, chaque protocole définit des "hooks" (accroches) qui sont des points bien déterminés dans le trajet du paquet dans la pile de protocole. IPv4 en définit cinq :

– PRE_ROUTING : avant exécution de l'algorithme de routage,

[5] *Type Of Service* : La classe de service utilisée par *DiffServ*

[6] Sur le modèle du premier paquet.

– LOCAL_IN : traitement des paquets destinés à ce routeur,
– FORWARD : dans le cas d'un paquet retransmis,
– POST_ROUTING : après exécution de l'algorithme de routage,
– LOCAL_OUT : traitement des paquets émis par ce routeur.

A chacun de ces points, le protocole va appeler le canevas de Netfilter et lui passer le paquet et le numéro de "hook" (cf. le schéma de la figure 4.1). Mais *netfilter* c'est aussi un puissant mécanisme de filtrage. En effet, seul les paquets qui vont passer au travers d'un ensemble de filtres se verront assigner un traitement spécifique.

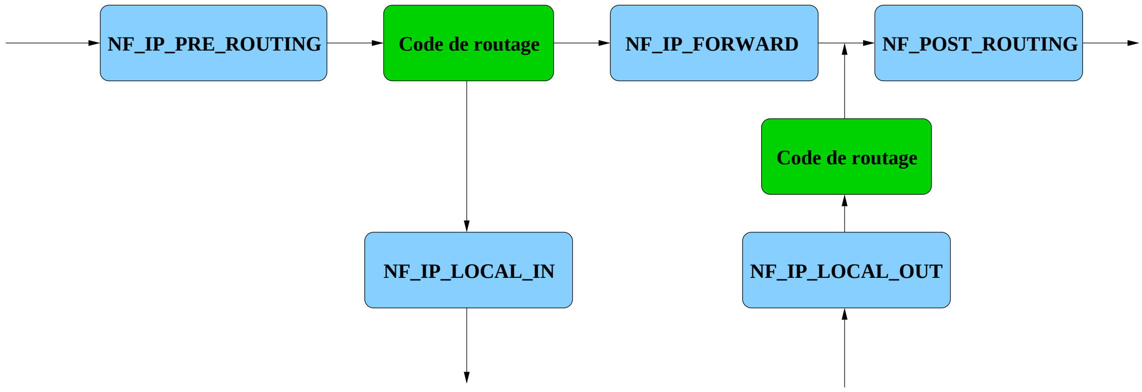

FIG. 4.1 – Schéma de traversée idéal de la pile avec netfilter

Ces "hooks" sont appelés si et seulement si ils ont été initialisés. Au boot du système, l'ensemble des "hooks" est initialisé à une valeur par défaut qui laisse passer tous les paquets. Un programme utilisateur, *iptables*, permet d'enrichir les tables de filtrages des "hooks" avec des valeurs particulières. Ce mécanisme est essentiellement utilisé pour effectuer des fonctions de *firewall*, *proxy*, *NAT* (translation d'adresses IP), ...

Donc quand un paquet passe au travers du canevas *netfilter*, ce dernier va vérifier si des règles ont été initialisées pour ce protocole et ce "hook". Si c'est le cas, ils vont tous avoir une chance d'examiner (et aussi modifier) le paquet dans l'ordre dans lequel les règles ont été instaurées. Le résultat d'un "hook" est :
– de se débarrasser du paquet (`NF_DROP`),
– de l'autoriser à passer (`NF_ACCEPT`),
– de dire à *netfilter* d'oublier le paquet (`NF_STOLEN`),
– de dire à *netfilter* de mettre le paquet en file d'attente dans l'espace utilisateur (`NF_QUEUE`).

Les paquets qui ont été mis en file d'attente sont récupérés pour être envoyés dans l'espace utilisateur de manière asynchrone.

La programmation de *netfilter* est également modulaire. Chaque action est décrite dans un module. Il se charge dans le noyau soit classiquement à l'aide des commandes *modprobe* ou *initmodule* soit à l'aide de la commande *iptables*. Une fois le module chargé, il peut s'enregistrer pour écouter à chacun des "hooks". Un module qui enregistre une fonction doit spécifier la priorité de cette fonction à l'intérieur du "hook". De telle sorte que quand ce "hook" est appelé par *netfilter*, le code réseau central appellera les fonctions enregistrées dans ce "hook" dans l'ordre des priorités.

4.3.3 L'utilitaire *Iptables*

Iptables a été construit au dessus du canevas *netfilter*. Cet utilitaire permet d'insérer et de supprimer des règles dans la table de filtrage des paquets du noyau. Il informe donc le noyau des filtres à appliquer sur les paquets et quels modules il doit exécuter sur ces paquets.

Iptables fournit simplement un tableau de règles en mémoire et des informations telles : "A partir de tel hook, ce paquet va être traité par tel module". Après qu'une table aie été enregistrée, elle est accessible[7] depuis l'espace utilisateur via des *sockets*.

Les modules du noyau peuvent demander à enregistrer une nouvelle table et demander à ce qu'un paquet traverse une table donnée. Cette méthode de sélection de paquet est utilisée pour filtrer les paquets (via la table *filter*), pour NAT (la table *nat*) et pour modifier des paquets avant routage (la table *mangle*).

4.4 TC et *iproute2*

Il existe également un autre mécanisme de traitement des paquets IP dans le noyau Linux : *TC* (Trafic Conditionner). Il fait partie intégrante de la nouvelle implémentation de la partie IP du noyau Linux depuis la version 2.4 et baptisé *iproute2* (cf. section 4.2). Ce mécanisme gère de multiples files d'attente associées à une interface donnée. De même que *netfilter*, il implémente un puissant mécanisme de filtrage pour déterminer quel paquet doit d'être placé dans quelle file d'attente. Mais il est beaucoup moins aisé à manipuler. La description et la syntaxe des règles de filtrage sont très complexes. De plus, ce filtrage agit après traitement de la couche IP, donc également après le traitement MPLS. Cependant, il sera, par la suite, intérressant d'effectuer un lien entre *netfilter*, MPLS et *TC* pour que le label d'un paquet serve de clé à *TC* pour le placer dans la bonne file d'attente. Ce dernier point assurera définitivement le respect de la qualité de service demandée.

4.5 Le module *MPLS-Netfilter*

4.5.1 L'idée

Comme nous l'avons vu précédemment, *netfilter* fournit un puissant mécanisme de filtrage qui répond tout à fait à notre attente. L'idée est donc d'écrire un nouveau module de traitement satisfaisant aux contraintes de MPLS. Il va agir au niveau du "hook" PRE_ROUTING afin de pouvoir filtrer tous les paquets qui arrivent. Il peut également s'installer sur le "hook" LOCAL_OUT afin de filtrer les paquets générés localement (mais cela a peu d'intérêt dans notre cas).
L'étude du code de *netfilter* a permis de dégager une architecture interne du filtrage au sein du noyau et par extension au sein de notre LER. La figure 4.2 permet de référencer les différents chemins que peut prendre un paquet.

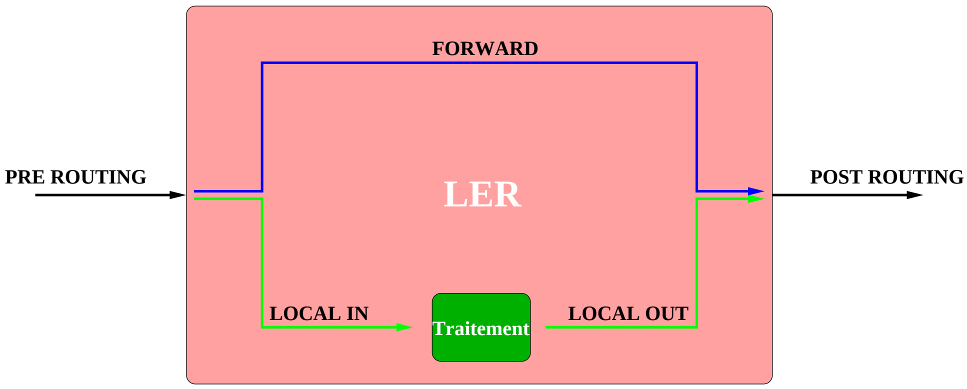

FIG. 4.2 – Schéma de routage dans le LER

La modification a porté à la fois sur :

1. le code de *netfilter*, afin d'ajouter le nouveau "hook" adéquat pour agir,

[7]lecture/écriture

2. le code de l'utilitaire *iptables* pour manipuler ce nouveau "hook".

Depuis sa version 2.4, le noyau Linux fourni une interface propre à l'utilisateur pour pouvoir enrichir *netfilter* (dont les structures sont internes au noyau) ainsi qu'à l'utilitaire appelé *iptables*. Un guide du développeur a été également écrit par Rusty RUSSEL [15].

4.5.2 La réalisation

L'extension du filtrage a nécessité un travail en deux parties :

1. extension du noyau en écrivant un nouveau module et installation au travers d'un patch pour le noyau,
2. extension d'*iptables* en écrivant une nouvelle librairie partagée.

Le module

L'idée de ce module est de se baser sur le principe d'un marquage de paquet semblable à celui du `NFMARK`. En effet, le noyau Linux offre la possibilité d'enrichir la recherche de la clé de "hash" dans la table de routage (cf. section 4.2), par l'indication `NFMARK`. Ce mécanisme utilise *netfilter* pour marquer les paquets. Il permet donc un routage aussi fin que possible.

On veut donc pouvoir marquer les paquets entrant en fonction de leur position dans la table et donc de leur FEC, pour qu'ensuite le traitement se fasse directement en fonction de cette marque qui correspond au `mpls_index` de la table de hashage de MPLS. Ce nouveau module a été baptisée MPLS.

Quatre nouveaux fichiers ont été écrits dans la section *netfilter* du noyau Linux : `ipt_mpls.c`, `ipt_mpls.h` (pour le filtrage), `ip_MPLS.c` et `ip_MPLS.h` (pour le marquage). Il permettent la création d'une nouvelle entrée dans la table `mangle`. Ils ont été conçus en adaptant les fichiers relatifs à `FWMARK` à MPLS, i.e. en remplaçant[8] la marque d'origine (`nfmark`) par une marque MPLS (`fwmpls`).

Nous aurions pu réutiliser le mécanisme `FWMARK`, mais nous souhaitions laisser la possibilité d'avoir les deux traitements séparés et non confondus[9].

Ensuite une recherche de correspondance est faite entre la marque de chaque paquet contenu dans le *skbuff*[10] pour différencier le traitement des paquets devant transiter par MPLS de ceux "voyageant" par la voie classique lors du traitement du premier paquet dans la fonction `rt_set_nexthop()`.

En addition à ce travail, une modification des fichiers de configuration du noyau a aussi été réalisée. Pour simplifier l'application des modifications, la solution a été de créer une archive compressée contenant un ensemble de patches répertoriant toutes les modifications. Parmi les patches livrés, un s'attache à modifier les sources de l'utilitaire *iptables* au moyen d'une nouvelle librairie partagée..

La librairie partagée

Pour pouvoir utiliser les nouvelles fonctionnalités du noyau, il a fallu aussi modifier l'utilitaire *iptables* par l'adjonction d'une nouvelle librairie partagée. Elle est en fait l'équivalent des modifications du noyau dans l'espace utilisateur.

Cette nouvelle librairie a aussi été réalisée sur le même modèle que celle gérant la `FWMARK`. Deux fichiers constituent les sources de cette librairie : `libipt_MPLS.c` et `libipt_mpls.c`. Il va maintenant être possible à *Iptables* de créer de nouvelles règle s'appuyant sur le marquage MPLS (et donc l'index dans la table de routage MPLS).

Les modifications ont aussi fait l'objet d'un patch qui fait partie de l'archive globale.

---

[8] Ce remplacement n'est effectif que si on choisi l'option dans le fichier de configuration du noyau Linux.

[9] En fait, l'utilisation de MPLS masquait complètement le routage basé sur les marques dans ce cas là

[10] zone mémoire contenant un paquet à traiter (en réception et/ou émission).

4.6 Bilan

Si on dresse un bilan du travail qui a été réalisé au niveau de la partie filtrage du noyau Linux, on constate plusieurs choses :

- la navigation dans les sources, l'étude et la compréhension du fonctionnement du noyau et plus particulièrement de la partie filtrage a pris beaucoup de temps. D'un autre côté, tout ce travail a été indispensable avant de pouvoir passer au stade de la réflexion, de la conception et ensuite du codage.
- les patches ont été proposés comme une évolution du noyau à la communauté Linux et plus particulièrement à Jim Leu qui assure le développement de la partie MPLS du noyau.
- en parallèle à tout cela j'ai aussi traduit en français et augmenté d'une partie sur MPLS le document présenté en annexe A[11].

Cette partie filtrage a nécessité entre un mois et demi et deux mois à plein temps et 2 mois de travail de fond pour collecter l'ensemble des données. Et sur cette période, c'est la partie compréhension et analyse qui a encore une fois pris le plus de temps.

[11] Dans sa version française augmentée.

Chapitre 5

La partie commande

5.1 Les besoins

Une fois la plateforme réalisée, il est apparu qu'une interface de commande simplifierait la gestion des labels et des FEC en évitant à l'administrateur de taper plusieurs lignes fastidieuses. Cette interface doit pouvoir répondre à plusieurs exigences :

- ajout d'un LSP associé à une FEC, complète ou partielle,
- suppression d'unLSP existante,
- affichage de tous les LSP et leurs FEC associées,
- utilisation simple en ligne de commande.

Comme dans toute application de ce type, deux parties vont être réalisées : un client et un serveur. Le client ne sert qu'a communiquer avec le serveur, qui lui se chargera de réceptionner les messages et d'agir en conséquence.

La première étape est de déterminer le mode de communication entre les deux entités et de décider du format des messages qu'ils vont s'échanger.

5.2 La communication

Le choix du support de communication a été celui des *sockets*. En effet les *sockets* permettent au client de pouvoir dialoguer avec le serveur quel que soit son lieu d'exécution. En effet, il est possible d'exécuter le client sur une machine différente de celle du serveur sans que cela aie d'influence sur le fonctionnement et la communication. Cette particularité est un atout pour plus tard ou il est quasiment certain que la partie commande sera délocalisée sur une station pendant que le serveur s'exécutera sur le routeur. Mais nous n'en sommes pas encore là.

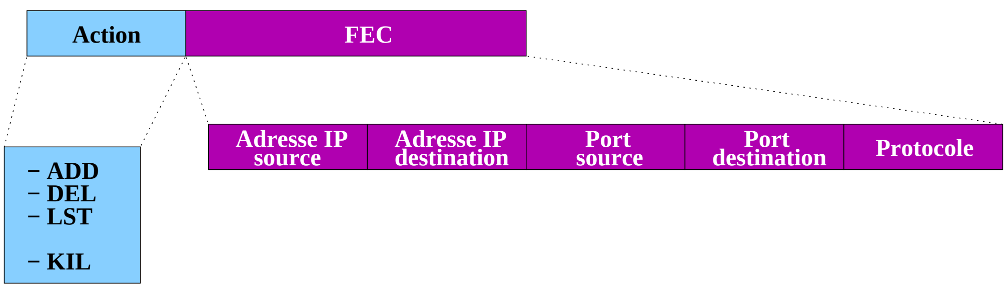

FIG. 5.1 – Type de message entre le client et le serveur

Un petit bilan nous permet de dégager quatre type d'actions que l'utilisateur voudra pouvoir exécuter, à savoir les trois permettant la gestion des LSP plus une permettant d'arrêter le serveur.

Chacune des actions sur les LSP nécessitera de joindre tout ou partie des informations nécessaires (adresse IP source, port source, etc...).

L'ensemble est regroupé au sein d'une structure qui constituera le message type, comme nous le montre la figure 5.1.

5.3 La partie serveur

Le serveur en lui même est assez simple, c'est une boucle infinie passive. Il reste à l'écoute du client[1] et attend l'arrivée des messages. A réception de chacun d'eux, il regarde l'action à réaliser et agit en conséquence en prenant en compte le reste du contenu du message (les données de la FEC dans le cas de la création/effacement/affichage). Il reprend ensuite son attente de message et ainsi de suite.

Dans le cas ou la demande correspond à une destruction il quitte la boucle et se termine.

Le serveur a été intégré à la partie LTCS de notre plateforme de développement et se lance après toutes les instances de créations (LDP, LPE, etc...).

5.4 La partie client

Le client est une partie indépendante du reste de la plateforme. Son fonctionnement est lui aussi très simple. Il contacte le serveur via une socket sur un port commun et après acceptation de la connexion par le serveur, lui envoie un message du type de celui de la figure 5.1. Il attend alors un accusé de réception de la part du serveur et se termine.

Le programme agit comme une commande Unix classique avec une série d'arguments correspondants aux différents types de messages et aux valeurs des champs de la FEC. L'ensemble des possibilités du client est résumé dans le tableau 5.1.

<table>
<tr><th colspan="2"><code>ldp_client -[hvg] -[K|ADL] [-s <v1> -d <v2> -i <v3> -o <v4> -p <v5>]</code></th></tr>
<tr><td>-h</td><td>menu d'aide</td></tr>
<tr><td>-v</td><td>mode bavard, détaille ce qui est fait</td></tr>
<tr><td>-g</td><td>mode debug, pour le déverminage</td></tr>
<tr><td>-K</td><td>pour tuer le serveur</td></tr>
<tr><td>-A</td><td>pour ajouter un LSP (options nécessaires)</td></tr>
<tr><td>-D</td><td>pour supprimer un LSP (options nécessaires)</td></tr>
<tr><td>-L</td><td>pour lister les LSP</td></tr>
<tr><td></td><td>(Description de la FEC)</td></tr>
<tr><td>-s</td><td>adresse IP source (argument nécessaire)</td></tr>
<tr><td>-d</td><td>adresse IP destination (argument nécessaire)</td></tr>
<tr><td>-i</td><td>port source (argument nécessaire)</td></tr>
<tr><td>-o</td><td>port destination (argument nécessaire)</td></tr>
<tr><td>-p</td><td>protocole (argument nécessaire)</td></tr>
</table>

TAB. 5.1 – Arguments et options du client

5.5 Bilan de la partie commande

L'ensemble client serveur est encore rudimentaire. En effet, le serveur se contente d'exécuter un ensemble de commandes compliquées au travers d'un appel a `execv` pour la partie filtrage. L'établissement

[1] sur un port précis de la machine commun au client et au serveur

d'un label se fait elle directement à l'aide du *netlink* MPLS. Une des améliorations futures serait d'attaquer directement le noyau en se basant sur les moyens de communication d'*iptables*.

Chapitre 6

État d'avancement et planning

6.1 Calendrier de développement

L'étude des outils et l'analyse du problème posé ont pris beaucoup de temps, et la phase de réalisation proprement dite a été beaucoup plus rapide, surtout en ce qui concerne le filtrage.

Période	**Travail réalisé**
Phase 1 (Octobre 2000 à mai 2001)	Réalisation du programme LDP et étude de fond sur le filtrage
Phase 2 (Mai - juin 2001)	Évolution du noyau Linux au niveau de *netfilter* Modification d'*iptables* Proposition de ces modifications à la communauté MPLS Linux
Phase 4 (Juillet 2001)	Intégration d'une commande utilisateur via un client/serveur pour les outils de classification

TAB. 6.1 – Calendrier de développement

6.2 État d'avancement

Le développement de ce routeur, qui est devenu la plateforme de développement, n'a pas avancé très rapidement en raison de la période d'acquisition des compétences de bases, mais aussi à cause de la complexité des sources de la version commerciale de MPLS. Il est en effet très difficile de trouver des informations pertinentes.

C'est en effet le point noir de ce projet, le stade de recherche de MPLS fait que tout est en cours de normalisation, de définition, mais que rien n'est à proprement parlé fixé. Il n'est pas rare de voir une nouvelle version de la solution MPLS commerciale intégrer une partie que j'étais en train de programmer de mon côté.

D'autre part, l'état de recherche de MPLS nous fait explorer des pistes qu'il faut ensuite abandonner au profit de nouvelles qui nous semblent plus prometteuses pour y revenir ensuite à la lumière de nouvelles informations...

Si on dresse le bilan de ce qui a été fait, on notera :
- le noyau Linux est correctement configuré, prend en compte MPLS et les utilitaires associés fonctionnent,
- l'outil commercial de Netplane est installé et fonctionne

- l'interconnexion entre le MPLS Linux et l'outil commercial est réalisée[1].
- le programme faisant le lien Linux et le protocole de gestion de labels CR-LDP est aussi terminé,
- les modifications au niveau du noyau pour intégrer MPLS à *netfilter* sont faites,
- une évolution du logiciel *iptables* permet d'exploiter les nouvelles fonctionnalité de *netfilter*,
- un module de commande sur le principe de client serveur a été mis en place et fonctionne.

Mais certaines choses restent encore à améliorer, entre autre :
- réussir à intégrer *Zebra*, à condition que la future version de LTCS finalise ce module,
- améliorer la partie commande en "attaquant" directement le code d'*iptables* pour éviter l'utilisation de la fonction*execv* trop lourde.
- et plus largement contribuer à intégrer CR-LDP dans la version libre du démon de routage pour s'affranchir de la solution commerciale dont nous n'utilisons qu'une petite partie.

[1] Sachant que toute évolution est souvent synonyme de modifications de l'étude

Conclusion

La première partie de mon tutorat m'a permis d'approfondir mes connaissances du protocole ATM, mais aussi de découvrir la notion de qualité de service au travers de l'étude du protocole MPLS. C'est aussi tout un pan de l'implémentation des réseaux sous Linux qui s'est présenté à moi. Et sur la base conjointe des cours de réseau et système de l'ENSSAT et de mes connaissance personnelles du système Linux, j'ai pu rapidement appréhender la nature du travail qui m'incombait.

Cette formule de tutorat permet une approche d'un projet radicalement différente de celle que l'on a l'habitude de rencontrer dans les projets scolaires. Cet abord beaucoup plus professionnel demande des capacités d'analyse et de synthèse que nous n'avons pas l'habitude de mettre à ce point en œuvre. C'est très enrichissant mais cela demande aussi un investissement de tous les instants et une réelle volonté de notre part à s'adapter aux différentes contraintes.
Par exemple, c'est en me confrontant à la documentation des outils utilisés que j'ai pris conscience de leur importance pour les utilisateurs. Et si une information nous manque, il faut synthétiser la base de nos connaissances pour la déduire de celles qui sont disponibles.

Enfin, c'est pour moi la première expérience dans l'univers de la recherche et du développement. Tout est très différent de ce que j'ai pu rencontrer avant, il est indispensable de savoir être autonome et de se donner les moyens pour résoudre soit même les éventuels problèmes auxquels on peut être confronté. Mais heureusement, il est toujours possible de compter sur l'expérience et les compétences des ingénieurs du labo et de mon tuteur en particulier.

Bibliographie

[1] **BOYER Jacqueline, BOYER Pierre, DUGEON Olivier, GUILLEMIN Fabrice, MANGIN Christophe**. *Accelerated Signalling for the Internet over ATM (ASIA).* France Télécom R&D /DAC, 1999. 12 p.

[2] **FERRERO Alexis**. *Les réseaux locaux commutés.* Paris : Masson, 1998. 354 p.

[3] **GUILLEMIN Fabrice, DUGEON Olivier, BOYER Jacqueline, MANGIN Christophe**. Lightweight signaling in ATM networks for high quality transfer of Internet traffic. *Computer Networks*, 2000, vol 34, p 263-278.

[4] **JANIAK Alexis**. *Programmation d'un PC sous Linux en Label Edge Router.* Tutorat Ing. : École National Supérieure des Sciences Appliquées et de Technologie. Lannion, 2000. 41 p.

[5] *Label Traffic Control System (LTCS) Subsystem : Product documentation.* Édité par NetPlane Systems, Inc. Release 3.0. Dedham MA, 2000. 550 p.

[6] **ROLIN Pierre**. *Réseaux haut débit.* Deuxième édition. Paris : Hermes Sciences Publications, 1999. 720 p.

[7] *Layered Environment for Accelerated Portability (LEAP) : Product documentation.* Édité par NetPlane Systems, Inc. Release 2.0. Dedham MA, 2000. 550 p.

[8] MPLS-Linux [En ligne]. Madison (WISC) : Division of Information Technology (DoIT) of the University of Madison-Wisconsin, 2000. Disponible sur Internet à : URL : mpls-linux@nero.doit.wisc.edu

[9] **RIFFLET Jean-Marie**. La communication sous Unix : Applications réparties. Paris : Ediscience International, 1995. 438 p.

[10] **STEVENS Richard W.**. Unix Network Programming volume 1. Networking APIs : Sockets and XTI. Upper Saddle River : Prentice Hall PTR, 1998. 1010 p.

[11] **WRIGHT Gary R., STEVENS Richard W.**. *TCP/IP Illustrated Volume 2 : The Implementation.* Reading : Addison-Wesley Publishing Company, 1995. 1175 p.

[12] **HUBERT B., MAXWELL G., VAN MOOK R., VAN OOSTERHOUT M., SCHROEDER P. B.**. *Linux 2.4 Advanced Routing HOWTO.*Donkerstraat 9a, The Netherlands. Disponible sur Internet à : URL : http ://www.ds9a.nl/.

[13] **RUSSEL R.**. *Linux 2.4 Packet Filtering HOWTO.* Rusty's Remarkably Unreliable Guides. Disponible sur Internet à : URL : http ://netfilter.samba.org/unreliable-guides/.

[14] **WELTE H.**. *The journey of a packet through the linux 2.4 network stack.* Gnumonks.org. Disponible sur Internet à : URL : http ://www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html.

[15] **RUSSEL R.**. *Linux Netfilter Hacking HOWTO.* Rusty's Remarkably Unreliable Guides. Disponible sur Internet à : URL : http ://netfilter.samba.org/unreliable-guides/.

Annexe A

Le voyage d'un paquet au travers de la pile réseau du noyau Linux 2.4

Ce document décrit le trajet d'un paquet réseau à l'intérieur du noyau Linux 2.4.x. Il a changé énormément depuis la version 2.2 avec le choix d'un nouveau système baptisé softirq. De plus on s'attachera à étudier les variantes introduites par la prise en compte du protocole MPLS[1].

A.1 Préface

Je dois m'excuser pour mon ignorance, mais ce document reste très concentré sur le cas par défaut : une architecture x86 et des paquets IP qui sont forwardés.

Ce document est en tout point similaire au fameux "The journey of a packet through the Linux 2.4 network stack" d'Harald Welte, mais contient une partie supplémentaire relative à MPLS.

Je ne suis absolument pas un gourou du noyau et les information fournies par ce document peuvent s'avérer fausses. Alors n'hésitez pas à envoyer vos commentaires ou corrections.

A.2 Réception du paquet

A.2.1 L'interruption en réception

Si la carte réseau reçoit une trame Ethernet qui correspond à l'adresse MAC locale ou qui est en broadcast, celá génère une interruption. Le driver réseau pour cette carte capte l'interruption, va chercher les données du paquet via DMA, PIO ou autre pour les mettre en mémoire RAM. Il alloue alors un *skbuff*[2] et appelle une fonction appartenant aux routines de support de la carte indépendamment du protocole : `net/core/dev.c :netif_rx(skb)`.

Si le driver n'a pas encore daté le skb, c'est alors fait. Après cela, le *skbuff* est placé dans la file d'attente appropriée pour que le processeur puisse le traiter. Si la file est pleine, le paquet est détruit. Après la mise en file d'attente du skbuff la réception de l'interruption logicielle est marquée pour exécution via
`include/linux/interrupt.h :__cpu_raise_softirq()`.

Le traitement de l'interruption se termine et toutes les interruptions sont ré-autorisées.

[1]MPLS : Multi Protocol Label Switching
[2]skbuff : socket buffer

A.2.2 Le *network RX softirq*

Là nous rencontrons l'un des changements majeurs entre 2.2 et 2.4 : la pile réseau entière n'est plus géré par principe de *polling* mais par interruptions logicielles[3]. Ces dernières ont l'avantage de pouvoir s'exécuter sur plusieurs processeurs simultanément. La gestion précédente ne pouvait se faire qu'avec un seul processeur à la fois.

L'enregistrement de la softirq de réception réseau se fait dans `net/core/dev.c :net_init()` en utilisant la fonction `kernel/softirq.c :open_softirq()` fournie par le sous-système de softirq.

Le traitement ultérieur des paquets est réalisé dans la softirq de réception réseau (NEXT_RX_SOFTIRQ) qui est appelée depuis `kernel/softirq.c :do_softirq()`. La fonction `do_softirq()` est elle-même appelée depuis trois endroits dans le noyau :

1. depuis `arch/i386/kernel/irc.c :do_IRQ()`, qui assure le traitement des IRQ génériques,
2. depuis `arch/i386/kernel/entry.S` dans le cas où l'on revient juste d'un appel système,
3. à l'intérieur du processus principal de l'échéancier dans `kernel/sched.c :schedule()` .

Donc si l'exécution accompli un de ces points, la fonction `do_softirq()` est appelée, et détecte la marque NET_RX_SOFTIRQ et appelle `net/core/dev.c :net_rx_action()`. Le skb est enlevé de la queue de réception de ce processeur et ensuite traité par la fonction de traitement de paquet approprié. Dans le cas d'IPV4, il s'agit de celle associée aux paquets IPV4.

A.2.3 La fonction de traitement des paquets IPV4

La fonction de traitement des paquets IP est enregistrée via `net/core/dev.c :dev_add_pack()` appelé depuis `net/ipv4/ip_output.c :ip_init()`.

La fonction en charge du traitement du paquet IPV4 est `net/ipv4/ip_input.c :ip_rcv()`.
Dans le cas de MPLS, on sera passé avant par la fonction `mpls_rcv()`. Cette fonction va rechercher dans la table de hashage des labels (Radix_Tree) une entrée correspondant au label reçu. En fonction du *moi* récupéré, elle va :
- supprimer le label et envoyer le paquet à `ip_rcv()` pour un traitement normal (cas du LER),
- réexpédier le paquet avec un nouveau label (cas du LCR) en faisant abstraction de la couche IP,
- dépiler le label et utiliser le label suivant comme nouvelle entrée dans la table de hashage pour connaître le traitement à effectuer (cas du *label stacking*)

Après quelques vérifications préliminaires (si l'entête du paquet est correcte[4] et si la taille est supérieure à 20 octets) le code vérificateur[5] est calculé.

Tous les paquets échouant à l'un des tests précédents est alors détruit[6].

Si le paquet passe les tests avec succès, on détermine la taille du paquet IP et on débarrasse le skb du bourrage éventuellement rajouté par la couche transport.

A cet instant, c'est le premier appel aux *netfilter hooks.*

[3] softirq
[4] IPv4
[5] checksum
[6] dropped

Netfilter fournit une interface générique et abstraite au code de routage standard. Il est utilisé pour filtrer les paquets, les découper, la traduction d'adresse (NAT)[7] et la gestion des files d'attente vers l'espace utilisateur. Pour plus de détails, jetez un œil sur ma présentation "Le sous système netfilter dans Linux 2.4" ou un des guides douteux de Rusty, i.e. the netfilter hacking guide.

Après une traversée réussie du netfilter hook, la fonction
`net/ipv4/ipv_input.c :ip_rcv_finish()` est appelée.

A l'intérieur de `ip_rcv_finish()`, la destination du paquet est déterminée par l'appel à la fonction de routage
`net/ipv4/route.c :ip_route_input()`. De plus, on recherche si une clef correspondant à ce flot de paquet[8] n'existe pas dans la table de hashage. En outre, si notre paquet IP possède des options IP, elles sont traitées à ce moment. Si aucune clef n'existe, elle est calculée et insérée dans la table. Il est alors fait appel à
`net/ipv4/route.c :ip_route_input_slow()`. Dans tous les cas, le trajet de notre paquet se poursuit par l'appel de la méthode `input` du skbuff qui à été positionnée dans la fonction
`net/ipv4/route.c :ip_route_input()` à l'une des fonctions suivantes :

- `net/ipv4/ip_input.c :ip_local_deliver()`
  La destination du paquet est locale, nous devons nous occuper de la couche 4 du protocole et le transmettre à un processus utilisateur.
- `net/ipv4/ip_forward.c :ip_forward()`
  La destination du paquet n'est pas locale, nous devons le transmettre à un autre réseau.
- `net/ipv4/route.c :ip_error()`
  Une erreur est survenue, il n'est pas possible de trouver une entrée appropriée pour ce paquet dans la table de routage.
- `net/ipv4/ipmr.c :ip_mr_output()`
  C'est un paquet en multicast et nous devons effectuer un routage multicast.

A.3 Transmission de paquet à une autre entité physique (device)

Si le routage décide que le paquet doit être transmis, la fonction
`net/ipv4/ip_forward.c :ip_forward()` est appelée.

La première tâche de cette fonction est de vérifier la partie de l'entête IP correspondant au TTL[9]. Si elle est inférieure ou égale à 1, on détruit le paquet et on renvoie un message ICMP indiquant une durée de vie dépassée à l'expéditeur du dit paquet.

On vérifie alors l'entête du skb pour savoir si il reste suffisamment de place en fin de zone mémoire pour y placer l'entête de la couche liaison de l'entité. Le cas échéant, on agrandit le skb en conséquence.

Ensuite le TTL est décrémenté de 1.

Si notre paquet est plus grand que le MTU[10] de destination et si le bit de non-fragmentation est à 1 dans l'entête IP, alors on détruit le paquet et on renvoie un message ICMP à l'expéditeur indiquant qu'une fragmentation est nécessaire.

Enfin il est temps de faire appel à un autre netfilter hook : NF_IP_FORWARD.

[7] Network Address Translation
[8] Ce paquet est soit le premier d'un flot soit membre d'un flot existant
[9] Time To Live : durée de vie du paquet
[10] Maximum Transport Unit : taille maximale d'un unité de transport dans un réseau

En considérant que le netfilter hook retourne la valeur NF_ACCEPT, la fonction `net/ipv4/ip_forward :ip_forward_finish()` est la prochaine étape du voyage de notre paquet.

La fonction `ip_forward_finish()` vérifie elle-même si il est nécessaire de traiter d'éventuelles options dans l'entête IP, et fait appel à la fonction `net/ipv4/ip_options.c :ip_forward_option()`. Après cela, elle appelle `include/net/ip.h :ip_send()`.

Ensuite, `ip_send()` est appelée et vérifie si une fragmentation est nécessaire[11], puis on continue dans la fonction `skb->dst->output()`, qui peut être l'une des deux suivantes :
- `net/ipv4/ip_forward :ip_output()` dans le cas de paquets classiques,
- `net/mpls/mpls_output :mpls_output()` avec MPLS.

A.3.1 Suite du voyage par la voie classique

La fonction `ip_output()` traite le NAT[12] puis appelle le netfilter postrouting hook NF_POSTROUTING_HOOK et `ip_finish_output2()` après une traversée réussie du hook.

L'appel à la fonction `ip_finish_output2()` met le matériel en attente de notre skb et appelle `net/ipv4/ip_output.c :ip_output()`.

La suite est conditionnée par le type d'interface :
- ATM : on fait appel a la fonction `send_ATM_SKB` pour expédier le contenu du skbuff sur le *vcc* déterminé lors de `ip_output()`.
- WAN : on ne fait rien (pas implémenté),
- Ethernet (cas par défaut) :
  - si l'entête correspondant au matériel est positionnée dans la structure *dst_entry*, on appelle `include/linux/skbuff.h :skb_push()` pour terminer la préparation du paquet dans le skbuff.
    Ensuite on fait appel à la fonction contenue dans `hh->hh_output(skb)`, qui n'est autre que `net/core/dev.c :dev_queue_xmit()`[13] qui se charge d'envoyer un paquet contenu dans un skbuff vers la carte réseau pour émission.
  - sinon, si `skb->dst->neighbour` est affecté, on fait appel à `skb->dst->neighbour->output()`
  - sinon on libère la mémoire occupée par le skbuff et rien n'est envoyé.

A.3.2 Suite du voyage par la voie MPLS

La fonction `mpls_output()` permet de récupérer le `moi`[14] soit directement depuis la structure `skb->dst->dst_proto_data[AUX_PROTO_DATA_MPLS]` soit par une recherche via l'index mpls[15]dans le *RADIX_Tree*.

Ensuite, on fait appel à la fonction `net/mpls/mpls_output.c :mpls_outpu2()` pour continuer notre voyage.

Dans `mpls_output2()`, pour chaque instruction du *moi*, on va tester le code de l'opération qui lui est associé. Il peut être de quatre type :
- `MPLS_OP_NOP` : il n'y a rien à faire,

[11]Si c'est le cas, il y aura appel à la fonction `net/ipv4/ip_fragment.c`

[12]Networking Address Translation : système de traduction d'adresses pour les réseaux privés dans le cas d'un firewall par exemple

[13]Cette association se réalisant dans `net/ipv4/route.c`

[14]`mpls output information` : les informations MPLS relatives à la sortie des paquets

[15]`skb->mpls_index`

- `MPLS_OP_FWD` : on doit retransmettre le paquet (surtout ce qu'il contient). Le *moi* prend la valeur de la donnée associée à l'instruction. Et on recommence le traitement sur la nouvelle valeur du *moi*.
- `MPLS_OP_PUSH` : le *moi* prend la valeur de la donnée associée à l'instruction. Ensuite, on appelle la fonction `net/mpls/mpls_opcode.c :mpls_opcode_push()` qui préparera le paquet (label, shim, ...) dans le skbuff avant l'envoi.
- `MPLS_OP_SET` : la carte réseau du skbuff devient celle du *moi*. Selon son type : ATM, WAN ou Ethernet (par défaut), on attribue certaines valeurs du *moi* au skbuff (vcc pour l'ATM, structure *dst* pour l'Ethernet, ...). Dans le cas de l'Ethernet, le protocole devient ETH_P_MPLS_U.

Ensuite, la fonction `net/mpls/mpls_input.c :mpls_finish()` va nettoyer la structure du paquet contenu dans le skbuff en éliminant le bourrage qui aurait pu être généré par MPLS.

La suite est conditionnée par le type d'interface :

- ATM : on fait appel a la fonction `send_ATM_SKB` pour expédier le contenu du skbuff sur le *vcc* déterminé lors du cas `MPLS_OP_SET`.
- WAN : on ne fait rien (pas implémenté),
- Ethernet (cas par défaut) :
  - si l'entête correspondant au matériel est positionnée dans la structure *dst_entry*, on appelle `include/linux/skbuff.h :skb_push()` pour terminer la préparation du paquet dans le skbuff.

    Ensuite on fait appel à la fonction contenue dans `hh->hh_output(skb)`, qui n'est autre que `net/core/dev.c :dev_queue_xmit()`[16] qui se charge d'envoyer un paquet contenu dans un skbuff vers la carte réseau pour émission.
  - sinon, si `skb->dst->neighbour` est affecté, on fait appel à `skb->dst->neighbour->output()`
  - sinon on libère la mémoire occupée par le skbuff et rien n'est envoyé.

[16] Cette association se réalisant dans `net/ipv4/route.c`

A.4 Le voyage classique d'un paquet en un schéma

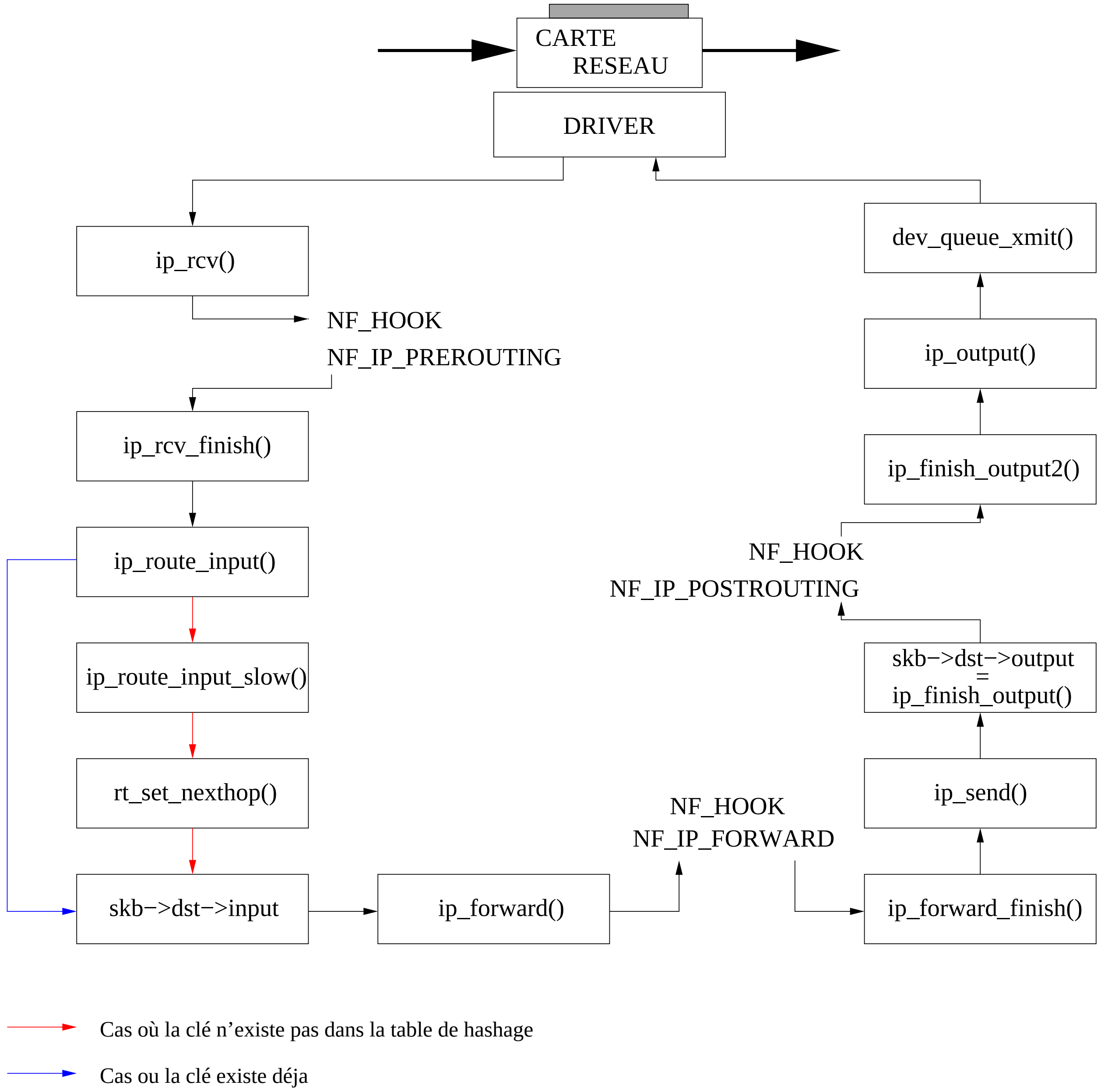

FIG. A.1 – Le voyage classique d'un paquet

A.5 Le voyage d'un paquet en un schéma dans le cas de MPLS

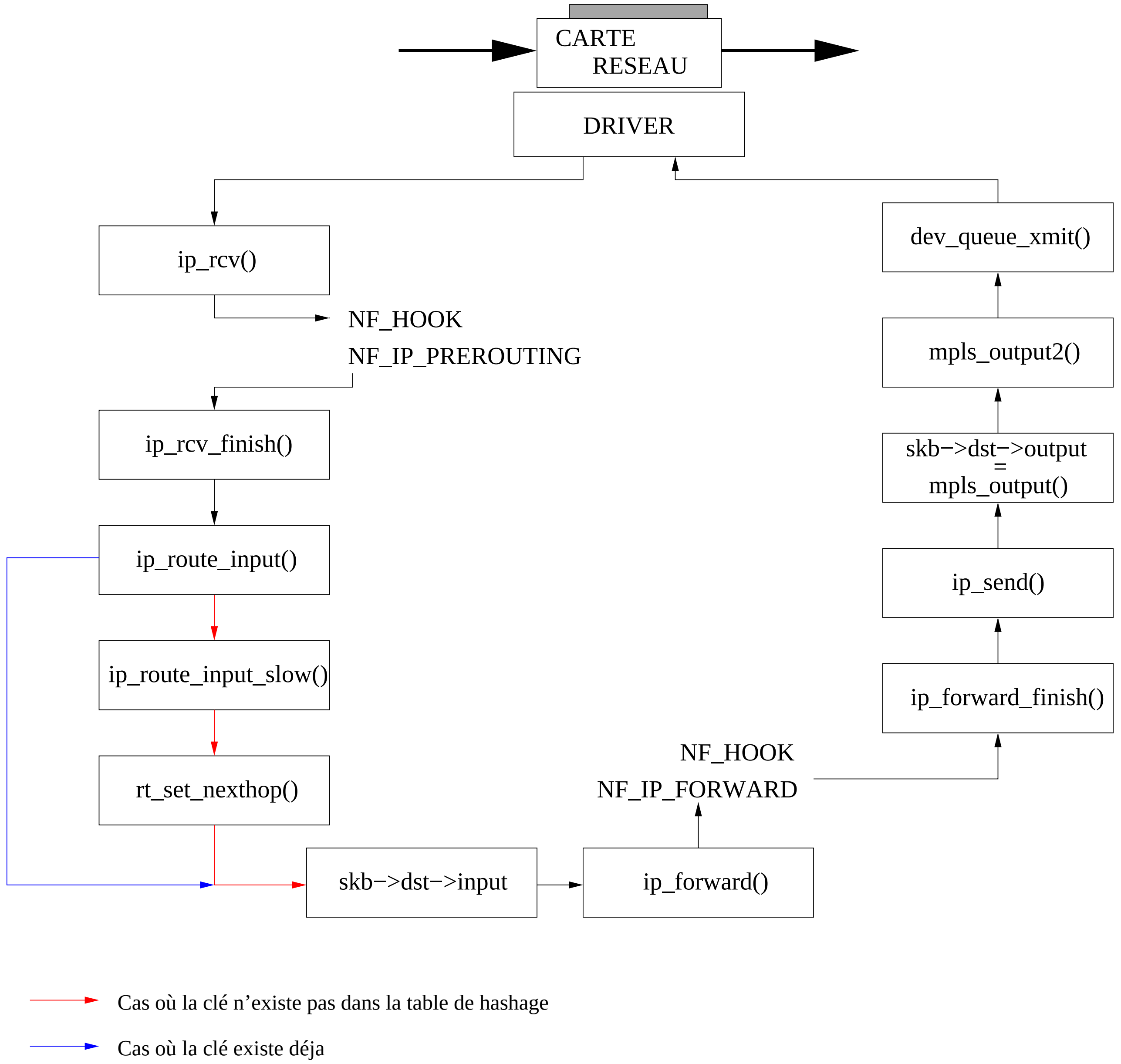

FIG. A.2 – Le voyage classique d'un paquet dans le cas MPLS