Routage sensible à la source

Matthieu Boutier

Thèse de l'université Sorbonne Paris Cité
Préparée à l'université Paris Diderot — Paris 7
Institut de Recherche en Informatique Fondamentale (IRIF)
École Doctorale 386 — Sciences Mathématiques de Paris Centre

Routage sensible à la source

Matthieu Boutier

Thèse de doctorat — Spécialité informatique
Soutenue à Paris le 20 septembre 2018 devant le jury composé de :

Juliusz Chroboczek *Maître de conférences à l'IRIF*	Directeur
Laurent Viennot *Directeur de recherche à l'INRIA*	Président du jury
Timothy G. Griffin *Professeur à l'Université de Cambridge*	Rapporteur
Thomas Clausen *Professeur à l'École Polytechnique*	Rapporteur
Anne Bouillard *Chercheur au LINCS*	Examinatrice

Routage sensible à la source

Résumé En routage next-hop, paradigme de routage utilisé dans l'Internet Global, chaque routeur choisit le next-hop de chaque paquet en fonction de son adresse destination. Le routage sensible à la source est une extension compatible du routage next-hop où le choix du next-hop dépend de l'adresse source du paquet en plus de son adresse destination. Nous montrons dans cette thèse que le routage sensible à la source est adapté au routage des réseaux multihomés avec plusieurs adresses, qu'il est possible d'étendre de manière compatible les protocoles de routage à vecteur de distance existants et que ce paradigme de routage offre avantageusement plus de flexibilité aux hôtes. Nous montrons d'abord que certains systèmes n'ordonnent pas correctement les entrées sensibles à la source dans leurs tables de routage et nous définissons un algorithme adapté aux protocoles de routage pour y remédier. Nous montrons comment étendre les protocoles à vecteur de distances au routage sensible à la source de manière compatible. Nous validons notre approche en concevant une extension d'un protocole existant (Babel), en réalisant la première implémentation complète d'un protocole sensible à la source et en utilisant ce protocole pour router un réseau multihomé. Enfin, nous montrons que le routage sensible à la source offre des possibilités de multichemin aux couches supérieures des hôtes. Nous vérifions qu'il s'intègre aux technologies existantes (MPTCP) et nous concevons des techniques d'optimisation pour les applications légères. Nous évaluons ces techniques après les avoir implémentées dans le cadre d'une application existante (mosh).

Mots-clefs routage source multihoming SADR SAD routing table conflit multichemin

Source-specific routing

Short abstract With next-hop routing, the routing paradigm used on the Global Internet, each router chooses the next-hop of each packet depending on its destination address. Source-specific routing is a compatible extension of next-hop routing in which the choice of the next-hop depends on the source address of the packet in addition to its destination address. In this thesis, we show that source-specific routing is well adapted to multihomed networks with multiple addresses, that extending a distance vector routing protocol and ensuring compatibility with the base protocol is possible and that source-specific routing gives more flexibility and thus new possibilities to hosts. First, we show that on some systems, source-specific routing tables are not correctly interpreted and we define an algorithm designed for a routing protocol to fix it. We show how to extend distance vector routing protocols to source specific routing while ensuring compatibility. We validate our approach with the conception of an extension to an existing protocol (Babel), with the realization of the first complete implementation of a source-specific routing protocol and with the use of this protocol to route a multihomed network. Lastly, we show that source-specific routing gives multipath possibilities to host's highest layers. We check that it works well with existing technology (MPTCP) and we design optimization techniques for lightweight applications. We evaluate these techniques after their implementation in an existing application (mosh).

Keywords routing source multihoming SADR SAD table conflict multipath

Remerciements

Cette thèse, longue et laborieuse, aura été à la fois l'objet de grandes joies et de grandes peines. Je tiens à remercier Juliusz qui m'a accompagné tout au long de cette aventure et avec qui j'ai découvert des lieux et des personnes insoupçonnés aux quatre coins de l'Europe, du fin fond de Paris à Varsovie en passant par Berlin (et sa faune) et Maribor. Merci à lui pour les nombreuses anecdotes qu'il sait raconter avec une touche parcimonieuse d'exagération qui relève le goût des faits sans les trahir. Merci à lui pour son indéfectible loyauté malgré les incompréhensions et les moments les plus difficiles. Je tiens aussi à remercier Yves, qui a joué le rôle d'un co-directeur aux heures où cela s'avérait nécessaire, et Alexis ; sans eux, cette thèse n'aurait certainement pas abouti. Je remercie Timothy Griffin pour notre heureuse (et inattendue) rencontre à Venise, pour son invitation à Cambridge et pour avoir accepté de rapporter ma thèse. Merci à Laurent Viennot pour nous avoir plusieurs fois écouté et conseillé. Je remercie l'ensemble des membres du jury d'avoir accepté d'en faire partie.

Je remercie tous ceux qui ont contribué à l'aboutissement de ce travail, notamment à ceux qui ont relu la totalité de ce manuscrit (ou de ses versions antérieures) : Clément, qui s'est intéressé à ce travail et qui m'a toujours proposé de relire mon manuscrit, jusqu'au bout ; Julien, qui a quitté à grand peine son animalerie pour l'informatique, qui m'a souvent prêté son canapé (grèves un jour...) et qui m'a enrichi de ses conseils et de son amitié ; Daphné, avec qui j'ai partagé, bien plus doux qu'une thèse, champagnes, liquoreux et surtout sa grande amitié ; mes parents pour leur constant soutien et leurs remarques pertinentes ; et Dorothée avec qui j'ai partagé bien plus que deux ans de thèse. Merci aussi à ceux qui m'ont aidé de diverses manières, et en particulier merci à Zeinab qui m'a souvent représenté auprès de l'administration et de la logistique, et m'a fait répéter ma thèse (à ce propos, merci aussi à Athénaïs, Léo et Rémi). Je lui dois beaucoup, même si elle appelle ça un *rien du tout*.

J'ai été très heureux de trouver sur mon chemin de nombreuses personnes de qualité. Je pense tout d'abord à ceux que j'ai directement rencontré dans le cadre de cette thèse, et en premier lieu Gabriel, mon talentueux prédécesseur. Merci à lui d'avoir encadré mes premiers travaux de recherche et pour son amitié constante.

Merci aux premiers thésards que j'ai rencontré alors qu'ils terminaient leur thèse : Grégoire qui faisait tourner Babel sous mac OS ; Thibaut qui a écrit sa thèse en plusieurs volumes ; et, bien sûr, ceux dont j'ai partagé le bureau : Stéphane qui a fait une thèse que l'on croyait on ne peut plus longue (en temps) ; Alexis qui m'a invité à rejoindre ses combats ; Flavien, l'homme qui parle plus vite que ses lèvres — un merci tout particulier à lui pour avoir imprimé ses remerciements séparément de sa thèse. Merci aux thésards qui ont suivi : merci à Lourdes pour son amitié, sa gaieté et son soutien aux heures qui sont passées les plus vite ; merci à Ioana pour son humour (et merci pour les sous-titres) ; merci à Pierre-Marie qui nous abreuva de thé au matcha ; merci à Guillaume pour sa "tisane du paradis" et surtout pour m'avoir attendu pour soutenir ; merci à Shahin pour la simplicité et la logique avec laquelle il peut défendre des positions parfois tordues ; merci à Kuba d'avoir mis ce coin de laboratoire à cheval avec une autre dimension ; merci à Charles — si si, un jour, on arrivera à prendre l'apéro ensemble ; merci à Cyrille, le spécialiste du moyen âge ; merci à Ludovic d'avoir partagé sa passion pour les serpents ; merci à Clément D. pour ses jeux de

mots sur les noms. Merci enfin à la plus jeune génération de thésards que j'ai connu, en commençant par ceux qui ont pris la relève dans mon bureau : Hadrien et sa passion pour les films à public restreint, Raphaëlle pour m'avoir fait découvrir la vraie nature du sarrasin, et Thibaut pour m'avoir montré que les araignées (et autres monstres ?) sont nos amies. Merci à Rémi de prendre la relève spirituelle, et à Yann d'avoir partagé les mêmes derniers combats administratifs. Merci à ceux avec qui j'ai partagé la charge du séminaire thésards, ou qui ont pris ma relève, dont Virginie, Alexis, Marie, Clément, Clément, et Théo. Merci à Kenji et Léo d'être régulièrement passé dans notre bureau. Merci aux autres thésards qui ont marqué le sol de ce laboratoire, dont Amina, Étienne, Pierre. Merci aussi à ceux qui seront passés comme stagiaires, notamment Baptiste.

Merci aussi à tous les permanents de PPS avec qui j'ai plus ou moins échangé mais dont quelque chose me restera. Merci au Liafa d'avoir voulu intégrer PPS pour former le prestigieux Institut de Recherche en Informatique Fondamentale [1]. Je ne saurais citer tous les noms de ceux qui m'auront marqué (surtout que je suis déjà en retard pour les remerciements), mais je tiens à nommer Yann, à la fois pour son enseignement et son amitié répétée ; Yves, une des premières Grandes figures de PPS que j'ai rencontré ; Antonio qui me partagea le fond de sa gourde alors que nous étions assoiffés ; Pierre-Louis qui n'hésite pas à se faire proche de tous ; Michel grâce à qui les étudiants ne viennent pas trop nous embêter (merci surtout pour cette poursuite de la justice et de la vérité) ; Fabien ; Delia ; Mihaela ; Ines ; Sylvain et tant d'autres. Merci à Alexis et Christine de prendre soin des thésards.

Enfin, merci à ceux qui font vivre le laboratoire. Elle n'est plus parmi nous, mais merci à Odile qui a été l'âme de PPS pendant de nombreuses années. Merci à Étienne qui a pris la relève et qui s'intéresse aux recherches de tous. Merci à Nicolas qui veille sur l'UFR et grâce à qui nous connaissons le vrai numéro de salle pour la soutenance.

Je tiens aussi à remercier la société Intersec chez qui je travaille actuellement, et en particulier Romain qui s'est montré toujours conciliant pour que je puisse au mieux finir la thèse en me permettant de passer à temps partiel et acceptant de poser les demi-journées à la volée. Un grand merci à lui surtout pour la manière dont il gère l'équipe. Merci à Fabien pour son humour raffiné, à Hugo et Benoît et leur partage du thé, à Christèle pour son calme, et à RE de réagir si bien aux taquineries.

Un grand merci aussi à tous mes amis qui m'ont porté et supporté, dans le désordre : Maxime et Thérèse, Daphné, Florence, Edmond et Évelyne, Blandine, Damien, Véronique et Mahéri, Ariane et Bruno, Anna et Christophe (J-2), Anne et Ludovic, Solange et Guy, Julien, Sarah, Diane, Thibaut et Marie-Lys, Agnès, Wendy, Franck et Héliette, Madeleine et Laurent, Marie-Caroline et Pierre-Louis, Pauline et Colin, Laurène, Loïse, tous ceux qui ont prié pour moi et tous ceux pour qui je vais m'en vouloir de les avoir oubliés dans cette liste...

Enfin, je remercie d'une manière particulière ma famille qui m'a accompagné tout au long du chemin (et pas seulement jusqu'à la gare). Merci à Dorothée d'avoir été là (presque) au quotidien, de m'avoir toujours soutenu durant cette thèse, et avec qui nous avons découvert de nouveaux horizons. Merci à elle de m'avoir choisi pour fiancé. Merci à Jésus, mon "ami imaginaire" pour ceux qui ne le connaissent pas, et mon roc inébranlable pour ceux qui ont eu la chance de le rencontrer.

1. ;-)

Table des matières

Abstract

Le routage sensible à la source est une extension du paradigme de routage le plus répandu dans l'Internet Global. Nous montrons comment il résout des problèmes difficiles, notamment le routage des paquets dans certains types de réseaux multihomés*, et qu'il offre avantageusement plus de flexibilité aux hôtes.*

Le paradigme de routage utilisé dans l'Internet est le routage *next-hop*. Les paquets sont acheminés de routeur en routeur jusqu'à leur destination. Lorsqu'un paquet arrive à un routeur, le routeur décide vers quel routeur voisin envoyer le paquet — ce voisin est appelé *next-hop*. Le choix du *next-hop* ne dépend que de la destination du paquet et de la table de routage du routeur. Celle-ci est une liste d'association qui fait correspondre des *next-hop* à des adresses destination. Pour peupler la table de routage de manière automatique et dynamique, on a recours à un protocole de routage.

Dans ce manuscrit, nous nous intéressons à une modeste extension du routage *next-hop* : le routage sensible à la source. En routage sensible à la source, le choix du *next-hop* par les routeurs dépend, en plus de l'adresse destination, de l'adresse source des paquets. Le routage sensible à la source résout des problèmes difficiles, notamment le routage d'un certain type de réseaux *multihomés*, tout en étant compatible avec le routage *next-hop*.

Réseaux multihomés Un réseau se connecte à l'Internet par un fournisseur d'accès (FAI). Celui-ci fournit au réseau des adresses IP et une route vers l'Internet. Les tables de routage des routeurs du réseau sont configurées par un protocole de routage dynamique. Celui-ci assure la fiabilité des connexions entre deux points du réseau (si un routeur tombe en panne, le protocole de routage détecte la panne et achemine les paquets par un autre chemin). Dans cette configuration, nous voyons qu'il existe un point unique de défaillance : le routeur connecté au FAI — s'il tombe en panne, tout le réseau est coupé de l'Internet. Pour augmenter la fiabilité du réseau, nous pouvons connecter le réseau à plusieurs FAI : c'est le *multihoming*. Notons qu'il y a d'autres raisons de vouloir faire du *multihoming*, comme par exemple augmenter les performances ou réduire les coûts.

Un exemple courant de nos jours, mais réduit à un seul hôte, est le téléphone portable. Celui-ci peut être connecté à la fois au wifi et au réseau cellulaire. Dans un certain nombre de cas, le réseau wifi offre au téléphone de meilleures performances, tant en débit qu'en latence, et n'a pas de limite de téléchargement [2]. En revanche, le réseau cellulaire offre une meilleure fiabilité puisqu'il est plus souvent accessible. Dans ce cas, le téléphone a intérêt à rester connecté autant que possible en wifi et, lorsqu'il en devient hors de portée, à basculer sur le réseau cellulaire.

Or, le téléphone portable est directement connecté à ses deux fournisseurs d'accès. Il sait exactement quel fournisseur d'accès il choisit en choisissant l'interface par laquelle il y est connecté. Aucun routage en l'hôte et ses fournisseurs n'est nécessaire. Notons pour la suite que les paquets sortant par une interface reçoivent l'adresse IP associée à cette interface.

2. Qui sait, dans quelques années on ne trouvera peut-être plus que des forfaits sans limite de données.

Routage sensible à la source dans un réseau multihomé Le routage sensible à la source s'applique à un certain type de *multihoming* où chaque fournisseur d'accès se comporte comme s'il était le seul fournisseur du réseau. Le réseau se retrouve avec autant de plages d'adresses IP et de routes vers l'Internet que de fournisseurs. Or, puisque chaque fournisseur a fourni une plage d'adresses IP, il s'attend à ce que le réseau lui envoie des paquets qui ont pour adresse source une adresse de cette plage d'adresses. Si ce n'est pas le cas, les paquets sont détruits. Il convient donc de router chaque paquet en fonction de son adresse source vers le fournisseur de son adresse source.

Le routage sensible à la source répond exactement à ce besoin. Chaque route fournie par un FAI se voit associée aux adresses fournies par le FAI. Les paquets à destination de l'Internet et ayant pour adresse source celle d'un FAI sont bien routés par cet FAI. Au contraire, en routage *next-hop*, toutes les routes fournies par les FAI sont identiques et tous les paquets sont envoyés au même FAI, indépendamment de leur adresse source.

Mais ne nous arrêtons pas là. Si chaque hôte du réseau a une adresse par fournisseur et que le réseau est routé par routage sensible à la source, nous voyons que le choix de l'adresse source des paquets coïncide avec le choix d'un fournisseur d'accès. Cette situation est identique au cas de l'hôte directement connecté à plusieurs fournisseurs d'accès, illustré par le téléphone. Cela signifie que les mêmes techniques utilisées pour un hôte *multihomé* peuvent être utilisées pour les réseaux *multihomés* avec routage sensible à la source.

En effet, des techniques particulières sont nécessaires pour que les hôtes tirent pleinement profit du *multihoming*. Le protocole de transport le plus répandu, TCP, identifie une connexion par deux adresses IP (locale et distante) et deux numéros de ports (local et distant). Si une panne survient au niveau d'un fournisseur d'une de ces deux adresses, TCP perd la connexion, même si l'utilisation d'autres adresses rendrait possible la connexion.

MPTCP est une extension récente et compatible de TCP qui est capable d'utiliser différents chemins pour survivre aux pannes. En plus du flot TCP initial pour une paire d'adresses donnée, MPTCP établit un nouveau sous-flot pour chaque autre paire d'adresses source et destination possible. Lorsqu'une panne survient sur le chemin emprunté par un un sous-flot (y inclus le flot initial), le trafic peut passer par un autre sous-flot sans que la connexion soit perdue. MPTCP est aussi capable d'augmenter le débit en répartissant la charge sur plusieurs chemins. Puisqu'un sous-flot dépend d'une paire d'adresses destination et source, MPTCP est capable d'utiliser toutes les routes fournies par le routage sensible à la source.

Donc, en plus de router correctement les paquets, le routage sensible à la source rend possible la mise en place de techniques multichemin qui peuvent augmenter les performances des applications. Cela est d'autant plus vrai que les performances mesurées par un protocole de routage sont internes au réseau où est déployé ce protocole, tandis que les performances mesurées par les applications sont celles du chemin de bout en bout. Le routage sensible à la source garde bien la séparation entre ces deux problèmes complémentaires qui peuvent être résolus de manière indépendante.

Ambiguïtés dans les tables de routage En routage *next-hop*, les tables de routage associent des *next-hop* à des plages d'adresses, appelées préfixes. Il arrive fréquemment que ces plages d'adresses se chevauchent, et soient donc susceptible de router un même paquet. Il y a alors une ambiguïté quant au choix à effectuer. Celle-ci est levée en appliquant la règle du préfixe le plus spécifique : l'entrée qui route le moins de paquets est sélectionnée. Cette règle est notamment assez naturelle en présence d'une "route par défaut", qui accepte tous les paquets : si aucune autre entrée de la table de routage (nécessairement plus spécifique que la route par défaut) n'accepte le paquet, le *next-hop* de la route par défaut est utilisé.

En routage sensible à la source, les tables de routage associent des *next-hop* à des paires de préfixes : un préfixe pour les adresses destination et un préfixe pour les adresses source. Des ambiguïtés similaires apparaissent, mais la spécificité des préfixes ne suffit plus à les lever. En effet, une entrée peut avoir son préfixe destination plus spécifique que celui d'une autre entrée et au contraire son préfixe source moins

spécifique que celui de l'autre entrée. Aucune de ces deux entrées n'est alors plus spécifique que l'autre. Le consensus actuel est de préférer l'entrée qui a le préfixe destination le plus spécifique, et en cas d'égalité le préfixe source le plus spécifique. Nous parlons d'ordre par la destination d'abord.

Contributions Dans ce manuscrit, nous nous intéressons principalement à la mise en œuvre d'un routage sensible à la source dynamique, mais aussi aux bénéfices qu'une application peut en tirer (dans le cadre des réseaux *multihomés*).

Nos contributions, détaillées à la section 1.7, sont les suivantes :

— Nous définissons un algorithme adapté aux protocoles de routage qui leur permet de configurer les tables de routage des routeurs pour qu'elles se comportent comme des tables dont les entrées sont ordonnées par la destination d'abord.

— Nous montrons comment réaliser ou étendre un protocole de routage sensible à la source à partir d'un protocole de routage *next-hop* à vecteur de distances.

— Nous concevons et implémentons une extension sensible à la source pour un protocole de routage à vecteur de distances, en mettant en œuvre nos deux précédentes contributions. Cette implémentation constitue, à notre connaissance, la première implémentation complète du routage sensible à la source.

— Nous montrons que le routage sensible à la source offre aux couches supérieures plusieurs chemins et que certains protocoles existants (notamment MPTCP) sont capables de les exploiter.

— Nous montrons que la connaissance d'une application permet d'optimiser ses performances dans un milieu multichemin. Nous nous attachons notamment aux applications légères et interactives, où la duplication des paquets nous permet de récupérer la connexion plus vite après une panne et de limiter les pertes de paquets.

Chapitre 1

Introduction

Le paradigme de routage utilisé dans l'Internet est le routage *next-hop* (section 1.1). Dans ce manuscrit, nous nous intéressons au routage sensible à la source, une extension du routage *next-hop*, et au lien entre le routage et les hôtes. Cette extension nécessite à la fois la modification du routage du réseau (section 1.4), et la modification de l'opération de transfert des routeurs (section 1.5). De plus, en routage sensible à la source, le chemin emprunté par les paquets dépend de leur adresse source (en plus de leur adresse destination). Si un hôte a plusieurs adresses, ce routage peut lui offrir plusieurs routes pour une même destination : l'hôte choisit alors le chemin des paquets en choisissant leur adresse source (section 1.6). Le routage sensible à la source a son application principale dans certains types de réseaux *multihomés* (section 1.3.3).

1.1 Un réseau Internet classique

Un réseau est composé d'hôtes et de routeurs interconnectés par des liens reliant leurs interfaces. Les hôtes sont usuellement représentés par des carrés, et les routeurs par des ronds. Par exemple, sur la figure 1.1, H et I sont des hôtes, et A et B sont des routeurs. L'interface H_α de l'hôte H est connectée à l'interface A_α du routeur A : les deux nœuds peuvent s'envoyer des paquets par ces interfaces.

Les hôtes sont les émetteurs et les destinataires des paquets : ils ne servent pas au transit des paquets. Au contraire, les routeurs ne servent qu'au transit des paquets pour les acheminer jusqu'à l'hôte auquel ils sont destinés [1]. Les paquets que H envoie à I transitent par A puis par B avant d'arriver à leur destination.

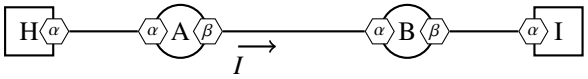


FIGURE 1.1 – Hôtes et routeurs.

Deux opérations assurent le bon acheminement des paquets dans le réseau : le routage et le transfert. Le routage est le processus de sélection des chemins dans le réseau, et le transfert est le processus d'envoi des paquets par les routeurs vers le nœud suivant. Ce nœud est un routeur, en principe plus proche de la destination du paquet, ou l'hôte destinataire.

En routage *next-hop*, ces opérations s'effectuent au niveau des routeurs : les hôtes n'y participent pas. La décision de transfert ne dépend que de l'adresse destination des paquets. Le routage configure le transfert de chaque routeur pour qu'à chaque adresse destination corresponde un *next-hop*, c'est-à-dire une paire constituée d'une interface de sortie du routeur et d'une adresse d'un voisin. Cette dernière doit être associée à une interface connectée par un lien à l'interface de sortie du *next-hop*. Par exemple, sur la figure 1.1, le

1. Dans certains cas, un même nœud peut à la fois être un hôte et un routeur.

routage configure le routeur A pour que les paquets à destination de I soient transférés à B en passant par les interfaces A_β et B_α. Si b est une adresse associée à B_α, le *next-hop* est la paire $A_\beta \rightarrow b$, que nous nous autorisons aussi à noter $A_\beta \rightarrow B_\alpha$.

1.2 Routage sensible à la source

Outre l'adresse destination des paquets, l'entête des paquets IP a d'autres champs qui peuvent être utilisés pour le transfert des paquets. Le routage sensible à la source est une extension du routage *next-hop* où la décision de transfert dépend de l'adresse source des paquets en plus de leur adresse destination.

Le routage sensible à la source a des applications concrètes. Il résout notamment le problème du routage d'une certaine catégorie de réseaux *multihomés* (section 1.3.3) où les paquets doivent être routés en fonction de leur adresse source pour ne pas être perdus. Bien qu'un routage configuré à la main à base d'ingénierie de trafic (*policy-based routing*) soit parfois utilisé, il n'existait pas, avant notre travail, de protocole de routage dynamique qui implémente complètement le routage sensible à la source.

Le déploiement d'un routage sensible à la source requiert un protocole de routage sensible à la source (section 1.4) et un transfert (section 1.5) sensible à la source. Les différences avec le routage *next-hop* classique nécessitent des modifications qui posent des difficultés tant techniques que théoriques que nous résolvons dans ce manuscrit.

Par ailleurs, le routage sensible à la source a des conséquences pour les hôtes. En effet, le routage dépend à la fois de l'adresse source et de l'adresse destination des paquets, que l'hôte choisit. L'hôte a donc une influence sur le chemin qu'empruntent les paquets dans le réseau. Dans le cas du *multihoming* que nous décrivons, le choix des adresses source et destination est particulièrement important (section 1.6).

Les contributions de ce manuscrit portent principalement sur le routage et le transfert sensible à la source. Nous fournissons notamment les outils théoriques qui permettent l'implémentation du routage sensible à la source (pour une certaine classe de protocoles de routage) et nous montrons qu'ils sont suffisants en réalisant la première implémentation complète d'un protocole de routage sensible à la source. Nous décrivons nos contributions en détail en section 1.7.

1.3 De l'hébergement simple au multihoming avec plusieurs adresses

Le *multihoming* (hébergement multiple) consiste, pour un réseau, à être connecté à plusieurs fournisseurs d'accès : le réseau est alors dit *multihomé*. Le *multihoming* présente plusieurs intérêts. L'un d'eux est la fiabilité : si la connexion à un fournisseur d'accès tombe en panne, le réseau reste connecté à l'Internet par l'autre fournisseur d'accès. Le *multihoming* permet aussi d'augmenter les performances du réseau, par exemple en répartissant la charge sur les différents FAI.

Le *multihoming* est une notion qui s'applique à tous les réseaux, du simple hôte aux FAI eux-mêmes. Par exemple, un téléphone portable est *multihomé* lorsqu'il est à la fois connecté au réseau cellulaire et à un wifi. Ou encore, un réseau frontière (de petite entreprise ou domestique) est *multihomé* s'il est connecté à plusieurs DSL.

Le *multihoming* est un problème difficile pour lequel de nombreuses solutions existent [dLB06]. Ces solutions nécessitent de faire un compromis entre coopération avec les FAI (section 1.3.2) et déploiement de nouveaux protocoles (section 1.3.3).

1.3.1 Hébergement simple

La manière la plus simple pour un réseau de se connecter à l'Internet est d'être client d'un seul FAI. Celui-ci prête au réseau un préfixe d'adresses, sous-préfixe d'un préfixe plus large appartenant au FAI. Nous

illustrons cette configuration sur la figure 1.2 : le réseau reçoit le préfixe 2001:db8:1111::/48, sous-préfixe du préfixe 2001:db8:1000::/36 du FAI.

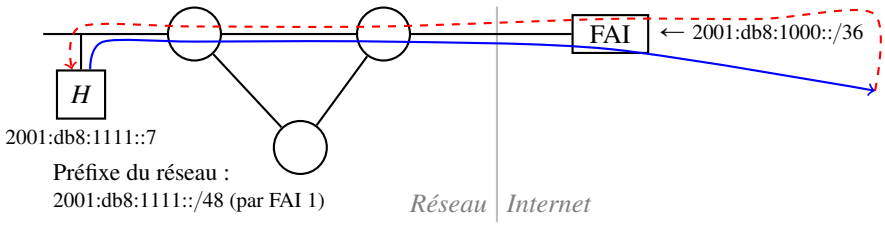


FIGURE 1.2 – Hébergement simple.

La destination du paquet est suffisante pour router les paquets. Le FAI annonce à l'Internet l'ensemble de ses adresses (dont celle du réseau) et au réseau l'ensemble des adresses de l'Internet.

Considérons un paquet provenant d'un hôte dans l'Internet à destination d'un hôte du réseau (par exemple $(dst = 2001{:}db8{:}1111{::}7, src = 2001{:}db8{:}8888{::}8)$ dont le chemin est en pointillés rouges sur notre figure). Dans l'Internet, le paquet est routé en routage *next-hop* jusqu'au FAI par la route annoncée par le FAI (2001:db8:1000::/36), puis le FAI route le paquet jusqu'au réseau (2001:db8:1111::/48), puis le réseau route le paquet jusqu'à l'hôte porteur de l'adresse destination du paquet. À l'inverse, considérons un paquet envoyé par un hôte du réseau et à destination d'un hôte dans l'Internet. Le paquet est acheminé par routage *next-hop* dans le réseau jusqu'au FAI, puis par le FAI vers sa destination dans l'Internet.

Remarquons que le réseau n'a qu'un préfixe d'adresses fourni par le FAI. Celui-ci filtre (détruit) les paquets envoyés par le réseau avec une adresse source différente des adresses fournies : ces paquets sont considérés comme faisant objet d'usurpation d'adresses [BS04].

1.3.2 Multihoming classique

En *multihoming* classique, le réseau n'a toujours qu'un seul préfixe d'adresses. Ce préfixe est annoncé à tous les FAI qui l'annoncent aussi au reste de l'Internet (figure 1.3). Les FAI annoncent donc le préfixe du réseau en plus du leur. Par exemple, dans notre figure, FAI 1 annonce à la fois 2001:db8:1000::/36 et 2001:db8:ffff::/48.

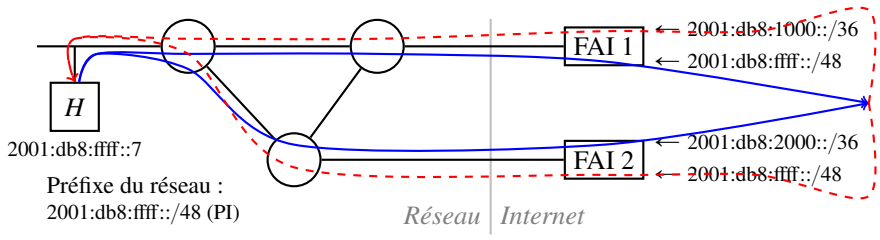


FIGURE 1.3 – *Multihoming* classique.

Les paquets entrants et sortants peuvent passer par l'un ou l'autre des FAI et les protocoles de routage *next-hop* assurent la fiabilité du réseau. Si une panne est détectée sur une route passant par un FAI, une autre route est empruntée.

Considérons comme précédemment un paquet provenant d'un hôte dans l'Internet à destination d'un hôte du réseau (par exemple $(dst = 2001{:}db8{:}ffff{::}7, src = 2001{:}db8{:}8888{::}8)$ dont le chemin est en pointillés rouges sur notre figure). Dans l'Internet, le paquet est routé en routage *next-hop* jusqu'à un FAI par la

route annoncée par le réseau aux FAI (par exemple FAI 1). Si ce FAI tombe en panne, le paquet peut passer par l'autre FAI qui annonce aussi une route vers le réseau. Nous observons un comportement similaire avec un paquet provenant du réseau et à destination de l'Internet.

Toutefois, le préfixe associé au réseau doit être indépendant des FAI (PI — *Provider Independent*) et il doit être annoncé à l'Internet : une nouvelle route est annoncée à l'Internet pour chaque réseau *multihomé* avec cette technique. Celle-ci n'est pas adaptée aux petits réseaux. En pratique, un réseau doit répondre à certaines exigences pour avoir droit à un préfixe PI [NCC11]. L'hébergement simple n'a pas cet inconvénient : un FAI n'annonce pas à l'Internet l'ensemble des préfixes de ses clients, mais seulement son large préfixe.

1.3.3 Multihoming avec plusieurs adresses

Il est possible de faire du *multihoming* comme plusieurs hébergements simples, sans coopération des FAI (figure 1.4). Ainsi, chaque FAI fournit au réseau un préfixe d'adresses et filtre les paquets en fonction de ce préfixe. Les hôtes reçoivent une adresse par fournisseur (par exemple, *H* reçoit l'adresse 2001:db8:1111::7 et l'adresse 2001:db8:2222::7).

Le *multihoming* avec plusieurs adresses n'a aucune incidence sur le nombre de préfixes annoncés à l'Internet par les FAI mais nécessite une modification du routage du réseau *multihomé* et une coopération des hôtes, comme nous allons le voir.

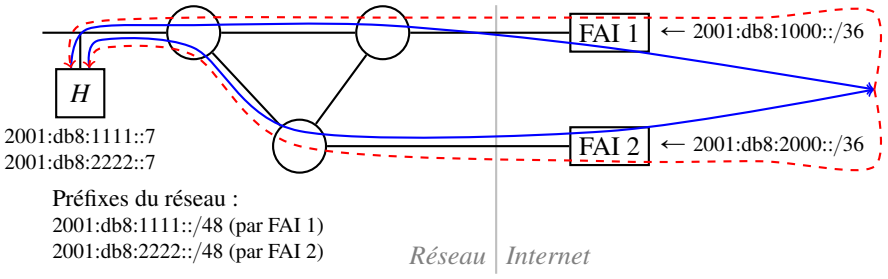


FIGURE 1.4 – Double hébergement simple.

Modification du routage Considérons un paquet envoyé par un hôte du réseau à un hôte dans l'Internet. Ce paquet a l'une des adresses de l'émetteur comme adresse source (par exemple (dst = 2001:db8:8888::8, src = 2001:db8:1111::7) dont le chemin est la flèche supérieure en bleu plein sur notre figure). Il doit aller vers le fournisseur de son adresse source (ici, FAI 1) pour ne pas être filtré par un autre fournisseur (ici, FAI 2). Le routage doit donc dépendre de l'adresse source du paquet : le routage *next-hop*, qui ne tient compte que de l'adresse destination, ne convient pas. Il y a plusieurs façons de résoudre le problème.

Une première solution consiste à connecter tous les FAI au même routeur (figure 1.5*a*). Tous les paquets à destination de l'Internet sont alors acheminés vers ce routeur avec un protocole de routage *next-hop* classique. Il suffit de configurer le routeur avec des règles d'ingénierie de trafic pour transférer les paquets en fonction de leur adresse source vers le FAI adéquat. Cette solution introduit un point unique de défaillance : si le routeur connecté aux FAI tombe en panne, le réseau est sans connectivité.

D'autres manipulations sont possibles pour contourner le problème : connecter plusieurs routeurs à tous les fournisseurs (figure 1.5*b*), ou utiliser des tunnels. La première solution impose des contraintes fortes sur la topologie physique du réseau. Au contraire, les tunnels sont des liens virtuels qui simulent une connexion directe entre deux pairs. Nous pouvons les utiliser pour connecter directement chaque hôte du réseau à chaque routeur de frontière (figure 1.5*c*) et configurer les hôtes pour envoyer les paquets dans le

bon tunnel. Si la modification des hôtes n'est pas possible, nous pouvons utiliser un procédé similaire avec tous les routeurs du réseau (figure 1.5*d*) ou, dans une moindre mesure, avec tous les routeurs de frontière (figure 1.5*e*). Toutes ces solutions ajoutent de l'état au réseau qu'il est nécessaire d'établir manuellement ou à l'aide d'un protocole de signalisation. Par ailleurs, l'encapsulation des paquets dans les tunnels a une conséquence sur la taille des paquets, et le chemin de la source jusqu'au FAI n'est pas nécessairement aussi direct que sans tunnel (le routeur le plus proche ayant un tunnel pour le bon fournisseur peut être à l'opposé de ce fournisseur).

On peut aussi penser à utiliser un paradigme de routage totalement différent : le routage par la source. En routage par la source, l'hôte émetteur d'un paquet écrit dans l'entête du paquet la liste de tous les routeurs par lesquels le paquet doit passer. Les routeurs ne choisissent pas le *next-hop* mais se contentent de suivre les directives inscrites dans l'entête du paquet. Un hôte peut donc choisir le routeur de frontière adapté au paquet. Toutefois, cette solution n'est probablement pas si simple à mettre en œuvre : il faut que les hôtes connaissent à la fois le préfixe source associé aux FAI et quel routeur de frontière est connecté à quel FAI.

Enfin, la solution que nous proposons dans ce manuscrit, le routage sensible à la source (figure 1.5*f*), est une extension compatible au routage *next-hop*. Avec le routage sensible à la source, les routes fournies par les FAI sont spécifiques au préfixe source fourni. Les paquets sont routés vers le fournisseur d'accès qui leur correspond sans autre besoin que d'étendre les protocoles de routage existants. Avec cette méthode, un seul protocole de routage est déployé dans le réseau, de manière uniforme. La configuration manuelle, si elle est nécessaire, reste confinée aux routeurs de frontière. De plus, nous verrons (sections 3.3.2) que la compatibilité avec le routage *next-hop* nous permet de garder des routeurs *next-hop* dans le réseau : il suffit en fait que les routeurs de frontière soient dans une composante sensible à la source connexe.

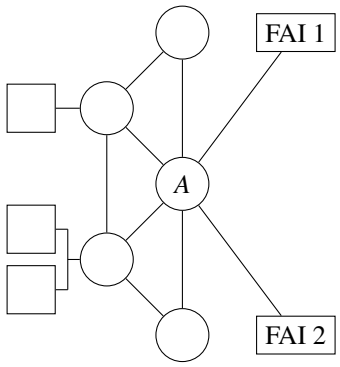
Coopération des hôtes Le *multihoming* avec plusieurs adresses nécessite aussi une modification au niveau des hôtes. Considérons un paquet provenant de l'Internet et à destination d'une adresse d'un hôte du réseau (sur la figure 1.6 : $(dst = 2001{:}db8{:}1111{::}7, src = 2001{:}db8{:}8888{::}8)$). Dans l'Internet, une seule route peut router ce paquet : celle annoncée par le fournisseur de l'adresse destination du paquet (ici, FAI 1). Si ce fournisseur tombe en panne, le paquet est perdu. À l'inverse, un paquet provenant du réseau et à destination de l'Internet passera par le fournisseur de son adresse source. Si ce fournisseur tombe en panne, le paquet est perdu.

Le routage n'assure donc pas la fiabilité du réseau. Cette responsabilité revient à l'hôte émetteur : le choix de l'adresse source du paquet coïncide avec le choix du FAI par lequel le paquet sort du réseau local, et le choix de l'adresse destination du paquet coïncide avec le choix du FAI par lequel le paquet entre dans le réseau distant. Le choix d'une paire d'adresses source et destination coïncide avec le choix d'un chemin entre les deux hôtes. Par exemple, sur la figure 1.6, *H* peut choisir quatre chemins différents :

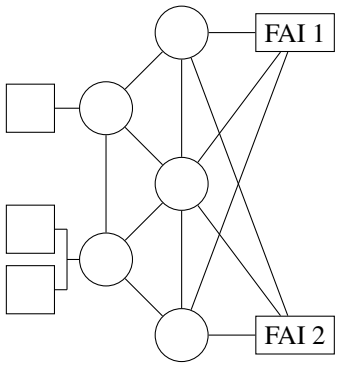
— $(dst = 2001{:}db8{:}8888{::}8, src = 2001{:}db8{:}1111{::}7)$;
— $(dst = 2001{:}db8{:}8888{::}8, src = 2001{:}db8{:}2222{::}7)$;
— $(dst = 2001{:}db8{:}9999{::}8, src = 2001{:}db8{:}1111{::}7)$;
— $(dst = 2001{:}db8{:}9999{::}8, src = 2001{:}db8{:}2222{::}7)$.

Ainsi, au moment d'établir une connexion, un hôte choisit un chemin. C'est à lui de s'assurer que le chemin est fonctionnel. C'est à lui de détecter une éventuelle panne et dans ce cas de changer de chemin pour assurer la fiabilité des connexions. Dans le cas où plusieurs chemins fonctionnent, c'est encore à lui de choisir le ou les chemins à utiliser.

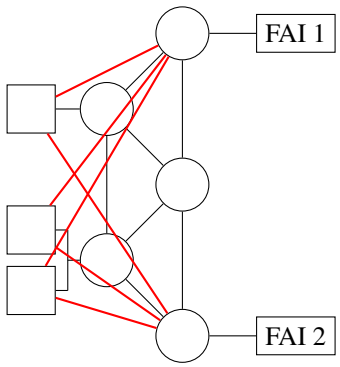
Ce rôle qu'ont les hôtes sur la fiabilité de leurs connexions peut être vu comme une contrainte, mais il s'agit aussi d'une opportunité : celle pour les hôtes d'augmenter leurs performances (section 1.6). Même si les fournisseurs ne faisaient pas de filtrage, le routage sensible à la source serait intéressant car il donne aux hôtes cette flexibilité.



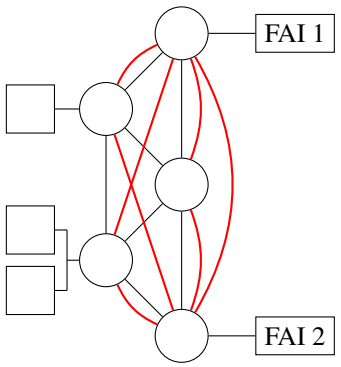
(a) Un routeur connecté à tous les FAI : A est un point unique de défaillance.



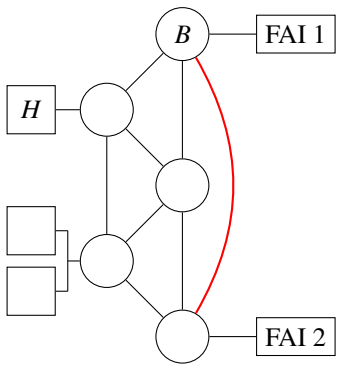
(b) Plusieurs routeurs connectés à tous les FAI.



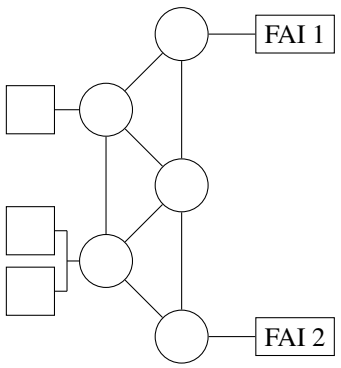
(c) Hôtes connectés aux routeurs de frontière par tunnels.



(d) Tous les routeurs sont connectés aux routeurs de frontière par des tunnels.



(e) Les routeurs de frontières ont des tunnels entre eux — un paquet de H à FAI 2 fera un détour par B où il sera encapsulé pour passer par le tunnel.



(f) Routage sensible à la source — les paquets vont directement au FAI correspondant à leur adresse source, sans encapsulation.

FIGURE 1.5 – Quelques solutions au routage des paquets dans un réseau *multihomé*.

FIGURE 1.6 – Chemins entre deux réseaux *multihomés*.

1.4 Protocole de routage sensible à la source

Un protocole de routage *next-hop* calcule des routes pour des préfixes destination : un paquet suit une route si son adresse destination est dans le préfixe associé à la route. Pour calculer les routes, un protocole de routage *next-hop* annonce et relaie un certain nombre d'informations dont le préfixe destination associé aux routes.

Similairement, un protocole de routage sensible à la source calcule des routes pour des paires de préfixes destination et source : un paquet suit une route si son adresse destination est dans le préfixe destination associé à la route et si son adresse source est dans le préfixe source associé à la route. Pour calculer ses routes, un protocole de routage sensible à la source annonce le préfixe source associé aux routes en plus des mêmes informations qu'un protocole *next-hop*.

Bien que les mécanismes fondamentaux d'un protocole sensible à la source soient les mêmes que ceux d'un protocole de routage *next-hop*, l'extension d'un protocole de routage *next-hop* au routage sensible à la source n'est pas évidente. Dans le cas des protocoles à vecteur de distances, il est nécessaire que la version de base ignore totalement les annonces sensibles à la source de l'extension : garder le préfixe destination et ignorer le préfixe source d'une annonce peut engendrer des boucles de routage persistantes.

Toutefois, la compatibilité entre le protocole de base et son extension est possible. En effet, un préfixe de longueur nulle accepte toutes les adresses. Donc, une route sensible à la source pour une paire de préfixes destination et source avec un préfixe source de longueur nulle accepte les paquets indépendamment de leur source. Une telle route accepte exactement les mêmes paquets qu'une route *next-hop* pour le même préfixe destination. L'extension sensible à la source d'un protocole à vecteur de distances assure la compatibilité avec sa version de base :

— en annonçant les routes sensibles à la source avec un préfixe source de longueur nulle par le protocole de base, en omettant le préfixe source,

— en considérant les annonces reçues par le protocole de base comme ayant un préfixe source de longueur nulle.

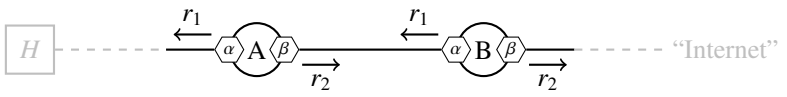
Le protocole de routage calcule des routes pour le transfert des paquets. Son but est de peupler les tables de transfert pour que les paquets suivent effectivement les routes qu'il a calculées.

1.5 Transfert sensible à la source

En routage next-hop, le transfert des paquets ne dépend que de leur adresse destination tandis qu'elle dépend aussi de leur adresse source en routage sensible à la source. Les adresses destination valables pour une route, comme les adresses source en routage sensible à la source, sont exprimées à l'aide de préfixes. Or, les préfixes peuvent se chevaucher, et deux routes peuvent router un même paquet. En routage next-hop, il est usuel d'utiliser la spécificité des préfixes pour faire un choix entre ces routes ; ce n'est plus possible en routage sensible à la source. C'est ce que nous voyons dans les paragraphes suivants.

Transfert next-hop et préfixe le plus long En routage *next-hop*, les tables de transfert associent des *next-hop* à des préfixes. Un paquet dont l'adresse destination est dans le préfixe associé à une entrée est transféré au *next-hop* correspondant. Par exemple, sur la figure 1.7, la route r_1 de la table de transfert du routeur B

indique que tous les paquets ayant leur destination dans le préfixe 2001:db8:1111::/56 sont transférés au *next-hop* $B \rightarrow A$.



	destination	*next-hop*
(r_1)	2001:db8:1111::/56	$A_\alpha \rightarrow$ *gauche*
(r_2)	::/0	$A_\beta \rightarrow B_\alpha$

(a) Table de transfert de A

	destination	*next-hop*
(r_1)	2001:db8:1111::/56	$B_\alpha \rightarrow A_\beta$
(r_2)	::/0	$B_\beta \rightarrow$ *droite*

(b) Table de transfert de B

FIGURE 1.7 – Table de transfert *next-hop* et ambiguïtés.

Une table peut contenir plusieurs entrées routant un même paquet. Dans ce cas, tous les routeurs du réseau doivent choisir la même entrée pour transférer le paquet ; autrement, des boucles de routage peuvent persister. Dans notre exemple, à la fois r_1 et r_2 routent le paquet 2001:db8:1111::7. Sur la figure 1.7, nous voyons que si A sélectionne r_2 pour router ce paquet et si B sélectionne r_1, alors le paquet oscille entre A et B.

Pour lever ces ambiguïtés, l'entrée ayant le préfixe le plus spécifique est retenue : il s'agit aussi de l'entrée ayant le préfixe le plus long — on parle communément de *longest match*. On remarque que la spécificité des préfixes induit un ordre total sur les entrées routant un même paquet : c'est pourquoi il existe toujours un "plus spécifique" (minimum) sur ces entrées. Dans notre exemple, r_1 est plus spécifique que r_2 et doit être utilisée pour transférer le paquet. Remarquons que les deux routeurs A et B peuvent être assimilés aux deux routeurs du haut de la figure 1.2 : r_1 est une route vers des hôtes du réseau et r_2 est une route pour l'Internet. Le choix de la route la plus spécifique correspond bien au comportement désiré : les paquets destinés aux hôtes suivent r_1 et les paquets à destination de l'Internet suivent r_2.

Transfert sensible à la source En routage sensible à la source, les tables de transfert associent des *next-hop* à des paires de préfixes destination et source. Un paquet dont l'adresse destination et l'adresse source sont respectivement dans le préfixe destination et le préfixe source associés à une entrée est transféré au *next-hop* correspondant. Par exemple, sur la figure 1.8, la route r_4 indique que tous les paquets ayant leur adresse source dans le préfixe 2001:db8:1111::/48 sont transférés au *next-hop* $A_\beta \rightarrow B_\alpha$.

	destination	source	*next-hop*
(r_3)	2001:db8:1111::/56	::/0	$A_\alpha \rightarrow$ *gauche*
(r_4)	::/0	2001:db8:1111::/48	$A_\beta \rightarrow B_\alpha$

FIGURE 1.8 – Table de transfert sensible à la source et ambiguïtés.

Comme pour les tables du routage *next-hop*, plusieurs entrées peuvent router un même paquet et il est nécessaire de déterminer laquelle doit être utilisée. Il serait naturel d'étendre la spécificité utilisée en routage *next-hop* aux paires de préfixes. Mais cela n'induit pas un ordre total sur les entrées routant un même préfixe et n'est donc pas un critère de choix suffisant. Dans notre exemple, à la fois r_3 et r_4 routent le paquet (dst = 2001:db8:1111::7, src = 2001:db8:1111::2) et aucune de ces deux entrées n'est plus spécifique que l'autre.

Pour lever les ambiguïtés, le consensus actuel est de sélectionner l'entrée ayant le préfixe destination le plus spécifique et en cas d'égalité l'entrée ayant le préfixe source le plus spécifique. Il s'agit de la linéarisation de la spécificité par l'ordre lexicographique $(\text{destination}, \text{source})$; cela induit un ordre total sur les entrées routant le même paquet. On parle d'ordre par la *destination d'abord*. Dans notre exemple, r_3 a un préfixe destination plus spécifique que r_4 et doit donc être utilisée pour router le paquet.

Il existe plusieurs implémentations de tables de transfert capables de faire du routage sensible à la source. Certaines l'implémentent nativement et utilisent l'ordre par la destination d'abord. D'autres nécessitent l'utilisation de règles d'ingénierie de trafic qui choisissent, en fonction de la source d'un paquet, une table de routage *next-hop* à utiliser pour transférer ce paquet. L'ordre obtenu est l'inverse de celui par la destination d'abord : cela revient à choisir l'entrée ayant la source la plus spécifique, et en cas d'égalité l'entrée ayant la destination la plus spécifique. Le protocole de routage doit pouvoir forcer les tables à suivre le comportement voulu, ce qui sera l'objet de la section 3.2.

1.6 Couches supérieures

Nous avons vu (section 1.3.3) que dans le cas du *multihoming* avec plusieurs adresses, le routage sensible à la source offre aux hôtes le choix entre plusieurs chemins. Le choix d'un chemin par un hôte coïncide avec le choix de la paire d'adresses source et destination des paquets. L'hôte, par ce choix, peut assurer la fiabilité de ses connexions et augmenter ses performances, par exemple en choisissant le chemin de plus faible latence ou en répartissant la charge sur plusieurs chemins.

Le choix des adresses des paquets émis par les hôtes se fait au niveau des couche supérieures : réseau, transport et application. Plus le choix se fait proche des couches basses (réseau), plus il aura d'impact sur l'ensemble des connexions et du trafic. Mais plus le choix se fait proche des couches supérieures (application), plus il peut répondre aux besoins spécifiques des applications. Voici quelques exemples de solutions indépendantes qui ne se combinent pas.

Couche réseau Shim6 est un protocole situé entre la couche réseau et la couche transport visant à assurer la fiabilité des connexions par la réécriture des adresses IP. Considérant qu'un chemin est une paire d'adresses source et destination, Shim6 détecte la panne d'un chemin lorsque le trafic entrant est interrompu tandis que le trafic sortant persiste (sur ce chemin). Il évalue alors la qualité de tous les chemins pour en trouver un fonctionnel. Shim6 modifie alors le chemin des paquets sortants en réécrivant leurs adresses source et destination, et fait l'opération inverse pour les paquets entrants de sorte que les couches supérieures aient l'impression d'utiliser le chemin d'origine.

Shim6 ne nécessite aucune modification des couches supérieures et assure la fiabilité de leurs connexions. Toutefois, l'optimisation des connexions est difficile ou impossible car elle nécessite de connaître le fonctionnement des protocoles de couches supérieures et ce qu'elles cherchent à optimiser.

Couche transport MPTCP est une extension compatible de TCP qui assure la fiabilité des connexions TCP et augmente leurs performances par de la répartition de charge. MPTCP ne requiert pas la modification des applications TCP déjà existantes, et répond ainsi au besoin de la plupart des applications existantes (TCP est le protocole de couche transport le plus utilisé). Après avoir établi une connexion TCP, MPTCP établit autant de sous-flots qu'il y a de chemins. Il assure la fiabilité de sa connexion en changeant dynamiquement de sous-flots si une panne est détectée, et il peut augmenter le débit de sa connexion en répartissant les données envoyées sur l'ensemble de ses sous-flots.

QUIC [TI17] est un protocole de transport en cours de standardisation basé sur UDP. Un intérêt de QUIC, par rapport à TCP, est de multiplexer les connexion. Il est possible d'établir plusieurs connexions à la TCP dans une seule connexion QUIC sans payer à chaque fois le coût de la négociation de connexion — ce qui fait gagner en latence. Il existe une version multichemin pour QUIC basée sur MPTCP [DCB17].

Couche application La plupart des applications se satisfont des services offerts par les protocoles de transport comme (MP)TCP. Mais certaines applications ont des besoins spécifiques et définissent leurs

propres mécanismes de transport qui reflètent les besoins de l'application. Ces applications utilisent UDP qui fournit un service minimal.

Il existe déjà des applications qui implémentent leurs propres mécanismes multichemin dans le cadre du *multihoming*. Par exemple, quelques logiciels de *tunneling* sont multichemin [2] pour assurer la fiabilité des connexions ou pour répartir la charge.

Un autre exemple intéressant est celui de BitTorrent, qui est multichemin par nature au sens où il crée plusieurs chemins pour une ressource. En utilisant l'adresse destination de plusieurs pairs, plusieurs chemins sont crées pour la même ressource. L'utilisation de plusieurs adresses source [84715] augmente encore le nombre de chemins pour une ressource et assure à la fois la fiabilité de l'accès aux ressources et la répartition de charge sur les différents fournisseurs.

1.7 Contributions

Dans ce manuscrit, nous apportons une solution complète aux problèmes du routage et du transfert sensibles à la source. Nous nous intéressons aussi à la question de la sélection de chemin à la couche application. Les contributions concernent les chapitres 3 à 5.

1.7.1 Routage

Nous fournissons les outils théoriques nécessaires à la réalisation d'un protocole de routage sensible à la source et à l'extension compatible d'un protocole de routage à vecteur de distances existant au routage sensible à la source. Nous montrons que ce que nous proposons est suffisant en réalisant la première implémentation complète du routage sensible à la source.

Contributions théoriques La principale contribution de ce manuscrit concerne la levée des ambiguïtés dans les tables de transfert (section 3.2). Nous définissons un formalisme pour exprimer les ambiguïtés des tables de transfert. Nous définissons une notion de complétude de table de transfert et montrons qu'une table complète n'est pas ambiguë. Enfin, sur cette base, nous définissons un algorithme de levée d'ambiguïté incrémental et sans mémoire, adapté aux protocoles de routage. Cet algorithme permet à un protocole de routage de peupler des tables de transfert qui n'ordonnent pas leurs entrées comme le protocole de routage.

Une autre contribution théorique concerne les protocoles de routage à vecteur de distances (section 3.3). Nous avons montré comment étendre ces protocoles au routage sensible à la source, et cela de manière compatible avec la version de base. Nous montrons que les alternatives naturelles à ce que nous proposons induisent des boucles de routage persistantes et nous en expliquons les raisons.

Contributions techniques Pour vérifier que nos solutions sont bien suffisantes, nous avons étendu un protocole à vecteur de distances au routage sensible à la source (chapitre 4). À notre connaissance, il s'agit de la toute première implémentation complète du routage sensible à la source. Pour cela, nous avons conçu le format de paquet de l'extension (en nous appuyant sur nos conclusions théoriques) et nous avons implémenté et documenté [BC14, BC18] le comportement de l'extension. Nous avons choisi le protocole Babel [Chr11, CS17] pour notre implémentation.

Notre implémentation utilise notre algorithme de levée d'ambiguïté défini en section 3.2. Par là, nous le validons et montrons qu'il est implémentable dans un protocole de routage. Il permet à notre implémentation d'être utilisée sur des systèmes ne comportant pas de tables de transfert sensibles à la source natives.

2. https ://github.com/cloudwu/mptun ; http ://mlvpn.fr

Évaluation expérimentale Nous avons mis en place un réseau expérimental *multihomé* utilisant le routage sensible à la source. Nous avons eu des cas d'usage à la fois en IPv4 et en IPv6. Nous avons testé le bon fonctionnement de ce réseau avec des applications standard tels que *ping*, *traceroute* ou *OpenVPN*. Nous avons aussi testé notre réseau avec *MPTCP* et montré qu'il était compatible avec le routage sensible à la source (section 4.3).

1.7.2 Couches supérieures

Les applications qui définissent leurs propres mécanismes de couche transport ont des besoins spécifiques : les mécanismes utilisés sont très différents. Nous contribuons à la recherche de mécanismes multichemin adaptables aux applications interactives et légères (chapitre 5). Nous mettons en évidence l'implémentabilité de ces mécanismes à la couche application.

Contribution théorique Nous proposons un algorithme asynchrone d'envoi de sondes et de paquets intégrant deux mécanismes d'optimisation de la connexion. Le premier mécanisme accélère la convergence lors de pannes, le second optimise la connexion en cas de taux de pertes importants sur une partie ou tous les chemins.

Contribution technique Comme nous l'avons dit, les applications utilisant leurs propres mécanismes de couche transport ont généralement des besoins spécifiques. Notre recherche s'effectue dans le cadre d'une application particulière : mosh. Il n'est pas naturellement possible d'étendre le format de paquet de mosh : nous avons donc conçu notre format de paquet, à partir de celui de mosh. Nous avons implémenté nos mécanismes et les avons intégré à mosh. La plupart de nos changements sont confinés dans une sous-couche du protocole et notre implémentation est compilée dans une bibliothèque de transport potentiellement réutilisable dans d'autres applications. (Cette compartimentation en bibliothèque était déjà présente.)

Évaluation expérimentale Nous avons réalisé un grand nombre de tests, en émulation, comparant la version originale de mosh et notre version multichemin (section 5.6). Nous avons conçu et réalisé plusieurs scénarios de tests pour mesurer le gain et le coût des différentes optimisations. Nous mesurons principalement le temps de reconvergence après une panne et la résistance aux pertes de paquets. Nous comparons mosh à mpmosh (multipath mosh), notre version multichemin de mosh, à divers niveaux d'optimisation.

Chapitre 2

Contexte : principes du routage et relation avec les couches supérieures

Dans l'Internet, deux applications distantes s'échangent des données à travers un réseau à commutation de paquets. Une donnée envoyée est fragmentée en petits paquets de taille bornée qui sont étiquetés avec l'adresse du destinataire, ou adresse destination, et l'adresse de l'émetteur, ou adresse source. L'adresse destination est utilisée par les routeurs du réseau pour acheminer les paquets jusqu'à sa destination, et l'adresse source est utilisée par le destinataire pour répondre à l'émetteur du paquet.

2.1 Routage *next-hop* et couches supérieures

On parle communément de routage pour désigner deux opérations assurant le bon acheminement des paquets à leur destination. Celle, locale, de *transfert* (*forwarding*) des paquets par un routeur, et celle, globale, de *routage* à proprement parler.

Le *transfert* est l'opération réalisée par un routeur pour faire suivre les paquets vers un autre routeur, en principe plus proche de leur destination. C'est une opération locale au routeur, qui ne nécessite aucune connaissance du réseau. Les données utilisées pour transférer un paquet sont extraites du paquet lui-même et de structures de données construites auparavant.

Le *routage* est le processus de sélection des chemins qu'empruntent les paquets pour aller à leur destination. Le routage est une opération globale en cela qu'elle dépend, pour chaque route, de tous les routeurs traversés par la route. Le routage peut être fait manuellement par un administrateur, on parle alors de *routage statique*, ou automatiquement par un programme implémentant un *protocole de routage*, on parle alors de *routage dynamique*.

En routage *next-hop*, seule l'adresse destination est extraite d'un paquet lors de son transfert. Le routeur possède une table de transfert qui associe des *next-hop* aux adresses destination des paquets (section 2.1.2) et que l'on appelle FIB (*Forwarding Information Base*). Le but du routage est de peupler les tables de transfert pour assurer le bon acheminement des paquets à leur destination. Le routage est entièrement géré par les routeurs, limitant le rôle des hôtes dans l'acheminement des paquets à la sélection de l'adresse destination et au choix du premier routeur. Dans le cas le plus simple, l'hôte destination n'a qu'une seule adresse et l'hôte émetteur n'est connecté qu'à un seul routeur : l'acheminement des paquets ne dépend alors que des routeurs.

2.1.1 Généralités

Nous voyons dans cette section comment représenter les réseaux. Nous voyons aussi les principaux protocoles Internet et leur agencement.

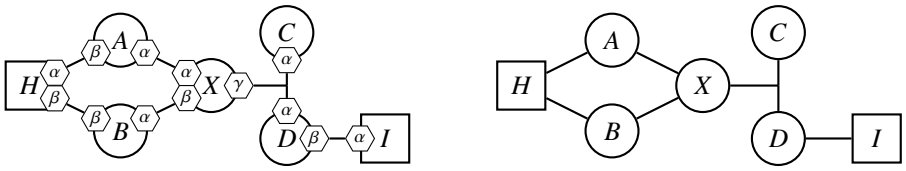
2.1.1.1 Représentation d'un réseau

Un réseau est composé de plusieurs éléments :
— deux types de nœuds, les hôtes et les routeurs ;
— les liens qui connectent ces nœuds ;
— les interfaces par lesquels les nœuds sont connectés aux liens.

Les hôtes sont représentés par des carrés et les routeurs par des cercles. Nous représentons l'interface d'un nœud par un hexagone empiétant sur le nœud. Enfin, les liens sont les arêtes qui connectent plusieurs interfaces. Par exemple, sur la figure 2.1, H est un hôte dont l'interface H_α est directement connectée à l'interface A_β du routeur A. Ces deux nœuds peuvent communiquer directement.

Lorsqu'une arête connecte plus de deux interfaces, cela signifie qu'un message envoyé par un nœud sur une interface liée à cette arête est potentiellement reçu par toutes les autres interfaces liées à cette arête. Ainsi, X, C et D peuvent communiquer directement, mais un message envoyé par X à C (via son interface X_γ) arrive aussi à D qui, n'étant pas destinataire, l'ignore.

Les interfaces complexifient parfois inutilement la représentation du réseau. Dans la suite de ce manuscrit, nous nous autorisons à masquer une partie ou la totalité des interfaces d'un nœud lorsque cela ne nuit pas à la compréhension du problème décrit. La vue simplifiée de la figure 2.1 représente le même réseau que la vue avec interfaces explicites.



Interfaces explicites Vue simplifiée

FIGURE 2.1 – Représentation d'un réseau.

2.1.1.2 Protocoles Internet

Les protocoles Internet sont le plus souvent classés suivant un modèle à cinq couches. La couche *physique* (1) fournit la communication sur un lien physique ; la couche *lien* (2) fournit la communication entre deux nœuds adjacents ; la couche *réseau* (3) fournit la communication entre deux nœuds du réseau. La couche réseau est particulièrement importante : tous les nœuds de l'Internet utilisent le même protocole de couche réseau (IP) ; on parle de *couche de convergence*. La couche *transport* (4) fournit la communication entre deux applications ; et enfin la couche *Application* (7) fournit à l'utilisateur le service demandé.

Dans ce manuscrit, nous nous intéressons à la couche réseau et aux couches dites *supérieures* (transport et application). Le routage des paquets est un problème de couche réseau : les adresses destination des paquets sont définies par le protocole IP. Les couches transport et application définissent des mécanismes qui dépendent de la couche réseau. Aux sections 2.2.1.1 et 5, nous voyons que le routage sensible à la source ouvre aux couches supérieures la possibilité de faire du multichemin sans changement de la couche réseau.

Le Protocole IP Le protocole de couche réseau utilisé dans l'Internet est IP *(Internet Protocol)*. Il fonctionne suivant le principe de la commutation de paquets : il transporte les segments de données transmis

par les couches supérieures sous forme de *paquets IP*. Chaque paquet IP est indépendant des autres. Il est constitué de quelques octets formant un *entête IP*, suivi des données qu'il contient (figure 2.2). L'entête IP est constitué de plusieurs informations, dont notamment les adresses de l'hôte émetteur et de l'hôte destinataire : on parle respectivement *d'adresse source* et *d'adresse destination*. Ces adresses ont un format défini par le protocole IP : elles sont appelées adresses IP.

FIGURE 2.2 – Un paquet IP, avec l'entête de couche réseau (IP).

IP fournit un service *non connecté*, *non fiable* et *non ordonné*. Il est non connecté : un paquet est envoyé directement sur le réseau, et le paquet en lui-même contient les informations suffisantes pour atteindre sa destination. Il est non fiable : un paquet peut être perdu, sans que l'émetteur n'en soit averti. Il est non ordonné : les paquets peuvent arriver dans un ordre différent de l'ordre d'origine.

Certaines applications ont besoin d'une communication fiable, connectée ou ordonnée. Par exemple, il est important à la fin du téléchargement d'un fichier que les données du fichier soient dans l'ordre (service ordonné) et que le fichier soit complet (service fiable). L'application peut aussi vouloir être notifiée de la perte de connexion (service connecté) : le téléchargement peut alors être annulé proprement, par exemple en supprimant le fichier. C'est aux couches supérieures (transport et application) d'assurer ces services.

Couche transport Généralement, plusieurs applications s'exécutent sur un même nœud. La couche *transport* permet aux applications de communiquer entre elles. Elle utilise la couche réseau pour transmettre les informations de l'application d'un nœud à l'autre, puis s'occupe de les transmettre à la bonne application. Les deux protocoles de couche transport les plus utilisés dans l'Internet sont TCP et UDP.

Ces deux protocoles utilisent un *numéro de port* pour identifier l'application à laquelle délivrer les données [1]. Chaque application réserve un numéro de port pour pouvoir recevoir des données, et utilise le numéro de port de l'application distante pour lui en envoyer. Comme illustré en figure 2.3, ces protocoles ajoutent un entête de couche transport qui précède les données de l'application et contient les ports source et destination.

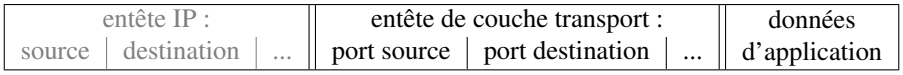


FIGURE 2.3 – Un paquet IP, avec les entêtes de couche transport et réseau.

TCP est un protocole de transport connecté, fiable et ordonné. Lorsqu'une application utilise le service fourni par TCP, elle doit d'abord attendre que TCP établisse la connexion : TCP vérifie que le pair distant existe et qu'il accepte la connexion. Si le pair distant ne répond plus, la connexion est coupée, et le service n'est plus utilisable : il faut se reconnecter. Une fois la connexion établie, TCP simule un flot d'octets entre les deux applications, s'occupant de la réémission des paquets perdus (les paquets sont numérotés) et de leur réordonnancement. Une application typique de TCP est le transfert de fichiers entre deux pairs : l'émetteur envoie séquentiellement les octets du fichier à TCP, et le destinataire écrit de même séquentiellement les données reçues de TCP.

UDP est un protocole de transport qui assure un service minimal : l'acheminement des données vers une application spécifique. Il est donc, à l'image du service IP qu'il utilise, non connecté, non fiable, et non ordonné. Cette flexibilité laissée à l'application lui laisse utiliser ses propres mécanismes, qui parfois

1. En réalité, il n'y a pas de correlation directe entre numéro de port et application : une application peut avoir plusieurs ports ouverts et un même port peut être partagé avec plusieurs applications (plus rare). Cette représentation est néanmoins pratique et correspond souvent à la réalité, au moins côté serveur : par exemple, le port 80 est normalement lié au serveur http de l'hôte.

donnent lieu à de meilleures performances. Une application typique utilisant UDP est la voix sur IP, où il est préférable de perdre un petit fragment de voix que de bloquer la conversation le temps d'une réémission.

Couche application Les protocoles que nous avons vus jusqu'à présent sont génériques au sens où ils sont adaptés à des besoins généraux. Certes, chaque protocole a ses avantages et inconvénients et nous avons vu qu'ils ne répondaient pas aux mêmes besoins, mais de nombreuses applications très variées utilisent l'un ou l'autre de ces protocoles.

Les protocoles de couche application répondent au besoin d'une application spécifique. Par exemple, un navigateur Web permet à son utilisateur d'accéder au contenu de serveurs Web, tandis qu'un client de messagerie permet à son utilisateur d'envoyer et de recevoir du courrier électronique. Bien sûr, une même application peut implémenter plusieurs protocoles de couche application et ainsi remplir plusieurs fonctionnalités. Une application peut à la fois contenir un navigateur, qui communique avec des serveurs Web, et une messagerie, qui communique avec des serveurs de courrier électronique.

Enfin, le déploiement d'un protocole de couche application est plus facile que le déploiement d'un protocole de couche transport. Les protocoles de couche transport (et inférieures) sont souvent implémentés dans le noyau (changer de protocole revient à changer de noyau), ou même dans le matériel. D'autre part, déployer de nouveaux protocoles de couche transport est difficile, car certains routeurs filtrent (détruisent) les paquets dont le protocole de couche transport est inconnu. En pratique, ces routeurs sont suffisamment nombreux pour qu'il soit quasiment impossible de déployer un nouveau protocole de couche transport. Au contraire, les protocoles de couche application peuvent être mis à jour sans impacter les applications existantes ni les noyaux et ils ne sont pas filtrés (ou du moins plus difficilement filtrables). Une application peut notamment utiliser UDP pour déployer ses propres mécanismes de transport.

2.1.2 Transfert en routage *next-hop*

Le routage *next-hop* est un paradigme de routage où le transfert d'un paquet par un routeur ne dépend que de l'adresse destination du paquet et de la FIB du routeur. L'adresse destination du paquet n'est pas modifiée. Une propriété importante de ce paradigme est qu'aucun nœud ne détermine le *chemin* complet du paquet dans le réseau mais chaque nœud détermine seulement le prochain saut : le *next-hop*.

Dans cette section, nous commençons par nous étendre sur cette propriété, ensuite nous définissons le *next-hop*, puis nous montrons comment s'effectue son choix. Nous terminons par montrer que le routage *next-hop* n'empêche pas les pathologies comme les boucles de routage ou les trous noirs.

Choix restreint au *next-hop* En routage *next-hop*, un nœud ne détermine pas le chemin d'un paquet dans le réseau mais seulement son prochain saut. Par exemple, sur la figure 2.4, un paquet qui va de H à I a quatre chemins possibles :

— (H, A, C, I),
— (H, A, B, C, I),
— (H, B, C, I),
— (H, B, A, C, I).

Mais H n'a que deux choix : envoyer le paquet à A (par son interface H_α) ou à B (par son interface H_β). Une fois ce choix effectué, il n'a plus aucun contrôle sur le paquet. Si A reçoit le paquet, il a trois choix, indépendants de celui effectué par H : le renvoyer à H, le transférer à B ou le transférer à C. Renvoyer le paquet d'où il vient est possible mais incorrect : nous parlons des pathologies à la fin de la section.

D'un point de vue des hôtes, et plus précisément des couches supérieures à la couche réseau, le contrôle sur le chemin est *presque* entièrement laissé au réseau (c'est-à-dire aux routeurs). Dans certains cas, une légère flexibilité leur est laissée : le choix de l'adresse destination, et celui de l'interface de sortie.

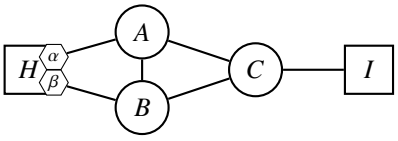
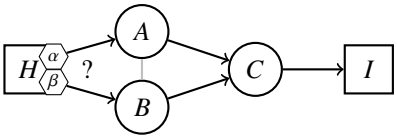


FIGURE 2.4 – Transfert *next-hop*.

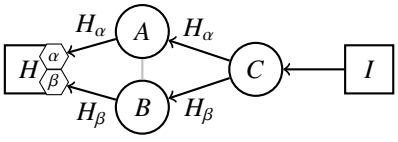
Considérons par exemple la figure 2.5*a*. Nous avons représenté un choix possible de *next-hop* pour l'adresse de I : par exemple, un paquet à destination de l'adresse de I est transféré par A à C. Le choix du premier saut par H a une influence sur l'ensemble du chemin emprunté par les paquets à destination de I. La portée de cette influence dépend du choix des *next-hop* par les routeurs : si B choisit A comme *next-hop*, les deux chemins possibles presque les mêmes.

À l'inverse, l'hôte I n'a pas le choix de son interface pour envoyer des paquets à H : il n'en a qu'une. Mais puisque H a deux adresses (une par interface), I peut choisir l'une ou l'autre pour le contacter. Nous avons partiellement [2] représenté sur la figure 2.5*b* un choix possible de *next-hop* pour les destinations des deux adresses de H : un paquet à destination de l'adresse affectée à l'interface H_α est transféré par C en A. Le choix de l'adresse destination a des conséquences similaires au choix de l'interface de sortie.

Il est possible de tirer un avantage de cette opportunité de multichemin. Par exemple, certaines techniques [WY12] tentent d'établir simultanément une connexion pour chaque adresse du pair distant. La première connexion réussie est sélectionnée. Nous voyons en section 3.1 que le transfert en routage sensible à la source offre un comportement similaire avec le choix de l'adresse source, et nous rentrons dans les détails liés au multichemin au chapitre 5.



(a) Un choix possible de routes vers I. Le choix du premier saut par l'hôte H détermine le chemin de ses paquets pour I.



(b) Un choix possible de routes de I à H. Le choix de l'adresse destination par I détermine le chemin de ses paquets pour H.

FIGURE 2.5 – Flexibilité des hôtes en routage *next-hop*.

Le *next-hop* Le *next-hop* désigne, pour un paquet et un routeur donnés :

— l'interface de sortie du routeur utilisée pour transférer le paquet ;

— l'adresse d'une interface du nœud auquel le paquet est transféré.

Par exemple, sur la figure 2.6, le lien central est relié à A par deux interfaces et à B par une interface. Pour aller de A à B, un paquet peut donc sortir de A par l'interface A_α ou par l'interface A_β et entrer dans B par l'interface B_α. Le nœud A a donc deux *next-hop* possibles pour transférer des paquets en B : soit $(A_\alpha \rightarrow$ 2001:db8::b), soit $(A_\beta \rightarrow$ 2001:db8::b). À l'inverse, B n'a qu'une seule interface de sortie pour atteindre A mais A a deux interfaces d'entrée. Le nœud B a donc aussi deux *next-hops* possibles : soit $(B_\alpha \rightarrow$ 2001:db8::a1), soit $(B_\alpha \rightarrow$ 2001:db8::a2).

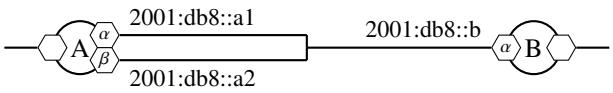


FIGURE 2.6 – *Next-hop* et interfaces.

2. Nous n'avons pas représenté les *next-hop* choisis par A et B pour les destinations respectives H_β et H_α.

L'écriture complète du *next-hop* est parfois lourde et nécessite d'ajouter sur les figures des informations qui ne sont pas utiles au problème décrit, comme par exemple l'ensemble de chaque adresse attribuées à chaque interface de chaque nœud du réseau. Nous nous autorisons donc quelques simplifications d'écritures qui sont sans ambiguïté :

— l'adresse du *next-hop* peut être remplacée par l'interface qui lui est associée : $(B_\alpha \rightarrow$ 2001:db8::a1$)$ peut s'écrire $(B_\alpha \rightarrow A_\alpha)$;
— s'il n'y a qu'une seule interface de sortie possible entre deux routeurs, nous pouvons l'omettre : $(B_\alpha \rightarrow A_\alpha)$ peut s'écrire $(B \rightarrow A_\alpha)$;
— s'il n'y a qu'une seule adresse possible pour le *next-hop*, nous pouvons simplement nommer le nœud : $(A_\alpha \rightarrow B_\alpha)$ peut s'écrire $(A_\alpha \rightarrow B)$.

Table de transfert Pour choisir le *next-hop* des paquets, chaque routeur a pour FIB une base de données appelée *table de transfert* (*forwarding table*). Cette table associe des *next-hop* à des destinations et se représente comme un ensemble d'entrées (D, NH) où D est une destination et NH un *next-hop*. Lorsqu'un routeur reçoit un paquet, il cherche dans sa table de transfert l'entrée qui a pour clef la destination du paquet, et en déduit le *next-hop* correspondant. On dit alors que l'entrée *route* le paquet. Par extension, si un nœud a une table de transfert avec une entrée qui route un paquet, on dit que la table de transfert et le nœud *routent* ce paquet.

Par exemple, supposons que l'interface du nœud I de la figure 2.4 a pour adresse 2001:db8::1. Nous considérons que A route vers C les paquets pour I. Le routeur A possède donc l'entrée ci-dessous (gauche) dans sa table de transfert.

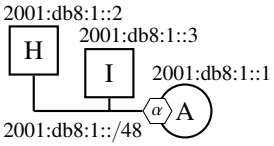
destination	*next-hop*
2001:db8::1	$A_\alpha \rightarrow C$

Lorsque la destination finale du paquet est aussi le *next-hop*, l'adresse du *next-hop* est la même que celle du paquet. Il n'est donc pas nécessaire de la préciser dans la table de transfert : l'interface utilisée pour transférer le paquet suffit. Considérons par exemple le réseau de la figure 2.7*a* et la table de transfert associée au routeur A sur la figure 2.7*b*. Nous observons que l'adresse du *next-hop* est exactement la même que l'adresse destination des paquets. Sur la table de transfert représentée en 2.7*c*, nous omettons l'adresse du *next-hop* mais indiquons qu'il faut se référer à l'adresse destination du paquet. Remarquons alors que toutes les entrées ont le même *next-hop*.

Agrégation En pratique, il n'est pas désirable ni même le plus souvent possible d'avoir des tables de transfert avec une entrée par adresse pour des questions de mémoire et de performances. L'Internet étant structuré en sous-réseaux, il est naturel de vouloir compacter les tables de transfert en associant les next-hop aux préfixes associés aux sous-réseaux. On parle *d'agrégation* lorsque plusieurs adresses, voire plusieurs préfixes, apparaissent comme un seul préfixe.

Par exemple, sur la figure 2.7, le préfixe 2001:db8:1::/48 est affecté au lien. Il est donc possible d'agréger les trois adresses de ce réseau (celles de H, I et A) dans le préfixe affecté au lien (figure 2.7*d*). L'adresse du *next-hop* se réfère toujours à l'adresse destination du paquet : la table est donc vraiment identique à sa version naïve. Un autre exemple d'agrégation est celui de la figure 2.8 : un routeur B est connecté à deux sous-réseaux inclus dans 2001:db8:100::/40. Le routeur A peut agréger ces sous-réseaux dans ce seul préfixe (deuxième entrée de sa table).

Le cas extrême de l'agrégation se trouve dans les routeurs de frontière qui reçoivent leur connectivité à l'Internet par une seule route. Tout l'Internet s'agrège alors dans la route de préfixe de longueur nulle (::/0 pour IPv6), appelée *route par défaut*.



(a) Réseau

destination	*next-hop*
2001:db8:1::2	$A_\alpha \rightarrow$ 2001:db8:1::2
2001:db8:1::3	$A_\alpha \rightarrow$ 2001:db8:1::3

(b) Table naïve de A

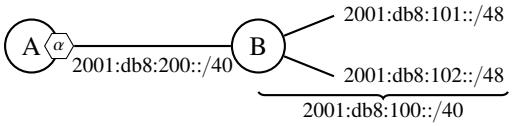
destination	*next-hop*
2001:db8:1::2	$A_\alpha \rightarrow$ *dest*
2001:db8:1::3	$A_\alpha \rightarrow$ *dest*

(c) Table naïve de A
l'adresse du next-hop *est déduite de la destination*

destination	*next-hop*
2001:db8:1::/48	$A_\alpha \rightarrow$ *dest*

(d) Table agrégée de A

FIGURE 2.7 – Dernier next-hop et agrégation d'adresses sur un même lien.



(a) Réseau

destination	*next-hop*
2001:db8:200::/40	$A_\alpha \rightarrow$ *dest*
2001:db8:100::/40	$A_\alpha \rightarrow B$

(b) Table agrégée de A

FIGURE 2.8 – Agrégation de préfixes d'un même sous-réseaux.

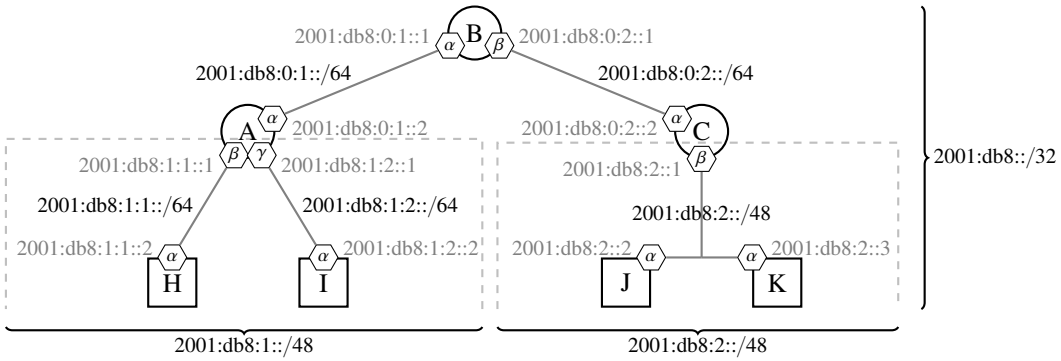
Au contraire, une entrée de la table de transfert qui ne correspond qu'à une seule adresse est appelée *route d'hôte*. Au sens d'une table agrégée, il s'agit encore d'un préfixe. Puisque nous avons défini les préfixes comme des ensembles d'adresses, une route d'hôte est un singleton. Par exemple, une route d'hôte pour la destination 2001:db8::1 a pour clef le préfixe 2001:db8::1/128.

Dans le cas d'une table agrégée, on dit qu'une entrée *route* un paquet si l'adresse destination du paquet *appartient* au préfixe correspondant à l'entrée. Formellement, une entrée (D, NH) route un paquet (d, s) si $d \in D$ — avec (d, s) la paire d'adresses destination et source du paquet.

Exemple complet Nous avons représenté sur la figure 2.9*a* un réseau adressé dans 2001:db8::/32 composé de deux sous-réseaux : l'un adressé dans 2001:db8:1::/48, l'autre dans 2001:db8:2::/48. À chaque lien est associé un préfixe (en noir) qui appartient à son sous-réseau. Chaque interface de chaque hôte reçoit une adresse dans le préfixe du lien auquel elle est connectée (en gris).

Le routeur *B* est à la jonction des deux sous-réseaux. Nous avons représenté ce que serait sa table de transfert sans agrégation (2.9*b*) : il y a une entrée par adresse, soit 9 entrées, dont beaucoup ont le même *next-hop*. Sa table agrégée (2.9*c*) ne comporte plus que quatre préfixes : deux pour les liens auxquels il est directement connecté, et deux pour les deux sous-réseaux. La table agrégée de *B* ne dépend pas de la complexité des sous-réseaux : nous pouvons rajouter des liens, des routeurs ou des hôtes dans l'un ou l'autre des sous-réseaux sans l'affecter.

Le routeur *B* transfère tout paquet à destination du sous-réseau de gauche (2001:db8:1::/48) à *A*. C'est ensuite à *A* de transférer les paquets au sein du sous-réseau. Pour cela, *A* possède plusieurs entrées dans sa table de transfert. Remarquons aussi que nous y avons ajouté une route par défaut (::/0) qui agrège tout ce qui est au-delà de *B*, y compris le sous-réseau de droite et le lien entre *B* et *C*.



(a) Vue du réseau

destination	*next-hop*
2001:db8:0:1::2	$B_\alpha \rightarrow$ *dest*
2001:db8:0:2::2	$B_\beta \rightarrow$ *dest*
2001:db8:1:1::1	$B_\alpha \rightarrow$ 2001:db8:0:1::2
2001:db8:1:1::2	$B_\alpha \rightarrow$ 2001:db8:0:1::2
2001:db8:1:2::1	$B_\alpha \rightarrow$ 2001:db8:0:1::2
2001:db8:1:2::2	$B_\alpha \rightarrow$ 2001:db8:0:1::2
2001:db8:2::1	$B_\beta \rightarrow$ 2001:db8:0:2::2
2001:db8:2::2	$B_\beta \rightarrow$ 2001:db8:0:2::2
2001:db8:2::3	$B_\beta \rightarrow$ 2001:db8:0:2::2

(b) Table naïve de B

destination	*next-hop*
2001:db8:0:1::/64	$B_\alpha \rightarrow$ *dest*
2001:db8:0:2::/64	$B_\beta \rightarrow$ *dest*
2001:db8:1::/48	$B_\alpha \rightarrow$ 2001:db8:0:1::2
2001:db8:2::/48	$B_\beta \rightarrow$ 2001:db8:0:2::2

(c) Table agrégée de B

destination	*next-hop*
2001:db8:0:1::/64	$A_\alpha \rightarrow$ *dest*
2001:db8:1:1::/64	$A_\beta \rightarrow$ *dest*
2001:db8:1:2::/64	$A_\gamma \rightarrow$ *dest*
::/0	$A_\alpha \rightarrow$ 2001:db8:0:1::1

(d) Table agrégée de A

destination	*next-hop*
2001:db8:0:2::/64	$C_\alpha \rightarrow$ *dest*
2001:db8:2::/48	$C_\beta \rightarrow$ *dest*
::/0	$C_\alpha \rightarrow$ 2001:db8:0:2::1

(e) Table agrégée de C

FIGURE 2.9 – Un réseau et ses tables de transfert.

Levée d'ambiguïtés En général, une table a plusieurs préfixes non disjoints parmi ses entrées. Dans ce cas, ces entrées sont susceptibles de router un même paquet : un choix doit être fait. Considérons par exemple le routeur *C* de l'exemple précédent (figure 2.9) : il a une route par défaut ::/0 et une autre route vers son sous-réseau 2001:db8:2::/48. Un paquet à destination de *J* (2001:db8:2::2) est routé par chacune des deux routes. Il semble ici évident que le paquet doit suivre la première route : s'il suit la route par défaut, il sort du sous-réseau.

La règle du préfixe le plus long (*longest prefix match*) est utilisée pour lever les ambiguïtés : parmi les entrées susceptible de router un paquet, celle qui a le préfixe le plus long est retenue — il s'agit aussi du préfixe le plus spécifique. Dans notre exemple, les deux préfixes ::/0 et 2001:db8:2::/48 contiennent l'adresse 2001:db8:2::2, mais le deuxième est long de 48 bits alors que le premier est long de 0 bit : l'entrée correspondant au deuxième préfixe est sélectionnée. La règle du préfixe le plus long repose sur le fait qu'il existe un ordre total sur les préfixes contenant une même adresse.

Ordre sur les préfixes Rappelons que les préfixes sont des ensembles d'adresses. L'inclusion est donc naturellement définie sur les préfixes : un préfixe est inclus dans un autre si toutes les adresses du premier appartiennent au second.

Or, la spécificité sur les préfixes coïncide avec l'inclusion : un préfixe inclus dans un autre est plus spécifique que celui-ci. Par exemple, le préfixe de longueur nulle ::/0, l'ensemble de toutes les adresses, contient clairement tout autre préfixe. C'est donc bien le préfixe le moins spécifique. En particulier, il inclut le préfixe 2001:db8:1::/48 de l'exemple du paragraphe précédent.

En fait, les préfixes sont soit disjoints, soit l'un inclus dans l'autre. (Ils forment même un arbre binaire dont la racine est le préfixe de longueur nulle.) L'inclusion induit un ordre total $\leqslant$ sur les préfixes contenant une même adresse, que nous appelons *ordre de spécificité*. Nous considérons que le préfixe le plus spécifique est le minimum : si ρ_1 et ρ_2 sont deux préfixes, alors $\rho_1 \subset \rho_2$ est équivalent à $\rho_1 \leqslant \rho_2$. Cette notion d'inférieur correspond bien à l'idée qu'il y a moins d'éléments (adresses) dans le préfixe le plus spécifique ; mais elle n'est pas à confondre avec la longueur du préfixe. En effet, un préfixe long est plus spécifique qu'un préfixe court (ou disjoint) : par exemple, $32 \geqslant 0$ et 2001:db8::/32 $\subset$ ::/0.

Table de transfert Linux La figure 2.10 montre des extraits d'une table de transfert sous Linux, obtenue avec la commande `ip`. Les lignes 1 et 5 demandent l'affichage des tables de transfert respectivement IPv4 et IPv6. Le préfixe destination est indiqué en premier ; il est suivi par d'autres informations, dont l'interface à utiliser, précédée du mot-clef *dev*, et de l'adresse IP du *next-hop*, précédée du mot-clef *via*.

L'entrée de la ligne 2 est celle d'une route par défaut IPv4 (ayant pour préfixe destination 0.0.0.0/0) dont le *next-hop* a pour adresse 172.23.47.254, accessible sur le lien connecté à l'interface *eth0*. L'entrée de la ligne 3 n'indique pas d'adresse IP du *next-hop*, mais seulement l'interface vers laquelle transférer le paquet : tous les nœuds dans le préfixe destination spécifié (172.23.32.0/20) sont directement accessibles par le lien connecté à l'interface *eth0*. Il n'y a donc pas à passer par un routeur, et l'adresse IP du *next-hop* est celle de l'adresse destination du paquet, d'où l'absence du mot-clef *via*. Les deux autres lignes (4 et 6) sont deux autres illustrations. La première a un préfixe destination restreint à une unique adresse, et la deuxième une route par défaut IPv6. On retrouve dans les deux cas les mêmes constructions que pour la première entrée (ligne 2).

Pathologies Chaque routeur a la responsabilité d'envoyer un paquet à un *next-hop* en fonction de sa table de transfert. Il ne tient pas compte de la table de transfert des autres nœuds du réseau et ne vérifie pas la validité des chemins empruntés. Des tables de transfert mal configurées peuvent faire apparaître des pathologies plus ou moins graves.

```
# ip route show
default via 172.23.47.254 dev eth0
172.23.32.0/20 dev eth0  proto kernel scope link src 172.23.36.45
192.168.4.49 via 192.168.4.39 dev eth1 proto babel onlink
# ip -6 route show
default via fe80::8618:8803:5474:9b01 dev eth0 proto zebra metric 3 pref medium
```

FIGURE 2.10 – Extraits d'une table de transfert Linux.

La plus grave pathologie est celle de la *boucle de routage (routing loop)* : les tables de transfert des routeurs sont telles que certains paquets passent indéfiniment par les mêmes routeurs, en bouclant. Cette pathologie est particulièrement grave car, d'une part, les paquets pris dans la boucle n'arrivent pas à destination et, d'autre part, ils congestionnent le réseau et ralentissent ainsi les paquets qui ne sont pas pris dans la boucle. Remarquons tout de même qu'un paquet IP ne boucle pas éternellement : il possède un champ appelé TTL (*Time To Live*, ou *Hop Limit* en IPv6) qui borne le nombre de routeurs par lequel il peut passer. À chaque saut, le TTL est décrémenté et s'il atteint 0, le paquet est détruit. La valeur initiale recommandée [3] est de 64.

Par exemple, la figure 2.9 possède une boucle de routage entre les nœuds *A* et *B*. En effet, *B* envoie tous les paquets contenus dans 2001:db8:1::/48 à *A*, et *A* envoie tous les paquets vers *B* à l'exception des trois préfixes 2001:db8:0:1::/64, 2001:db8:1:1::/64 et 2001:db8:1:2::/64. Un paquet à destination de 2001:db8:1::1 est routé jusqu'à *A* puis de *A* à *B* et de *B* à *A* jusqu'à ce qu'il expire.

Une autre pathologie possible est celle des *trous noirs (black holes)* : des paquets sont attirés vers un routeur qui, faute de pouvoir les router, les détruit. La plupart des trous noirs existent naturellement du fait que toutes les adresses d'un préfixe ne sont pas utilisées. Par exemple, le routeur *C* de la figure 2.9 envoie tous les paquets contenus dans 2001:db8:2::/48 sur un lien dont seulement trois adresses sont utilisées. Un paquet à destination de 2001:db8:2::4 est routé jusqu'à *C*, puis il est détruit.

Les exemples de pathologies que nous avons donnés sont liés à l'agrégation. Il est en particulier possible de résoudre le premier problème, critique, en ajoutant dans la table de transfert de *A* un trou noir artificiel : une entrée à destination de 2001:db8:1::/48 et sans *next-hop* — ces *next-hop* sont dit inaccessibles (*unreachable*). Cette entrée est plus spécifique que ::/0, ce qui empêche les paquets à destination du sous-réseau d'en sortir, et moins spécifique que les autres entrées, ce qui permet un bon routage des paquets.

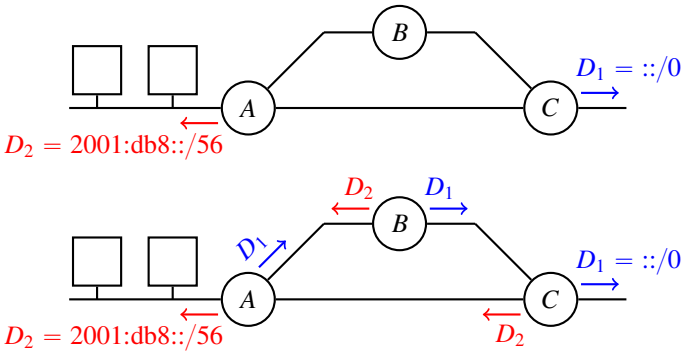
Toutefois, ce genre de pathologies peut aussi arriver à cause d'une mauvaise configuration du réseau, ce qui est l'objet de la section 2.1.3.

2.1.3 Protocoles de routage

Le routage du réseau consiste à peupler les tables de transfert du réseau pour que toute destination accessible par un routeur soit accessible à tout le réseau. Par exemple, considérons la figure 2.11 où *A* possède une route pour le préfixe 2001:db8::/56 et *C* une route pour le préfixe ::/0. Une possibilité de routage est représentée sur la figure du dessous, où chaque routeur du réseau a une entrée pour chaque préfixe.

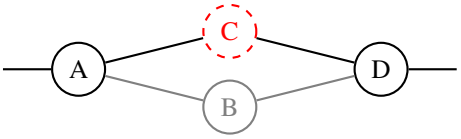
Dans cette section, nous voyons que le routage peut être automatisé par un protocole de routage. Le protocole de routage est capable de trouver les meilleurs chemins dans le réseau pour une métrique donnée. Nous voyons enfin comment fonctionnent les protocoles à vecteur de distances.

3. https ://www.iana.org/assignments/ip-parameters/ip-parameters.xml.



Avant routage, ci-dessus, A est directement au connecté au lien auquel est affecté D_2, et C est un routeur de frontière qui a une route par défaut. Deux routes sont à redistribuer au réseau. Ci-dessous est représenté un routage possible du réseau.

FIGURE 2.11 – Redistribution



Si le trafic passe par C et que celui-ci tombe en panne, il faut attendre l'intervention d'un administrateur dans le cas de routage statique, là où un protocole de routage dynamique reconverge automatiquement par B.

FIGURE 2.12 – Panne dans un réseau.

2.1.3.1 Routage statique et routage dynamique

Le routage statique est la configuration manuelle des tables de transfert du réseau, par un administrateur. Il a des limites importantes. Premièrement, il ne passe pas à l'échelle : administrer un grand réseau est difficile et sujet à l'erreur, surtout s'il est administré par plusieurs administrateurs différents. Deuxièmement, le routage statique n'est pas résilient aux pannes : si un routeur tombe en panne, il faut attendre l'intervention de l'administrateur pour retrouver la connectivité, même si un autre chemin est possible, comme illustré en figure 2.12.

Le routage dynamique résout ces problèmes. L'administrateur choisit un protocole de routage à utiliser sur le réseau, et installe un programme implémentant ce protocole sur chaque routeur du réseau. Le protocole décrit comment les programmes communiquent, quelles informations ils s'échangent, et comment établir et choisir les routes.

2.1.3.2 Métriques

Le but d'un protocole de routage n'est pas seulement de trouver un chemin pour chaque destination du réseau, mais de trouver un "bon" voire le "meilleur" chemin. Cependant, plusieurs notions de meilleur chemin existent : passer par le moins de routeurs, avoir la meilleure latence, limiter les pertes de paquets, ou encore maximiser le débit. Les protocoles de routage étiquettent les arêtes du réseau par des coûts, et définissent une algèbre [DG13] qui permet d'évaluer et de comparer le coût d'un chemin. Le coût du chemin calculé par le protocole est appelé *métrique*.

La métrique classique est la métrique du *plus court chemin (shortest path)*, aussi appelée *nombre de sauts (hop count)*. Chaque arête a un coût identique de 1, et la métrique associée à un chemin est la somme des coûts des arêtes. Cette métrique, très simple, fonctionne bien lorsque tous les liens du réseau ont les mêmes caractéristiques, comme par exemple un réseau câblé avec la même technologie.

Compter le nombre de sauts n'est pas adapté à certains réseaux où passer par plus de routeurs peut donner de meilleurs résultats. C'est le cas des réseaux sans fil, où les liens avec des nœuds éloignés ont de

très forts taux de pertes : faire plusieurs sauts intermédiaires peut diminuer le taux de pertes total du chemin et augmenter ainsi les performances globales. *ETX (Expected Transmission Count)* [DCABM05] est une métrique qui évalue le nombre de retransmissions nécessaires pour qu'un paquet soit transmis d'un nœud à l'autre et acquitté. Elle se base sur les taux de pertes entre les deux nœuds dans les deux sens, et donne de bons résultats dans les réseaux sans fil. D'autres paramètres peuvent être pris en compte pour encore augmenter la qualité des chemins empruntés. Aussi de nombreuses autres métriques ont été expérimentées pour les réseaux sans fil [DPZ04, YWK05, Chr14], mais trouver la meilleure métrique possible dépend du réseau et reste un sujet de recherche ouvert.

Certains réseaux ont, entre deux nœuds, un même nombre de sauts et des liens sans perte mais avec des latences très différentes. C'est le cas de réseaux superposés (*overlay* — cf. section 2.2.1.2) reliant des sites géographiquement éloignés. Par exemple, un Lille-Paris-Marseille a trois sauts, autant qu'un Lille-Tokyo-Marseille, mais la latence du second est sensiblement plus importante que celle du premier. Une métrique dépendante de la latence [JBC14] est particulièrement adaptée à ce cas de figure.

De nombreuses autres métriques existent. Elles considèrent divers aspects du réseau : le débit, la latence, le taux de pertes, la charge, le MTU, l'énergie, etc. EIGRP [SNM+16] est un exemple de protocole où de nombreux paramètres sont utilisables dans le calcul de la métrique. Cependant, la plupart des protocoles n'estiment pas le coût des liens dynamiquement. Celui-ci est renseigné statiquement par l'administrateur du réseau.

2.1.3.3 Protocoles à vecteur de distances

Il existe deux principales familles de protocoles de routage : les protocoles à état de liens et les protocoles à vecteur de distances. Les protocoles à vecteur de distances ont pour RIB une table de routage avec une colonne de plus : la métrique associée à la route. Le calcul de la RIB repose sur l'algorithme à vecteur de distances, souvent appelé Bellman-Ford distribué [BG92]. L'algorithme à vecteur de distances se découpe en plusieurs sous-algorithmes : l'initialisation, la transmission et la réception.

Notations Nous définissons la RIB comme un dictionnaire de triplets (D, m, n) indexé par D avec D, m et n qui sont respectivement un préfixe destination, une métrique et un *next-hop*. Nous écrivons :

- $D \in RIB$ pour tester l'existence d'une entrée indexée par D ;
- $(m, n) \leftarrow RIB[D]$ pour retrouver la métrique et le *next-hop* associés à D ;
- $RIB[D] \leftarrow (m, n)$ pour ajouter ou remplacer le triplet (D, m, n) ;
- $RIB[D] \leftarrow \bot$ pour supprimer l'entrée indexée par D ;
- $(D, m, n) \in RIB$ pour itérer sur les entrées de la RIB.

Nous définissons la FIB de manière analogue comme un dictionnaire de paires (D, n) indexé par D avec D et n qui sont respectivement un préfixe destination et un *next-hop*.

Initialisation

— La RIB contient chaque entrée redistribuée avec une métrique nulle (le *next-hop* n'est pas nécessaire) ;

```
1 RIB ← {}
2 for all (D, _) ∈ FIB
3 |  RIB[D] ← (0, ⊥)
```

Transmission

— Régulièrement, chaque entrée de la RIB (sans le *next-hop*) est annoncée à tous ses voisins ;

```
1 for all (D, m, _) ∈ RIB
2 |  send(D, m)
```

Réception d'une annonce

Soit (D, m) le préfixe et la métrique reçus du voisin n par un lien de coût c :

— la métrique de la route depuis le nœud local est calculée ;

```
1 m' = c + m
```

— si la route est de métrique finie et si la RIB n'a pas d'entrée meilleure pour ce préfixe, l'entrée est ajoutée à la RIB avec n comme *next-hop* ;

```
1 (m_RIB, n_RIB) ← RIB[D]
2 if m' < ∞ and (D ∉ RIB or m' < m_RIB or n = n_RIB)
3 |   RIB[D] ← (m', n)
4 |   FIB[D] ← (n)
```

— si la route n'est pas de métrique finie et qu'une entrée pour ce préfixe destination est dans la RIB avec n pour *next-hop*, alors la perte de la route est propagée et l'entrée de la RIB est supprimée ;

```
1 (m_RIB, n_RIB) ← RIB[D]
2 if m' ⩾ ∞ and D ∈ RIB and n = n_RIB)
3 |   send(D, ∞)
4 |   FIB[D] ← ⊥
5 |   RIB[D] ← ⊥
```

Perte de voisin Si la couche lien indique la perte d'un lien connecté à un voisin [4] n, ou si aucun message n'est reçu d'un voisin n depuis un certain temps (*timeout*), le voisin est déclaré perdu.
Dans ce cas, toutes les entrées qui ont n pour *next-hop* sont supprimées de la RIB et des rétractions sont envoyées pour les préfixes associés à ces entrées.

```
1 for all (D, _, n) ∈ RIB
2 |   if n_perdu = n
3 |   |   send(D, ∞)
4 |   |   FIB[D] ← ⊥
5 |   |   RIB[D] ← ⊥
```

La figure 2.13 représente l'exécution de l'algorithme à vecteur de distances pour le préfixe destination ::/0 redistribué par A. Initialement, seul A connaît ::/0. Il annonce ::/0 avec une métrique de 0 à B, qui ajoute le coût du lien à la métrique qu'il reçoit pour obtenir une nouvelle métrique de 1. Puisqu'il n'a pas de route pour ::/0, il choisit $B \rightarrow A$ pour *next-hop* et ajoute l'entrée $(::/0, 1, B \rightarrow A)$ dans sa RIB, avant de l'annoncer à ses voisins. Le routeur A ignore l'annonce de B car la métrique de l'annonce $(1 + 1)$ est supérieure à celle de la route qu'il a déjà dans sa RIB (0), tandis que C et D ajoutent une entrée pour cette route. Enfin, quand D annonce sa route et que C reçoit l'annonce, C a une entrée $(::/0, 4, C \rightarrow B)$ dans sa RIB, mais comme la route proposée par D a une métrique de $(2 + 1)$, C choisit de passer par D. Le réseau est à présent stable et les meilleurs chemins ont été calculés. Lorsqu'un nœud reçoit une annonce, il a une entrée dans sa RIB pour le préfixe associé à cette annonce avec une métrique meilleure (ou égale) à celle de l'annonce. La figure 2.14 représente de manière détaillée l'état de chaque RIB du réseau.

Supposons maintenant que le lien entre A et B tombe en panne et continuons de nous intéresser au préfixe ::/0 (figure 2.15). Lorsque B détecte la panne, il rétracte sa route (il annonce à ses voisins qu'il l'a perdue) et la supprime de sa RIB. Il est alors possible que tous les nœuds reçoivent l'annonce de B, fassent de même et ainsi de suite à travers tout le réseau, mais il est aussi possible que deux annonces se croisent. Ici, B reçoit l'annonce de D, et ajoute donc l'entrée $(::/0, 3, B \rightarrow D)$ à sa RIB. Comme nous le voyons, à chaque annonce, les métriques du réseau vont en augmentant, mais une boucle de routage persiste entre B et D.

4. La notion de voisinage rejoint celle de *next-hop* : nous avons vu qu'un nœud peut être plusieurs fois voisin d'un autre nœud (figure 2.6).

La méthode naïve de résolution des boucles est d'attendre que la métrique atteigne l'infini. Toutes les annonces sont alors rejetées et la route est supprimée. Il existe un certain nombre de techniques pour prévenir une partie ou la totalité des boucles. Nous verrons celles utilisées dans le protocole Babel au chapitre 4.

2.2 Routage sensible à la source

Le *routage sensible à la source* est une extension compatible du routage *next-hop* où l'adresse source des paquets est prise en compte lors du transfert. Il s'applique principalement à certains réseaux *multihomés*.

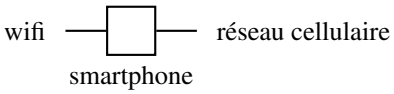
2.2.1 Applications

Le routage sensible à la source trouve sa principale application dans certains types de réseaux *multihomés*. Nous l'avons aussi (et d'abord) utilisé pour une autre application : celle de router les paquets vers un routeur de sortie dans un réseau superposé (*overlay*) distribué sur plusieurs sites à travers l'Internet.

2.2.1.1 Application aux réseaux multihomés

Le *multihoming* consiste à être connecté à l'Internet par plusieurs fournisseurs d'accès. La connexion d'un réseau à plusieurs fournisseur d'accès peut être motivée par plusieurs facteurs : le besoin de plus de fiabilité (résister à la panne d'un fournisseur), de plus de performances (par exemple par de la répartition de charge) ou pour optimiser les coûts financiers (utiliser préférentiellement le fournisseur le moins cher).

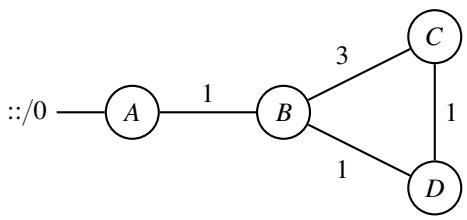
Le cas dégénéré où un réseau est réduit à un hôte est l'exemple le plus simple de *multihoming*, et probablement aussi le plus répandu. C'est en effet le cas des téléphones mobiles qui peuvent être connectés en même temps à un réseau wifi et au réseau cellulaire, comme l'illustre la figure ci-dessous.



En général, le réseau cellulaire est plus accessible mais offre de moins bonnes performances que le wifi : une limite en téléchargement, un débit inférieur et une latence plus élevée. Il est donc intéressant pour le téléphone d'être *multihomé* afin de bénéficier des avantages de l'un et de l'autre : utiliser le wifi lorsqu'il est disponible, et le réseau cellulaire autrement.

Dans le cas plus général des réseaux *multihomés*, plusieurs techniques existent [dLB06] et témoignent de la difficulté de faire du *multihoming*. Deux problèmes rendent le *multihoming* difficile : celui du routage, que nous abordons dans cette section, et celui de la capacité du réseau à utiliser plusieurs fournisseurs pour en tirer un bénéfice, que nous aborderons en section 2.3.

Nous verrons deux types de *multihoming* : le *multihoming* classique et le *multihoming* avec plusieurs adresses. Le routage sensible à la source est une solution élégante au routage des réseaux utilisant ce dernier.



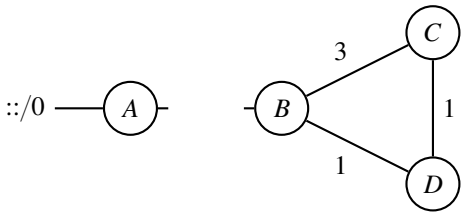
A	B	C	D
0, **FIB**	$\bot$	$\bot$	$\bot$
0, **FIB**	$1, B \to A$	$\bot$	$\bot$
0, **FIB**	$1, B \to A$	$4, C \to B$	$2, D \to B$
0, **FIB**	$1, B \to A$	$3, C \to D$	$2, D \to B$

Calcul des métriques et *next-hop* pour le préfixe $::/0$ redistribué par A.

FIGURE 2.13 – Calcul des RIB du réseau avec l'algorithme à vecteur de distances.

Itération	Nœud	destination	métrique	*next-hop*
1	A	::/0	0	**FIB**
	B		0	
	C		0	
	D		0	
2	A	::/0	0	**FIB**
	B	::/0	1	$B \rightarrow A$
	C		0	
	D		0	
3	A	::/0	0	**FIB**
	B	::/0	1	$B \rightarrow A$
	C	::/0	4	$C \rightarrow B$
	D	::/0	2	$C \rightarrow B$
4	A	::/0	0	**FIB**
	B	::/0	1	$B \rightarrow A$
	C	::/0	4	$C \rightarrow D$
	D	::/0	3	$D \rightarrow B$

FIGURE 2.14 – RIB complètes de tous les routeurs pendant la convergence.



A	*B*	*C*	*D*
0, **FIB**	$1, B \rightarrow A$	$3, C \rightarrow D$	$2, D \rightarrow B$
0, **FIB**	$\perp$	$3, C \rightarrow D$	$2, D \rightarrow B$
0, **FIB**	$3, B \rightarrow D$	$3, C \rightarrow D$	$4, D \rightarrow B$
0, **FIB**	$5, B \rightarrow A$	$5, C \rightarrow D$	$2, D \rightarrow B$
...	...	...	...

FIGURE 2.15 – Boucle de routage avec l'algorithme à vecteur de distances.

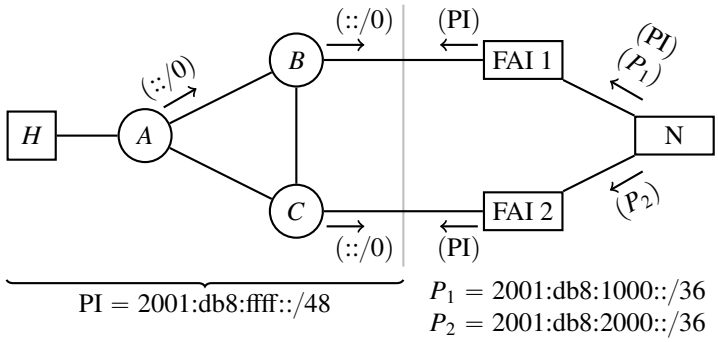
Multihoming classique d'un réseau Pour rendre possible le *multihoming*, un réseau peut acquérir un préfixe d'adresses indépendant (PI — *Provider Independent*) de tous les autres FAI. Le réseau annonce ce préfixe à tout FAI auquel il est connecté par le protocole BGP pour qu'il soit installé dans la table de routage globale. La figure 2.16 illustre ce type de réseaux *multihomé* : chaque FAI annonce à la fois son préfixe (P_1 ou P_2) et le préfixe du réseau PI.

Le routage ne pose pas de problème particulier. Chaque FAI fournit au réseau une route pour l'Internet et accepte tous les paquets en provenance du réseau. Ces routes ne se différencient que par leur métrique. Un protocole de routage *next-hop* suffit pour router les paquets du réseau : le FAI le plus proche (au sens de la métrique du protocole de routage) est choisi pour tous les paquets sortant du réseau. Sur la figure 2.16, nous avons choisi *B* comme *next-hop* de *A*.

La fiabilité est assurée naturellement par la nature dynamique du protocole de routage. Si une panne survient au niveau d'un FAI, le protocole de routage contournera la panne et enverra tous les paquets par un autre FAI, aussi bien pour les paquets sortants que pour les paquets entrants. La figure 2.16 illustre un cas de panne du lien entre *B* et FAI 1.

Un autre avantage du *multihoming* classique est qu'il n'est pas nécessaire de renuméroter les nœuds (changer leurs adresses) lors d'un changement de FAI, ce qui simplifie l'administration du réseau.

Mais le *multihoming* classique, en dépit de ses avantages, a un inconvénient de taille : le préfixe PI ne peut pas être agrégé par les fournisseurs d'accès. Par exemple, sur la figure 2.16, le plus petit préfixe agrégeant à la fois le préfixe PI (2001:db8:ffff::/48) et le préfixe de FAI 1 (2001:db8:1000::/36) est 2001:db8::/32, qui inclue aussi le préfixe de FAI 2 (2001:db8:2000::/36). En conséquence, le préfixe PI doit être annoncé à l'Internet pour être ajouté à la table de routage globale. Pour cette raison, il est maintenant



Chaque routeur de frontière du réseau a une route par défaut fournie par les FAI. Le protocole de routage choisit l'une ou l'autre. Chaque FAI annonce à l'ensemble de l'Internet son propre préfixe et le préfixe du réseau.

En cas de panne (ci-contre entre B et FAI 1), les protocoles de routage next-hop *convergent vers l'autre lien possible (entre C et FAI 2), à la fois du côté du réseau que du côté des fournisseurs d'accès.*

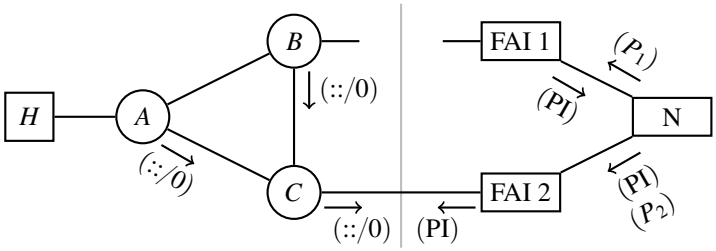


FIGURE 2.16 – *Multihoming* classique.

difficile d'obtenir un préfixe PI. Ce privilège est laissé à des réseaux d'une certaine taille qui justifient de leurs besoins [NCC11]. Par ailleurs, le *multihoming* classique nécessite la coopération des FAI. Nous voyons que si cette approche est probablement raisonnable pour les grands réseaux, elle semble impossible à généraliser aux réseaux de petites entreprises, et d'autant moins aux réseaux domestiques.

Multihoming avec plusieurs adresses Lorsqu'un petit réseau se connecte à l'Internet, par exemple un réseau domestique ou un réseau de petite entreprise, il choisit un fournisseur d'accès qui lui fournit une route pour l'Internet et un préfixe d'adresses. Ce préfixe est un sous préfixe du préfixe du FAI, de sorte que le FAI peut avoir plusieurs clients sans répercutions sur la table de routage globale. Les réseaux qui veulent être *multihomés* mais qui ne peuvent pas obtenir un préfixe d'adresses PI doivent utiliser les préfixes fournis par les FAI.

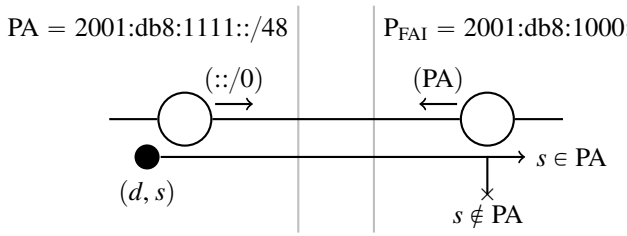
Sur la figure 2.17*b*, les réseaux R_2 et R_3 ont un seul fournisseur d'accès : FAI 1. Le réseau R_1 a deux fournisseurs : FAI 1 et FAI 2. Tous les préfixes fournis par FAI 1 sont agrégés dans le préfixe P_1 qui seul est annoncé à l'Internet.

Une bonne pratique pour un FAI consiste à filtrer (détruire) les paquets provenant de ses réseaux et qui ont une adresse source différente de celles fournies [BS04] (figure 2.17*a*). Pour qu'un hôte du réseau *multihomé* puisse envoyer un paquet via un FAI, il doit donc disposer d'une adresse fournie par ce FAI. D'autre part, pour que les paquets arrivent à leur destination, ils doivent passer par le FAI qui leur a fourni leur adresse source.

Or, toutes les routes fournies par les FAI ont comme destination le préfixe de longueur nulle. Le routage *next-hop* ne distinguant pas ces routes, il n'est pas adapté à cette situation : tous les paquets vont vers le même FAI et ceux dont l'adresse source n'est pas fournie par celui-ci sont détruits. Ici, il faut router les paquets en fonction de leur adresse source pour qu'un paquet aille vers le fournisseur de son adresse source.

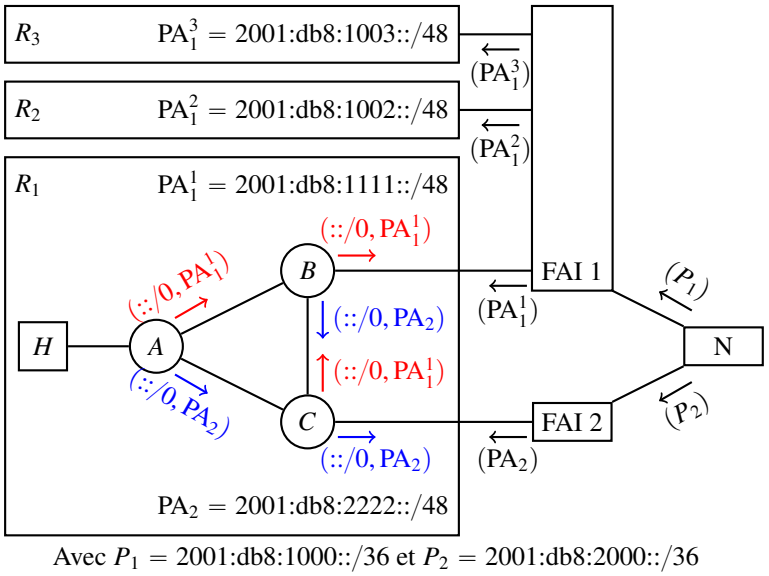
Le routage sensible à la source (section 2.2.3) répond précisément à ce cas d'usage : une route peut être associée à un préfixe source en plus du préfixe destination. En fait, une route fournie par un FAI est *spécifique* au préfixe d'adresses fourni par ce même FAI (figure 2.17*b*).

Le routage assure la fiabilité du réseau au sens où chaque nœud dispose toujours d'une route pour



Un paquet entrant dans le réseau d'un FAI depuis un de ses clients est détruit si son adresse source n'est pas fournie par le FAI.

(a) Filtrage entrant — ingress filtering.

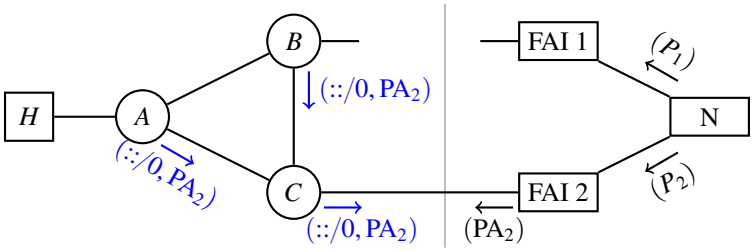


Avec $P_1 = 2001{:}db8{:}1000{::}/36$ et $P_2 = 2001{:}db8{:}2000{::}/36$

FAI 1 a trois clients à qui il fournit un sous-préfixe de son préfixe P_1. Seul P_1, qui agrège tous ces sous-préfixes, est annoncé à l'Internet.
FAI 1 filtre les paquets venant de ses clients comme indiqué figure a. Un paquet avec son adresse source dans PA_2 en provenance de R_1 est détruit : cette adresse a été fournie par un autre FAI. La route fournie par FAI 1 est en fait spécifique à PA_1^1 ; nous la notons $(::/0, \mathrm{PA}_1^1)$.

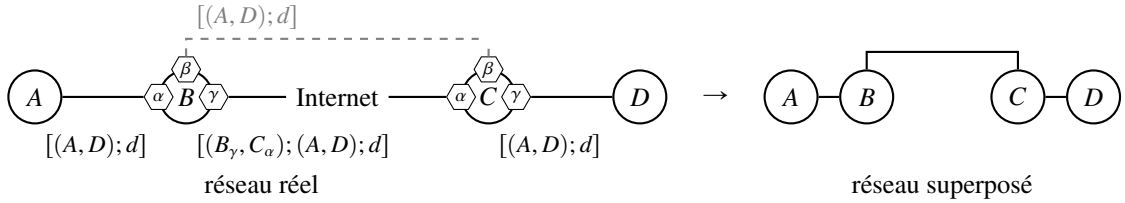
(b) Réseau hébergé indépendamment par deux fournisseurs.

En cas de panne (ci-contre entre B et FAI 1), la route est perdue. Le routage fournit toujours aux hôtes l'autre route : ce sera à eux d'assurer la fiabilité de leurs connexions (voir section 2.3).



(c) Panne d'un des fournisseurs.

FIGURE 2.17 – *Multihoming* avec plusieurs adresses.



destination	*next-hop*
Réseau superposé gauche	$B_\alpha \rightarrow A$
Réseau superposé droit	$B_\beta \rightarrow C_\beta$
par défaut	$B_\gamma \rightarrow$ “Internet”

FIGURE 2.18 – Réseau superposé et table de transfert de B.

l’Internet, mais il ne ne suffit pas pour assurer la fiabilité des connexions. En section 2.3, nous voyons que les hôtes perdent leur connexions TCP et que des mécanismes supplémentaires sont requis au niveau des hôtes pour assurer la fiabilité.

2.2.1.2 Application aux réseaux superposés

Dans certains cas, nous aimerions avoir une connexion directe entre des réseaux éloignés mais qui pourtant sont interconnectés par l’intermédiaire d’autres réseaux. Ça peut être le cas, par exemple, de réseaux d’une même entreprise présente sur plusieurs sites, ou d’un réseau et un serveur hébergé par un tiers. D’une manière générale, il n’est pas envisageable de “tirer un câble” entre les deux sites.

Un certain nombre de techniques d’encapsulation existent pour simuler un lien entre deux nœuds : ce lien est appelé *tunnel*. En particulier, les VPN (*Virtual Private Networks*) sont des tunnels sécurisés entre deux pairs : en plus d’être encapsulés, les paquets passant par le VPN sont chiffrés.

Réseaux superposés Un réseau superposé (*overlay network*) est un réseau construit au-dessus d’un autre. Les réseaux superposés qui nous intéressent sont des réseaux IP dont certains liens sont virtuels. Ces liens virtuels peuvent être obtenus à l’aide de tunnels connectant deux nœuds ; chaque nœud possède une interface virtuelle qui correspond à une extrémité du tunnel. Un paquet p entrant dans le tunnel à une extrémité est encapsulé dans un paquet IP p' qui circule sur le réseau réel jusqu’à sa destination : l’autre nœud. Là, le nœud décapsule le paquet p' et réceptionne le paquet p qu’il contient comme s’il venait de l’interface correspondant à l’autre extrémité du tunnel.

Par exemple, considérons la figure 2.18. Quatre nœuds A, B, C et D sont représentés, séparés par des liens réels (traits pleins) et un tunnel (pointillés) ; B et C sont séparés par l’Internet. Ici, A envoie un paquet IP contenant les données d à destination de D, d’entête $(dst = D, src = A)$. Arrivé au routeur B, le paquet est transféré à C par le tunnel : il est encapsulé dans un paquet envoyé par B à destination de C, d’entête (B_γ, C_α) et contenant tout le paquet envoyé par A ; ce paquet est envoyé via l’Internet. Le paquet IP envoyé par B arrive en C (par son interface α), qui se reconnaît destinataire et décapsule donc le paquet d’entête (B_γ, C_α) ; il détecte qu’il s’agit d’un paquet venant du tunnel, et route le paquet initial vers sa destination D. Au final, d’un point de vue du paquet envoyé par A, seuls les routeurs nœuds A, B, C et D ont été parcourus : le réseau superposé correspond au réseau représenté à droite de la figure 2.18. Il n’a traversé que trois liens : celui entre A et D, le tunnel, et celui entre C et D.

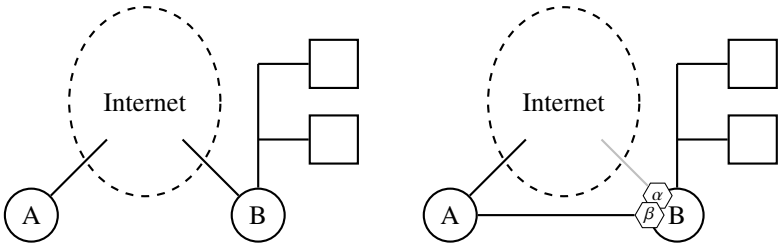
D’un point de vue du routage, les entrées de la table du routeur B ressemblent à la table suivante. Le réseau superposé gauche contient A, et le droit contient C et D. Les nœuds B et C sont des routeurs de frontière, et ont des adresses B_γ et C_α pour communiquer avec les nœuds de l’Internet qui n’appartiennent pas à ces deux réseaux. Notre paquet $[(A, D)]$ à destination de D est routé dans le tunnel par l’interface β.

Le tunnel encapsule ce paquet dans un nouveau paquet $[(B_\gamma, C_\alpha); (A, D)]$, lui-même routé par B. L'adresse de C accessible par l'Internet est spécifiée, et le paquet suit la route par défaut. Une fois décapsulé en C, le paquet $[(A, D)]$ est routé par C comme un paquet normal arrivant de son interface β.

Connexion à l'Internet Dans nombre de déploiements de tunnels et de VPN, le réseau superposé utilise la route par défaut du réseau physique pour atteindre l'Internet, tandis que seuls quelques préfixes plus spécifiques sont annoncés par le tunnel. Toutefois, dans certains déploiements, une route par défaut est aussi annoncée par le tunnel. L'extrémité du tunnel a alors deux routes par défaut : celle annoncée par le tunnel, et celle directement connectée à l'Internet, utilisée par les paquets encapsulés constituant le tunnel.

Fait naïvement, il arrive dans ces déploiements que tous les paquets à destination de l'Internet suivent la route passant par le tunnel, y compris les paquets encapsulés constituant le tunnel. Le tunnel rentre ainsi dans lui-même jusqu'à ce qu'il s'effondre.

Plusieurs contournements sont possibles en fonction des situations. Le plus simple est d'utiliser une route d'hôte (correspondant à une seule adresse destination) à destination de l'hôte situé à l'autre bout du tunnel. Cette route est on ne peut plus spécifique et donc préférée à toute autre route pour cette destination lors du transfert. Les paquets encapsulés empruntent nécessairement cette route et ne rentrent pas dans le tunnel.



(a) Réseau physique *(b) Réseau superposé*

Les deux sous-réseaux physiques contenant A et B sont adressés respectivement dans 2001:db8:01::/48 *et* 2001:db8:02::/48*, le réseau superposé est adressé dans* 2001:db8:10::/48*.*

	destination	source	*next-hop*
(r_1)	::/0	2001:db8:02::/48	$B_\alpha \rightarrow$ "Internet"
(r_2)	::/0	2001:db8:10::/48	B_β(VPN) $\rightarrow A$
	...	...	...

(c) Table de transfert de B

FIGURE 2.19 – Réseau superposé.

Routage sensible à la source Le routage sensible à la source résout le problème de la cohabitation des deux routes sans besoin de configuration statique : la route native est inchangée, et une route spécifique (par son préfixe source) au réseau superposé est installée à travers le tunnel. La route spécifique à la source étant plus spécifique que la route native, les paquets du réseau superposé sont routés par le tunnel, tandis que les paquets encapsulés, qui trouvent leur origine au routeur de frontière, suivent la route native, non spécifique.

Par exemple, en figure 2.19 sont représentés à gauche (2.19*a*) deux réseaux physiques disjoints, chacun ayant une route pour l'Internet. Il est possible de simuler un lien entre ces réseaux en utilisant un VPN, par exemple entre A et B, comme représenté à droite (2.19*b*), puis vouloir que la connectivité à l'Internet du réseau superposé ne se fasse que par A. La table de transfert de B (2.19*c*) a donc deux routes par défaut : celle qu'il obtient par son FAI (r_1), correspondant au réseau physique, et celle qu'il obtient par le VPN (r_2), correspondant au réseau superposé. Si la route du VPN est spécifique aux adresses du réseau superposé, les

paquets du réseau superposé à destination de l'Internet passent par le VPN, puis par *A*, tandis que les autres paquets sortent directement par *B*.

Seules les passerelles vers l'Internet du réseau superposé sont à configurer : elles doivent annoncer une route spécifique aux adresses du réseau superposé. Dans notre exemple, seul *A* doit être configuré : *B* reçoit l'annonce de la route par le VPN, et ne nécessite donc aucune configuration de routage.

Nous nous sommes servis de ce cas d'usage dans notre réseau expérimental, avec notre version sensible à la source de Babel. Il s'agissait même de notre premier cas d'usage, avant le *multihoming*.

2.2.2 Solutions existantes au routage des réseaux multihomés

Comme nous l'avons dit, beaucoup de solutions existent [dLB06], faisant plus ou moins intervenir la coopération des FAI. Nous présentons dans cette section quelques techniques qui ne demandent pas la coopération des FAI.

2.2.2.1 Avec routage *next-hop*

Il est possible de router un réseau *multihomé* avec du routage *next-hop* et la configuration statique (mais automatisable) d'une partie ou de tous les routeurs.

Une première solution consiste à connecter tous les FAI au même routeur. Tous les paquets à destination de l'Internet sont alors acheminés vers ce routeur avec un protocole de routage *next-hop* classique. Il suffit de configurer le routeur avec des règles d'ingénierie de trafic (*policy-based routing*) pour transférer les paquets en fonction de leur adresse source au FAI adéquat. Cette solution introduit un point unique de défaillance : si le routeur connecté aux FAI tombe en panne, le réseau est sans connectivité.

Pour éviter d'avoir un point unique de défaillance, on peut connecter plusieurs routeurs à tous les FAI et configurer tous ces routeurs de manière analogue avec des règles d'ingénierie de trafic. Les paquets seront acheminés à l'un des routeurs, puis au bon FAI. Si un des routeurs tombe en panne, le routage *next-hop* reconvergera vers un autre routeur qui transférera à son tour les paquets jusqu'au bon FAI. Connecter un ou plusieurs routeurs à tous les FAI impose certaines restrictions sur la topologie du réseau et peut être difficile dans certains cas.

Pour contourner les problèmes dûs à la topologie, il est possible d'utiliser des tunnels entre deux points du réseau. La fiabilité du tunnel est assurée par le protocole de routage. Le routage des réseaux *multihomés* peut s'obtenir en interconnectant tous les routeurs de frontière par des tunnels et en configurant ces routeurs avec des règles d'ingénierie de trafic : une règle par tunnel, et une règle par FAI connecté au routeur. Un paquet à destination de l'Internet arrive à l'un des routeurs de frontière par le protocole de routage classique, puis est transféré par le routeur de frontière soit directement au bon FAI, si celui-ci est directement connecté au routeur de frontière, soit par un tunnel au routeur de frontière directement connecté au bon FAI. Avec cette méthode, les paquets ne sont pas routés directement au bon fournisseur d'accès mais peuvent être amenés à partir à un bout du réseau par le routage *next-hop*, puis à l'autre bout par le tunnel, augmentant la congestion et diminuant les performances globales du réseau.

Pour acheminer les paquets plus directement au bon fournisseur d'accès, l'utilisation de tunnels connectés aux routeurs de frontières peut être généralisé à tous les routeurs ou même aux hôtes. Dès qu'un paquet à destination de l'Internet est émis, il est encapsulé dans un tunnel qui l'achemine au bon routeur de frontière. Mais l'utilisation de tunnel a un coût : un coût sur la quantité d'état nécessaire au fonctionnement du réseau, notamment au niveau des routeurs de frontières, un coût sur la configuration statique nécessaire à établir les tunnels, et aussi un coût sur la transmission des données. En effet, d'une part la quantité de données transmise dans un paquet est diminuée du fait de l'encapsulation (diminution du MTU), et d'autre part l'encapsulation et la décapsulation consomment des ressources.

2.2.2.2 Routage par la source

Le routage par la source est un autre paradigme de routage que le routage *next-hop*. En routage par la source strict [SRC80], l'hôte émetteur choisit pour chaque paquet la route complète que le paquet doit traverser. Il spécifie donc la liste des routeurs successifs à parcourir dans l'entête même du paquet IP. L'entête IP possède en effet un champ extensible, appelé *Options*, qui peut contenir ces informations.

Par exemple, dans la figure 2.20, l'hôte A envoie un paquet à l'hôte F en spécifiant l'une des deux routes raisonnables : la route (A, B, C, E, F), ou la route (A, B, D, E, F). Les adresses source et destination du paquet IP sont respectivement A et F dans les deux cas, mais le chemin préconisé est mis en option.

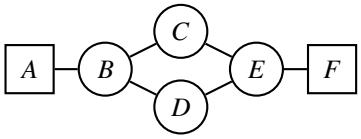


FIGURE 2.20 – Réseau à deux chemins de A à F.

Cette technique présente deux gros avantages. Premièrement, la décision de transfert des routeurs devient triviale : ils n'ont qu'à suivre les instructions fournies dans l'entête IP pour savoir vers quel routeur envoyer le paquet. Aucune opération de recherche dans une FIB n'est nécessaire, les couches basses des routeurs sont alors presque exonérées d'intelligence et de mémoire.

Deuxièmement, elle offre une grande flexibilité aux hôtes du réseau, qui peuvent utiliser plusieurs chemins différents à travers le réseau pour une même destination. Les applications de l'hôte peuvent alors facilement optimiser leurs performances en sélectionnant les chemins les plus adaptés à leur trafic, voire en sélectionnant plusieurs chemins simultanément.

C'est cette caractéristique qui laisse à penser que le routage par la source pourrait être une solution au routage des réseaux *multihomés* : le FAI par lequel passent les paquets serait choisi par l'hôte émetteur des paquets. Notons qu'il faudrait encore pouvoir identifier les préfixes sources associés aux FAI, ce qui n'est pas si évident.

Mais en dépit de ses avantages, ce paradigme a été jugé trop peu sûr [ASNN07], et n'a pas été retenu comme viable pour le routage dans l'Internet. En effet, il est possible à un attaquant de faire osciller un paquet entre deux routeurs dans le but de provoquer un déni de service. Au contraire, en routage *next-hop*, le contrôle du routage est laissé au réseau.

Le routage par la source a aussi un impact négatif sur la quantité de données pouvant être transmises par un paquet. Rappelons que la taille d'un paquet IP, avec son entête, est bornée par le MTU du réseau (cf. section 2.1.1). Or, en routage par la source, le chemin du paquet est contenu dans l'entête IP. L'espace occupé par cet entête est donc plus important qu'en routage classique et peut varier en fonction du chemin. Il en résulte alors, pour les couches supérieures, une diminution de l'espace disponible et une variation possible de cet espace.

2.2.3 Présentation du routage sensible à la source

Le routage sensible à la source étend le routage *next-hop* pour que la décision de transfert prenne aussi en compte l'adresse source des paquets. Contrairement au routage par la source, il ne pose pas plus de problème de sécurité que le routage *next-hop* car il garde la propriété fondamentale du routage *next-hop* : un nœud ne décide que du *next-hop* du paquet et non de son chemin dans le réseau. Contrairement aux tunnels, le routage sensible à la source n'a pas besoin d'un paquet IP modifié ni d'encapsulation : l'adresse source est déjà dans l'entête IP.

Le routage sensible à la source est une solution au routage des réseaux *multihomés*. Chaque route fournie par un FAI est annoncée comme spécifique au préfixe source fourni par ce FAI. Les paquets sont donc directement routés vers le bon FAI, comme le montre la figure 2.17.

Par ailleurs, le routage sensible à la source prévient l'émission et le transit de paquets indésirables. Si un hôte émet un paquet avec une adresse qui n'est fournie par aucun FAI, aucune route n'est possible pour ce paquet : il est détruit par le réseau, au premier routeur rencontré, et non par le FAI après avoir traversé le réseau et en être sorti.

2.2.4 État de l'art : routage

Le *multihoming* classique empêchant l'agrégation d'adresses (section 2.2.1.1), il est très vite apparu nécessaire de trouver des techniques permettant l'utilisation d'adresses agrégées. Lors de la conception d'IPv6, la distribution de préfixes indépendants des fournisseurs (PI) aux réseaux frontière a été prohibée. Depuis, le manque de solutions viables pour résoudre ce problème a poussé les organismes de distribution d'adresses à revenir sur cette pratique [NCC11], en distribuant malgré tout des PI aux réseaux frontière. Remarquons en particulier que dans le document cité, une organisation doit justifier sa nécessité de *multihoming* pour obtenir un préfixe PI.

En 2003, Huitema et Kessens ont proposé la mise en place de solutions pour un réseau *multihomé* constitué d'un seul lien [HK03]. Ils expliquent que de très nombreux réseaux domestiques ou de petites entreprises pourront ainsi bénéficier du *multihoming*. Grâce à ce cas particulier, ils soulèvent les principaux problèmes liés au *multihoming* PA :

— les paquets ayant une adresse source ou destination fournie par un FAI non accessible sont perdus car les FAI filtrent les paquets entrant en fonction de leur adresse source ;
— les connexions TCP sont perdues lors d'un changement d'adresses ;
— le trafic passant par un FAI peut être beaucoup plus lent ou congestionné que par un autre FAI : il faudrait choisir le meilleur.

Dans cette section, nous nous intéressons au premier point qui concerne le routage des paquets en milieu *multihomé*.

En 2004, le routage sensible à la source (SADR) statique apparaît. Il est décrit dans la littérature par Bagnulo et al. [BGmRA04], et la même année dans une RFC de Baker [BS04], et un ID (*Internet Draft*) de Huitema, Draves et Bagnulo [HM04]. La vision d'alors est d'introduire un réseau intermédiaire routé par SADR entre le réseau de l'entreprise et les fournisseurs d'accès pour guider les paquets vers le bon fournisseur. Ce réseau intermédiaire contient les routeurs de frontière. Il peut être connecté par des liens physiques, ou par des liens virtuels en utilisant des tunnels. Dans tous les cas, il nécessite une intervention manuelle de l'administrateur système.

Ces solutions se sont développées autour du groupe de travail *Site Multihoming in IPv6 (Multi6)* de l'IETF. Ce groupe de travail ne produit plus de documents à partir de 2005, et le groupe disparaît en 2007. Cependant, le problème du routage dans les réseaux *multihomés* n'était pas résolu pour autant, et de nouvelles solutions alternatives émergèrent depuis.

En 2011, Ohira et Al [OO11] proposent une forme simplifiée de routage sensible à la source. Ils montrent que l'introduction de routes utilisant les techniques classiques du SADR (tables multiples) peuvent causer des boucles de routage lorsque l'interprétation de l'information de routage du préfixe source n'est pas la même entre les routeurs. Bien qu'ils restent vagues sur la notion d'interprétation, il est probable qu'il s'agisse des mêmes problèmes que nous décrivons par la suite. Afin de simplifier leur modèle, ils décident : premièrement, de considérer comme deux problèmes séparés le routage des paquets à destination d'un nœud du réseau local et le routage des paquets à destination du reste de l'Internet ; deuxièmement, de considérer les préfixes source comme étant tous disjoints (ce qui fait sens dans leur cas d'usage) ; et troisièmement, de

considérer que les routes spécifiques à un préfixe source ont pour préfixe destination un préfixe par défaut (::/0). Pour cela, il y a deux protocoles en activité sur le même réseau : un protocole de routage classique pour router les paquets destinés au réseau local, et un nouveau protocole de routage pour router les paquets destinés au reste de l'Internet. Ce dernier est extrêmement simple : chaque routeur de frontière annonce en multicast son adresse IPv6 et le préfixe source dont il a la charge. Recevant l'annonce, les autres routeurs installent une route par défaut vers le routeur de frontière dans une nouvelle table ainsi que deux politiques de routage. La première (en terme de priorité) fait router les paquets à destination du préfixe source par la table de transfert classique — ces paquets sont en effet destinés au réseau. La seconde fait router les paquets ayant leur adresse source dans le préfixe source par la nouvelle table. Le next-hop associé à la route par défaut de la nouvelle table est celui utilisé par le protocole de routage classique pour joindre le routeur de frontière.

En 2013, Jin et al. s'intéressent au cas d'une configuration client-serveur où le réseau du serveur est *multihomé*. Leur idée est d'utiliser l'extension d'entête de paquet IPv6 *Routing* [JYY+13] pour spécifier l'adresse d'un routeur intermédiaire à traverser. Le serveur ajoute cet entête, tandis que le routeur de frontière le retire. Ainsi, chaque paquet envoyé par le serveur au client a pour adresse destination l'adresse du routeur de frontière ; l'adresse du client est spécifiée dans l'extension. Le routeur de frontière, lorsqu'il reçoit un paquet lui étant destiné mais ayant l'extension *Router*, substitue l'adresse destination du paquet par l'adresse contenue dans l'extension et transmet le paquet au fournisseur d'accès auquel il est connecté (sans entête *Routing*). Cette solution a plusieurs problèmes : elle nécessite une modification du plan de données des routeurs (leur manière de transférer les paquets) et modifie la quantité de données pouvant être envoyée par le serveur dans chaque paquet (Maximum Transmission Unit — MTU). En effet, de l'espace doit être réservé pour l'extension de l'entête IPv6. Jin et Al. proposent donc des mécanismes supplémentaires à ajouter au routeur pour permettre au serveur de déterminer la valeur réelle du MTU.

Enfin, dans le cadre du groupe de travail *Homenet* de l'IETF apparaît un nouvel attrait pour le routage sensible à la source. Fin 2012, Stenberg et Pfister réalisent une extension sensible à la source d'OSPF v3 [5] décrite par Troan [TC13] (IPv6). Cette extension, actuellement à l'abandon, est partielle : seules les routes par défaut peuvent être spécifiques à la source. Peu après, en 2013, nous apportons notre contribution en implémentant intégralement et indépendamment le routage sensible à la source dans le protocole de routage Babel [BC13, BC15], en IPv4 et IPv6. Le routage sensible à la source apparaît alors comme une bonne solution pour les réseaux *multihomés*. Lamparter puis Henning étendent respectivement les protocoles de routage IS-IS [Lam15] et OLSRv2 [CDJH14] au routage sensible à la source (IPv6).

2.3 Couches supérieures et multihoming

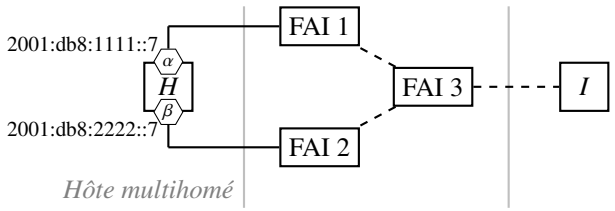
Nous avons vu dans la section précédente que le routage sensible à la source résout le problème du routage dans les réseaux *multihomés* avec plusieurs adresses. Même en cas de panne d'un FAI, le routage sensible à la source fournit au réseau une route pour l'Internet (spécifique au préfixe fourni par un autre FAI), mais il ne suffit pas à assurer la fiabilité des connexions du réseau. Cette responsabilité est laissée aux couches supérieures.

2.3.1 Couches supérieures et multihoming

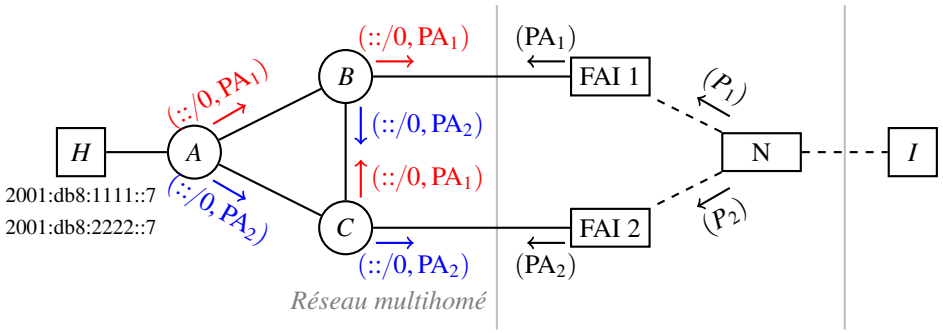
Une des difficultés du *multihoming* avec plusieurs adresses réside dans le fait qu'il n'est pas suffisant de bien router les paquets mais que des mécanismes supplémentaires sont nécessaires au niveau des hôtes pour assurer la fiabilité des connexions.

5. https ://github.com/fingon/hnet-core/commit/bcda8190c6a5fd6964e6ce30ecba7d9159b4ee82

Pour H, choisir une interface revient à choisir une adresse IP.



(a) Hôte multihomé *par deux interfaces*



(b) Hôte dans un réseau multihomé

Dans les deux cas :

- *le choix de l'adresse source des paquets envoyés par H détermine leur chemin (via FAI 1 ou FAI 2) ;*
- *le choix de l'adresse destination des paquets envoyés par I détermine leur chemin.*

FIGURE 2.21 – Hôte *multihomé* et hôte dans un réseau *multihomé* : même problème à résoudre.

Reprenons le cas de l'hôte *multihomé*, directement connecté par ses interfaces aux différents fournisseurs d'accès. Pour pouvoir communiquer par l'un et l'autre des fournisseurs, il dispose d'une adresse IP de chaque fournisseur — l'adresse fournie par un fournisseur est affectée à l'interface par laquelle l'hôte est connecté à ce fournisseur.

L'hôte *multihomé* n'a pas de problème de routage puisqu'il est directement connecté aux deux fournisseurs. Pourtant, le *multihoming* est déjà difficile. Une connexion TCP dépend des adresses IP source et destination (et aussi des ports source et destination). Lorsqu'une interface n'est plus connectée, l'adresse liée à cette interface n'est plus accessible et une connexion TCP qui l'utilisait est coupée. Par exemple, une connexion TCP initialisée avec l'adresse fournie par un wifi est perdue lorsque le téléphone devient hors de portée du wifi, même s'il a encore accès à l'Internet par le réseau cellulaire.

Disposer d'une connexion au réseau n'est donc pas suffisant pour assurer la fiabilité des connexions : de nouveaux protocoles sont nécessaires au niveau de l'hôte. En effet, l'hôte, au moment où il envoie un paquet, peut choisir son interface de sortie (et l'adresse IP qui lui est liée comme adresse source du paquet) et donc le fournisseur qu'il utilise (figure 2.21*a*). Inversement, son pair peut choisir l'interface d'entrée en choisissant l'adresse qui lui est liée comme adresse destination. Certaines technologies basées sur ces principes existent. Par exemple, MPTCP permet de passer d'une adresse à l'autre sans perte de connexion et de répartir la charge sur les différentes connexions (nous en reparlerons en section 5.2.1). Ces technologies doivent être déployées des deux côtés de la communication.

Le cas des réseaux *multihomés* avec plusieurs adresses est analogue (figure 2.21*b*). Rappelons que les paquets à destination de l'Internet sont routés via le fournisseur de l'adresse source du paquet. Envoyer des paquets avec une adresse source coïncide avec le FAI de sortie du réseau. Donc, pour qu'un hôte puisse être

connecté à l'Internet par chaque fournisseur d'accès, il doit disposer d'une adresse par fournisseur (même s'il n'a qu'une seule interface).

L'hôte du réseau *multihomé* se retrouve alors dans le même cas que l'hôte directement *multihomé* : il choisit son fournisseur en choisissant l'adresse source de ses paquets et c'est à lui d'assurer la fiabilité de ses connexions. Les mêmes mécanismes peuvent donc être utilisés dans un cas comme dans l'autre — c'est notamment le cas de MPTCP (section 5.2.1).

2.3.2 État de l'art : sélection d'adresses

En section 2.2.4, nous avons fait état de l'art en matière de routage pour les *multihomés*. Nous avons alors vu qu'en 2003, Huitema et Kessens soulèvent les principaux problèmes liés au *multihoming* PA dans le cadre d'un réseau *multihomé* constitué d'un seul lien [HK03] :

— les paquets ayant une adresse source ou destination fournie par un FAI non accessible sont perdus car les FAI filtrent les paquets entrant en fonction de leur adresse source ;
— les connexions TCP sont perdues lors d'un changement d'adresses ;
— le trafic passant par un FAI peut être beaucoup plus lent ou congestionné que par un autre FAI : il faudrait choisir le meilleur.

Dans cette section, nous nous intéressons à l'état de l'art pour les deux derniers points qui concernent la sélection d'adresses.

La modification des couches supérieures pour l'utilisation d'adresses différentes est encore (et étonnamment) plus ancienne. En effet, Huitema propose Multi-Home TCP, ancêtre de Multipath TCP, dès les années 1995 [Hui95]. Bien que l'objectif d'alors est de résister aux changements d'adresse IP *(roaming)*, pour la mobilité ou la renumérotation des réseaux, le *multihoming* apparaît déjà dans la liste des cas possibles d'usage. Ce n'est pourtant que trois ans après, en 1998, qu'est publié une RFC informationnelle incitant le filtrage des paquets par le FAI [FS98], laquelle devient bonne pratique depuis 2000 [FS00, BS04]. Un foisonnement de propositions voit le jour depuis 1999 [dLB06, NC11]. Nous en présentons les principales.

En 1999, Stewart décrit Multi-network Datagram Transmission Protocol (MDTP), un protocole de couche application pour la signalisation téléphonique, utilisé au-dessus d'UDP [SX99]. Le protocole est destiné à être utilisé dans des réseaux *multihomés* avec plusieurs adresses, chaque adresse pouvant donner lieu à un chemin différent. Le protocole est symétrique : chaque pair peut avoir plusieurs adresses, et envoie au pair distant la liste de ses adresses. Lorsqu'une panne est détectée sur le chemin utilisé, le protocole essaie un autre chemin.

Ce protocole ne sera pas normalisé, mais évolue rapidement et devient Simple Control Transmission Protocol (toujours un protocole de couche application) puis Stream Control Transmission Protocol (SCTP). Ce dernier est un protocole de couche transport, normalisé en 2000, et actualisé en 2007 [SXM+00, Ste07] [6]. Un point intéressant de SCTP est d'évaluer l'existence d'une liaison fonctionnelle régulièrement, par l'utilisation de sondes, appelées battement de cœur *(heartbeat)*.

À la même époque, en avril 1999, Draves décrit un algorithme pour la sélection d'une adresse source pour un paquet IPv6 [Dra99]. Ce travail aboutit à la standardisation de deux algorithmes de sélection d'adresses, l'un pour l'adresse source et l'autre pour l'adresse destination [Dra03] ; cette spécification est actualisée en 2012 par Thaler [TDMC12]. Les deux algorithmes sont indépendants, en ce sens qu'il n'en résulte pas un ensemble de paires d'adresses, mais la sélection d'une adresse destination dépend des adresses source disponibles, et la sélection d'une adresse source dépend de l'adresse destination choisie. En fait, l'adresse destination est souvent choisie avant l'adresse source.

En 2003, De Launois propose *NAROS (Name, Address and ROute System)*, une architecture dans laquelle les hôtes du réseau demandent à un serveur du réseau quelle adresse source utiliser pour une adresse

6. Le protocole est ainsi passé de 36 à plus de 150 pages, et de 2 à 10 auteurs.

destination donnée [dLBL03]. Il peut y avoir plusieurs serveurs NAROS, car celui-ci ne maintient pas d'état : tout au plus, il peut donner une adresse différente à chaque requête, par exemple en *round-robin*, pour effectuer une répartition de charge.

À partir de 2003, Nordmark décrit une couche intermédiaire entre la couche réseau et la couche transport, pouvant changer l'adresse source des paquets pour augmenter la fiabilité du réseau. Ce travail naît à travers le groupe de travail *multi6*, puis s'étend à travers son propre groupe de travail, *Site Multihoming by IPv6 Intermediation (shim6)*, où il est finalement normalisé en 2009 sous le nom de Shim6 [NB09]. L'avantage de Shim6 est de fournir la résilience aux pannes aux couches supérieures, qui n'ont pas besoin d'être modifiées. Toutefois, Shim6 ne permet pas à un hôte de survivre à une renumérotation simultanée de toutes ses adresses. Il lui est nécessaire, pour établir de nouveaux chemins, qu'au moins une des précédentes adresses soit valide et permette la communication.

Enfin, l'idée d'un nouveau TCP capable d'utiliser plusieurs paires d'adresses IP, déjà présentée par Huitema [Hui95], et ayant connu plusieurs propositions, se concrétise finalement avec MultiPath TCP (MPTCP) à partir de 2009 [FRHB09, RPB+12]. MPTCP est une extension compatible de TCP qui est capable de résister aux changements d'adresses et de faire de la répartition de charge avec contrôle de congestion. MPTCP a son propre groupe de travail au sein de l'IETF, et a de grandes chances de devenir un standard Internet. Nous décrivons davantage MPTCP en section 5.2.1.

Chapitre 3

Routage sensible à la source : mise en œuvre

Dans le chapitre précédent, nous avons vu que router les paquets en fonction de leur adresse source résolvait des problèmes difficiles comme celui du routage des réseaux *multihomés* avec plusieurs adresses (section 2.2.1.1). Nous avons aussi vu que les techniques utilisées pour router les paquets reposaient sur la configuration statique de tunnels ou de règles de trafic plutôt que sur un protocole de routage sensible à la source (section 2.2.2). Cependant, l'utilisation d'un protocole de routage sensible à la source a plusieurs intérêts :

— la seule configuration nécessaire est celle du protocole de routage, déployé uniformément dans tout le réseau ;
— le protocole réagit de manière dynamique aux changements du réseau ;
— le protocole trouve les meilleurs chemins ;
— il n'y a pas de surcoût similaire à ceux induits par les tunnels (modification du MTU, encapsulation).

Dans ce chapitre, nous montrons comment obtenir un protocole de routage sensible à la source, en partant du transfert. Nous commençons (section 3.1) par décrire les tables de transfert et à montrer qu'en cas d'ambiguïté, les différentes implémentations existantes peuvent avoir des comportements différents ce qui cause des boucles de routage persistantes. En section 3.2, nous définissons un algorithme utilisable par un protocole de routage qui permet de modifier la FIB des routeurs en la préservant des ambiguïtés. Cet algorithme est la principale contribution de ce manuscrit. Enfin, en section 3.3, nous montrons comment réaliser et étendre de manière compatible un protocole de routage sensible à la source à vecteur de distances.

3.1 Transfert sensible à la source

Le routage sensible à la source est une extension au routage *next-hop* où :

— chaque nœud choisit le *next-hop* d'un paquet (et non son chemin dans le réseau) — cette caractéristique fondamentale du routage *next-hop* reste inchangée ;
— ce choix dépend de l'adresse destination du paquet, *de l'adresse source du paquet*, et de la FIB des routeurs.

Dans cette section, nous décrivons en détails le transfert sensible à la source et les modifications nécessaires aux tables de transfert. Nous voyons que, comme pour le routage *next-hop*, la question de l'ambiguïté des tables de transfert se pose. Mais, contrairement au routage *next-hop*, la spécificité des préfixes ne suffit plus : nous utilisons l'ordre lexicographique sur les paires de préfixes $(\text{destination}, \text{source})$ induit par la

spécificité. Nous voyons aussi que tous les routeurs d'un réseau doivent implémentent le même ordre et que ce n'est pas toujours le cas en pratique. La levée d'ambiguïté est l'objet de la section 3.2.

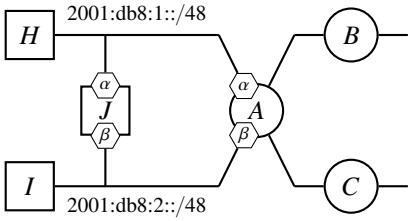
3.1.1 Une extension au routage *next-hop*

En routage *next-hop*, un paquet est transféré par un routeur en fonction de son adresse destination et de la FIB du routeur. Celle-ci est une table de transfert qui associe un *next-hop* à un préfixe destination. Le routeur ne détermine que le *next-hop* du paquet et non son chemin dans le réseau.

Le routage sensible à la source étend le routage *next-hop* pour que la décision de transfert prenne aussi en compte l'adresse source, mais garde la propriété fondamentale du routage *next-hop* : un nœud ne décide que du *next-hop* du paquet et non de son chemin dans le réseau. Notons aussi que le routage sensible à la source n'a pas besoin d'un paquet IP modifié : l'adresse source est déjà dans l'entête IP.

Donc, un paquet est transféré par un routeur en fonction de son adresse destination, de son adresse source et de la FIB des routeurs. Celle-ci est étendue pour associer un *next-hop* à une paire de préfixes destination et source. Une entrée d'une table sensible à la source est donc sous la forme $((D, S), NH)$, avec D un préfixe destination, S un préfixe source et NH le *next-hop* associé. On dit qu'une telle entrée *route* un paquet (d, s) lorsque $(d, s) \in D \times S$.

Considérons par exemple la figure 3.1 et la table de transfert du routeur A. Celle-ci comporte deux routes pour la même destination $::/0$ mais pour des sources différentes. Les paquets ayant leur adresse source dans 2001:db8:1::/48 sont transférés à B, tandis que les paquets ayant leur adresse source dans 2001:db8:2::/48 sont transférés à C. Concrètement, les paquets émis par H (à destination de l'Internet) sont transférés à B, et ceux émis par I sont transférés à C. Remarquons aussi qu'il n'y a aucune route à proprement parler par défaut (qui accepte tout paquet) : par exemple un paquet ayant pour source 2001:db8:3::1 n'est pas routé.



destination	source	*next-hop*
2001:db8:1::/48	::/0	$A_\alpha \rightarrow dest$
2001:db8:2::/48	::/0	$A_\beta \rightarrow dest$
::/0	2001:db8:1::/48	$A \rightarrow B$
::/0	2001:db8:2::/48	$A \rightarrow C$

FIGURE 3.1 – Un réseau avec deux routes sensibles à la source, et la table de A.

Une extension plus expressive Remarquons qu'une entrée spécifique à un préfixe source de longueur nulle, par exemple $(2001{:}db8{:}1{::}/48, ::/0)$, est équivalente à une entrée non spécifique de même préfixe destination, ici 2001:db8:1::/48. Ces deux entrées routent, en effet, tous les paquets à destination de 2001:db8:1::/48, quelle que soit l'adresse source de ces paquets.

Une table de transfert classique peut donc toujours s'exprimer en terme de table sensible à la source. Il suffit pour cela d'ajouter un préfixe source de longueur nulle à chaque entrée. De même, une table sensible à la source dont chaque entrée a un préfixe source de longueur nulle est équivalente à la table de transfert classique composée des mêmes entrées, le préfixe source omis. Le routage sensible à la source est plus expressif que le routage *next-hop*.

Plus de flexibilité aux hôtes Rappelons qu'en routage *next-hop*, les hôtes ne décident pas du chemin du paquet dans le réseau mais que ce rôle revient au réseau (section 2.1.2). Les hôtes ont tout de même une certaine flexibilité : le choix de l'interface de sortie et le choix de l'adresse destination du pair. Le choix de l'adresse destination a une influence sur l'ensemble du chemin, puisqu'à chaque nœud, une décision de

transfert est prise en fonction de cette adresse. Le choix de l'interface de sortie a un impact plus limité : pour une même destination, les routes partant d'une interface suivent la même route dès le premier routeur commun. Par exemple, sur la figure 3.1, supposons que J envoie des paquets à destination de l'Internet, et que A soit un routeur *next-hop* — il n'a donc qu'une route vers l'Internet (soit par B, soit par C). L'influence du choix de l'interface par J sur le chemin des paquets est alors restreint au premier lien traversé.

En routage sensible à la source, le choix de l'adresse source d'un paquet a une influence similaire au choix de l'adresse destination du paquet : à chaque nœud, le transfert dépend de l'adresse source du paquet et deux paquets avec des adresses sources différentes sont susceptibles de suivre des chemins différents. C'est le cas sur la figure 3.1 : les paquets émis par l'hôte J avec son adresse dans 2001:db8:1::/48 passent par B, les autres passent par C.

La flexibilité liée au choix de l'adresse source est complémentaire de celle liée au choix de l'adresse destination. Nous avons vu en section 2.3 que dans le cas du *multihoming*, ce choix était nécessaire aux hôtes pour qu'ils puissent assurer la fiabilité de leurs connexions. Nous revenons sur les possibilités offertes par le choix de l'adresse source au chapitre 5.

Ambiguïtés Dans la suite de cette section, nous nous intéressons aux ambiguïtés des tables sensibles à la source. Comme pour le routage *next-hop*, plusieurs entrées peuvent router les mêmes paquets. En section 3.1.2, nous voyons que choisir les entrées les plus spécifiques n'est plus suffisant à lever les ambiguïtés, nous décrivons des comportements réalistes possibles, nous montrons qu'il est essentiel de lever les ambiguïtés, et nous décrivons le consensus de la communauté qui est de préférer les entrées ayant le préfixe destination le plus spécifique et, en cas d'égalité, le préfixe source le plus spécifique. La résolution des ambiguïtés est traitée en section 3.2. Enfin, nous faisons en section 3.1.3 un tour des implémentations existantes de FIB dans les routeurs.

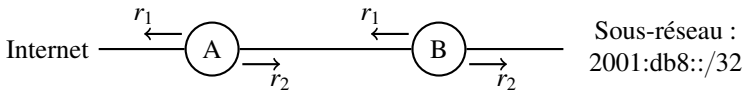
3.1.2 Ambiguïtés des tables sensibles à la source

Rappelons qu'en routage *next-hop* classique, une table de transfert peut être ambiguë. C'est le cas lorsqu'un paquet est routé par deux entrées différentes. Les ambiguïtés sont levées par la règle du préfixe le plus spécifique : c'est l'entrée ayant le préfixe destination le plus spécifique qui est retenue pour router le paquet. Cette règle suffit à lever les ambiguïtés car la spécificité des préfixes induit un ordre total sur l'ensemble des routes non disjointes (routant un même paquet). Nous voyons dans les prochains paragraphes qu'en routage sensible à la source, les tables de transfert peuvent aussi être ambiguës mais que, par contre, la spécificité des préfixes ne suffit pas à lever les ambiguïtés. En effet, celle-ci n'induit plus un ordre total sur les paires de préfixes non disjointes.

Considérons l'exemple de la figure 3.2 où à la fois A et B ont une route $r_1 = (::/0, 2001{:}db8{::}/32)$ vers la gauche, et une route $r_2 = (2001{:}db8{::}/32, ::/0)$ vers la droite. La table de transfert de A (comme celle de B) est ambiguë : r_1 et r_2 routent toutes deux les paquets à destination de 2001:db8::/32 et en provenance de 2001:db8::/32, comme par exemple le paquet (dst = 2001:db8:1::2, src = 2001:db8:5::3). Si A préfère r_1 le paquet sera routé à gauche, et s'il préfère r_2 il sera routé à droite.

L'ordre de spécificité sur les préfixes utilisé pour lever les ambiguïtés en routage *next-hop* (cf. section 2.1.2) s'étend aux paires de préfixes. Une paire de préfixe (D, S) est plus spécifique qu'une autre paire (D', S') si $(D, S) \subset (D', S')$, c'est-à-dire si toutes les paires (d, s) de (D, S) sont aussi dans (D', S'). Formellement, $(D, S) \leqslant (D', S')$ si $D \leqslant D'$ et $S \leqslant S'$.

Dans notre exemple, r_1 n'est pas plus spécifique que r_2 et inversement : la première a un préfixe source plus spécifique, et la seconde un préfixe destination plus spécifique. La spécificité des préfixes ne suffit plus à lever les ambiguïtés, car elle n'induit plus un ordre total sur l'ensemble des routes routant un même paquet.



(a) Topologie du réseau

	destination	source	*next-hop*
(r_1)	::/0	2001:db8::/32	$A \rightarrow$ "Internet"
(r_2)	2001:db8::/32	::/0	$A \rightarrow B$

(b) Table de transfert de A

FIGURE 3.2 – Réseau avec routes spécifiques.

Router par la destination d'abord En présence d'ambiguïté, nous voyons qu'un choix se pose. Le routeur A de la figure 3.2 transfère le paquet $(dst = 2001{:}db8{:}1{::}2, src = 2001{:}db8{:}5{::}3)$ vers la gauche s'il préfère la route r_1 à la route r_2, et vers la droite s'il préfère la route r_2 à la route r_1. Or, il est primordial que tous les routeurs d'un même réseau aient le même comportement en cas d'ambiguïté : il faut que deux routeurs ayant les mêmes entrées dans la table de transfert préfèrent la même entrée pour le routage d'un même paquet. Considérons notre exemple de la figure 3.2. À la fois A et B possèdent r_1 et r_2 dans leur table de transfert. Or, si A préfère r_1 à r_2, et B l'inverse, une boucle de routage persiste entre A et B pour tous les paquets routés à la fois par r_1 et r_2. Nous en déduisons la propriété suivante.

Propriété 1 *Tous les routeurs d'un même réseau doivent avoir la même politique de routage, sans quoi des boucles de routage persistantes peuvent exister.*

Il convient donc de choisir une politique de routage déterministe et uniforme pour un réseau donné. Le consensus actuel, aussi bien dans le groupe de travail *Homenet* de l'IETF qu'en dehors, est de router les paquets en préférant les entrées ayant le préfixe destination le plus spécifique et, en cas d'égalité, les entrées ayant le préfixe source le plus spécifique (on parle de routage par la destination d'abord). Plus formellement, il s'agit de la linéarisation de l'ordre de spécificité $<$ par l'ordre lexicographique $(\text{destination}, \text{source})$ défini par

$$(D,S) \preceq (D',S') \text{ si } D < D' \text{ ou } D = D' \text{ et } S \leqslant S'.$$

Un exemple de topologie réaliste nous poussant à choisir cet ordre est celui de la figure 3.1. Ici, un paquet envoyé par H à I passe par A. Ce dernier a deux routes acceptant de router le paquet : une non spécifique qui mène directement à I, $(2001{:}db8{:}2{::}/48, ::/0)$, et une spécifique qui passe par B et fait sortir les paquets du réseau, $(::/0, 2001{:}db8{:}1{::}/48)$. Clairement, si A envoie le paquet par la route spécifique, le paquet sort du réseau et n'a donc aucune chance d'accéder à sa destination. Même s'il revenait dans le réseau (par C), il repasserait par A où il serait à nouveau expulsé par B. Ici, router par la destination d'abord est correct, contrairement aux autres politiques de routage que nous présentons dans le paragraphe suivant.

Autres politiques de routage Le routage par la destination d'abord est arbitraire et d'autres comportements sont possibles en cas d'ambiguïté. Nous en présentons quelques uns et donnons des algorithmes simples les implémentant en figure 3.3. Le routage par la destination d'abord est représenté sur l'algorithme 1.

Tout d'abord, il est naturel de penser au symétrique du routage par la destination d'abord : router par la source d'abord (algorithme 2). Il s'agit de la linéarisation de l'ordre de spécificité par l'ordre lexicographique $(\text{source}, \text{destination})$ défini par

$$(D,S) \preceq_s (D',S') \text{ si } S < S' \text{ ou } S = S' \text{ et } D \leqslant D'.$$

```
1  Function dst-src(d, s)
2      (D', S') ← ⊥
3      nh ← ⊥
4      for all ((D, S), NH) ∈ T
5      s.t. d ∈ D and s ∈ S
6          if D' = ⊥ or D < D' or
7          (D = D' and S < S')
8              (D', S') ← (D, S)
9              nh ← NH
10     return nh
```

Algorithme 1 : Destination d'abord.

```
1  Function src-dst(d, s)
2      (D', S') ← ⊥
3      nh ← ⊥
4      for all ((D, S), NH) ∈ T
5      s.t. d ∈ D and s ∈ S
6          if D' = ⊥ or S < S' or
7          (S = S' and D < D')
8              (D', S') ← (D, S)
9              nh ← NH
10     return nh
```

Algorithme 2 : Source d'abord.

```
1  Function first-match(d, s)
2      (D', S') ← ⊥
3      nh ← ⊥
4      for all ((D, S), NH) ∈ T
5      s.t. d ∈ D and s ∈ S
6          if D' = ⊥ or (D < D' and
           S < S')
7              (D', S') ← (D, S)
8              nh ← NH
9      return nh
```

Algorithme 3 : Première plus spécifique trouvée.

```
1  Function reject(d, s)
2      (D', S') ← ⊥
3      nh ← ⊥
4      for all ((D, S), NH) ∈ T
5      s.t. d ∈ D and s ∈ S
6          if D' = ⊥ or (D < D' and S < S')
7              (D', S') ← (D, S)
8              nh ← NH
9          else if not (D > D' and S > S')
10             (D', S') ← (min(D, D'), min(S, S'))
11             nh ← ⊥
12     return nh
```

Algorithme 4 : Ne pas router les ambiguïtés.

FIGURE 3.3 – Algorithmes naïfs de sélection de route.

Contrairement au routage par la destination d'abord, nous ne connaissons pas de topologie réaliste qui l'utilise. Mais en pratique, il arrive que la seule interface qu'offre une FIB pour faire du routage sensible à la source route les paquets par la source d'abord. Nous abordons en plus de détails ces interfaces en section 3.1.3.

Il est aussi possible de choisir la première entrée parmi les entrées satisfaisantes les plus spécifiques (algorithme 3). Ce comportement introduit une forme de choix aléatoire, puisqu'avec un même réseau, le choix peut être différent en fonction de l'ordre d'apparition des routes. Parce qu'il n'est pas tout à fait déterministe, il est fondamentalement incompatible avec un routage sensible à la source correct. Toutefois, on peut argumenter que les routes peuvent être installées dans le bon ordre : il est alors facile d'imposer l'ordre qu'on veut au routeur. En revanche, ajouter une route peut nécessiter la suppression et la réinsertion de beaucoup d'autres routes ; l'erreur est d'autant plus probable avec un routage statique.

Enfin, il est aussi possible de ne pas router les paquets qui sont soumis à une ambiguïté (algorithme 4). Cette option nécessite de détecter les ambiguïtés. L'algorithme proposé est une recherche de minimum : nous cherchons une entrée plus spécifique que toutes les autres, c'est-à-dire une entrée dont le préfixe source est plus spécifique que tous les préfixes source rencontrés et dont le préfixe destination est plus spécifique que tous les préfixes destination rencontrés. Lorsqu'une ambiguïté est trouvée, nous mettons à jour la destination la plus spécifique rencontrée, et la source la plus spécifique rencontrée ($(D', S') \leftarrow (\min(D, D'), \min(S, S'))$). L'avantage de cette méthode est de ne pas risquer de faire de boucles de routage. Par contre, même si tous les routeurs ont ce comportement, elle génère des trous noirs.

Considérons l'exemple de la figure 3.1 et un paquet envoyé par H à I. Nous avons vu que, pour atteindre sa destination, ce paquet doit être routé par la deuxième entrée du routeur A. Or, router ce paquet par la

source d'abord sélectionne la troisième entrée : le paquet sort du réseau. De même, si on lit la table de bas en haut, alors router par la première entrée la plus spécifique sélectionne la troisième entrée. Enfin, ne pas router les paquets pour lesquels il y a ambiguïté mène à la destruction du paquet. Parmi les quatre politiques de routage que nous avons décrites, seule celle de router par la destination d'abord route le paquet vers sa destination.

Vers une table sans ambiguïté Remarquons qu'en ajoutant l'entrée $r_3 = ((2001{:}db8{::}/32, 2001{:}db8{::}/32), A \to B)$ à la table de la figure 3.2, comme illustré ci-dessous, l'ordre de spécificité est suffisant pour lever toutes les ambiguïtés. Si un paquet est dans r_3, intersection de r_1 et r_2, alors il est routé par r_3, qui est plus spécifique que les deux autres routes. Sinon, si un paquet est dans r_1, alors r_1 est la seule entrée à pouvoir le router : il n'y a pas d'ambiguïté ; de même pour r_2.

	destination	source	*next-hop*
(r_1)	::/0	2001:db8::/32	$A \to$ "Internet"
(r_2)	2001:db8::/32	::/0	$A \to B$
(r_3)	2001:db8::/32	2001:db8::/32	$A \to B$

Nous remarquons aussi que les quatre algorithmes précédents sont compatibles avec l'ordre de spécificité : ils choisissent tous les entrées les plus spécifiques. Dans notre exemple, ajouter r_3 force le routeur à suivre le comportement équivalent à la destination d'abord, quelle que soit la manière dont il traite les ambiguïtés. Une table non ambiguë est interprétée de la même manière par tous les routeurs.

Nous voyons en section 3.2 qu'il est possible de lever toutes les ambiguïtés d'une table de transfert sensible à la source en y ajoutant des entrées. Nous y définissons un algorithme, utilisable dans un protocole de routage, qui maintient la FIB sans ambiguïté. Lever les ambiguïtés est essentiel pour les protocoles de routage qui sont chargés de peupler les tables de transfert. En effet, comme nous allons le voir, toutes les FIB supportant le routage sensible à la source n'implémentent pas le même ordre.

3.1.3 Implémentations existantes

Chaque système d'exploitation implémente sa propre FIB et possède une interface de programmation (API) qui permet de la configurer. Nous avons trouvé deux API permettant de faire du routage sensible à la source. Nous les utilisons toutes les deux dans notre implémentation sensible à la source de Babel (chapitre 4).

Nous décrivons ici ces deux API, et voyons que les ordres de ces deux API ne sont pas les mêmes. La première utilise une table de transfert nativement sensible à la source avec l'ordre, que nous voulons, par la destination d'abord. La seconde utilise plusieurs tables de transfert classiques et une table de règles avec l'ordre, contraire à celui que nous voulons, par la source d'abord.

Table sensible à la source native L'API *netlink* du noyau Linux permet d'associer un préfixe source à une route : la FIB correspondante est nativement sensible à la source par la destination d'abord — l'ordre que nous voulons. Toutefois, cette API ne fonctionne qu'en IPv6 pour des noyaux récents (depuis 3.11) compilés avec la bonne option ("ipv6-subtrees"). Les versions antérieures traitent ces routes comme inaccessibles (*unreachable*), problème corrigé en partie grâce à notre travail. En IPv4, le préfixe source est toujours ignoré.

L'utilisation de cette API est illustrée en figure 3.4, à l'aide de la commande `ip` : une route spécifique à la source $(::/0, 2001{:}db8{:}1{::}/48)$ est ajoutée. L'ajout de la route se fait en spécifiant le préfixe destination (`add ::/0`) et le préfixe source (`from 2001:db8:1::/48`) dans le même message. Une fois dans la FIB, elle correspond bien à une seule entrée, spécifique à la source.

```
# sudo ip -6 route add ::/0 from 2001:db8:1::/48 dev eth0
# ip -6 route show
default dev eth0 from 2001:db8:1::/48 metric 1024
...
```

FIGURE 3.4 – Ajout d'une route spécifique avec les *subtrees* Linux.

Si le routage sensible à la source se développe, nous pouvons espérer que toutes les FIB des différents systèmes disposeront de tables sensibles à la source natives. Actuellement, nous ne connaissons pas d'autre support que celui, partiel, que nous avons décrit.

Tables multiples La fonctionnalité des tables multiples est disponible sur beaucoup de systèmes mais ordonne les entrées par la source d'abord — le contraire de l'ordre que nous voulons. Plusieurs tables de transfert classiques sont associées chacune à un préfixe source par l'intermédiaire de règles d'ingénierie de trafic. Lors du transfert d'un paquet, l'adresse source du paquet sert à trouver une table de transfert, puis l'adresse destination du paquet est utilisée par cette table pour trouver le *next-hop*.

Nous n'avons testé l'usage des tables multiples que sous Linux, bien que FreeBSD et Cisco IOS disposeraient aussi d'une API plus ou moins similaire. Sous FreeBSD, il semble que ce système soit peu flexible, puisque le nombre de tables de transfert dont le système dispose doit être spécifié à la compilation : à la base, les tables de transfert n'étaient pas destinées à l'usage que nous en faisons. Sous Linux, l'utilisation est plus flexible : les tables de transfert et les règles d'ingénierie de trafic sont créées à la volée. Les règles comportent au moins 3 paramètres : une priorité qui détermine l'ordre d'évaluation des règles, le motif du paquet (le préfixe source), et l'action associée (choix de la table). Les règles sont évaluées dans l'ordre de la priorité. Si un paquet correspond au motif de la règle, il est routé selon la table de transfert spécifiée. Si la table de transfert ne route pas le paquet, la règle suivante est évaluée.

La figure 3.5 montre l'ajout de la route spécifique à la source $(::/0, 2001{:}db8{:}1{::}/48)$ en utilisant des tables multiples. Deux instructions sont utilisées : celle de la création de l'entrée dans la nouvelle table de transfert (`route add`) et celle de la création de la nouvelle règle (`rule add`). L'ordre d'exécution de ces deux instructions est sans importance. Ici, nous commençons par créer une entrée non spécifique dans la table de transfert de numéro 70, qui est alors créée à la volée. Puis, nous ajoutons la règle correspondante au préfixe source en lui donnant une priorité de 300, le préfixe source (2001:db8:1::/48) et le numéro de la table associée (70). Après cette opération, la table des règles contient 3 entrées. La première indique, avec une priorité de 0 (la plus prioritaire), que tous les paquets doivent d'abord être routés par la table `local` : les paquets à destination d'une adresse locale au nœud ne doivent pas sortir sur le réseau. Lorsqu'un paquet ne peut être routé par cette table, la deuxième règle, celle que nous avons ajoutée, est invoquée. Enfin, la troisième règle, avec la plus haute valeur de priorité (la moins prioritaire), indique le comportement par défaut : les paquets sont routés par la table de transfert principale.

```
# ip -6 route add ::/0 dev eth0 table 70
# ip -6 rule add prio 300 from 2001:db8:1::/48 table 70
# ip -6 rule show
0:      from all lookup local
300:    from 2001:db8:1::/48 lookup 70
32766:  from all lookup main
# ip -6 route show table 70
default dev eth0 metric 1024
```

FIGURE 3.5 – Ajout d'une route spécifique avec l'API des tables multiples de Linux.

Il convient, lors de l'utilisation de cette API pour effectuer du routage sensible à la source, de garder en mémoire les règles ajoutées. Lorsqu'un nouveau préfixe source est ajouté, une nouvelle table de transfert

doit être créée avec la règle associée. Un soin tout particulier doit être apporté lors de l'attribution des priorités afin de garder les règles dans un ordre cohérent avec la spécificité des préfixes.

L'utilisation des tables multiples implémente alors un ordre compatible avec l'ordre de spécificité : l'ordre par la source d'abord. Les techniques que nous proposons en section 3.2 et que nous avons évoquées à la section précédente construisent des tables sans ambiguïté pour l'ordre de spécificité : il est possible de peupler ces tables à partir d'une RIB ordonnée par la destination d'abord.

3.2 Levée d'ambiguïtés

Nous venons de voir que la spécificité des préfixes ne suffit pas à résoudre les ambiguïtés des tables de transfert sensibles à la source, que les FIB supportant le routage sensible à la source n'implémentent pas nécessairement le même ordre et que tous les nœuds du réseau doivent implémenter le même ordre pour éviter des boucles de routage persistantes.

Nous définissons dans cette section un algorithme de levée d'ambiguïté utilisable par un protocole de routage. Il s'introduit comme une couche intermédiaire entre la RIB et la FIB et fournit trois opérations d'ajout, de suppression et de modification de la FIB. Il garde la FIB sans ambiguïté en maintenant des entrées additionnelles.

3.2.1 Placement de l'algorithme

Nous avons représenté sur la figure 3.6 les différentes étapes effectuées par les protocoles de routage pour peupler les FIB des routeurs. En vertical, nous voyons que les protocoles s'échangent des données pour calculer leur RIB. Ils installent ensuite directement les entrées de la RIB dans la FIB (certaines informations peuvent être perdues, comme la métrique). La table intermédiaire que nous avons appelée *RIB complète* est l'étape que notre algorithme rajoute.

La RIB complète est une table de routage similaire à la RIB mais sans ambiguïté pour l'ordre de la FIB. Elle est construite en ajoutant à la RIB des entrées additionnelles comme nous en avons donné une illustration en section 3.1 et comme nous le voyons en détail dans la suite. Parce que la RIB complète est sans ambiguïté pour la FIB, ses entrées peuvent être installées directement dans la FIB. Notre algorithme préserve la complétude de la FIB.

Notre algorithme ne construit pas explicitement la RIB complète, mais calcule les entrées à installer à la volée à partir de la RIB. Aucune entrée de la RIB complète n'est gardée en mémoire, même les entrées additionnelles. Notre algorithme est donc sans mémoire.

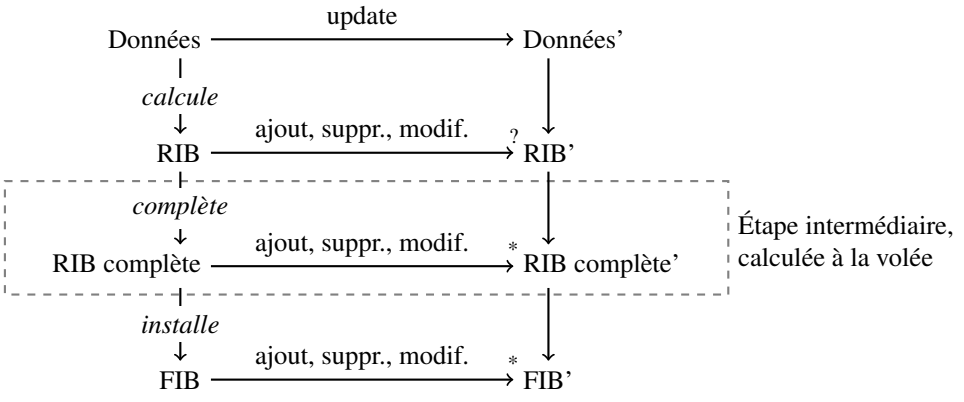


FIGURE 3.6 – Étapes de passage de la RIB à la FIB, ajout de la complétion.

Les modifications dans le réseau sont généralement incrémentales, ce que nous représentons en horizontal sur la figure 3.6. Ainsi, une mise à jour incrémentale du réseau (*update*) ne donne pas lieu à un changement de la RIB ou donne lieu à une modification incrémentale de la RIB : ajout, suppression ou modification d'une entrée. De même, une modification incrémentale d'une entrée de la RIB peut donner lieu à une modification incrémentale de la FIB. Ainsi, les primitives de modification de la FIB sont incrémentales.

Notre algorithme, en se plaçant entre la RIB et la FIB, fournit des opérations incrémentales similaires à celle de la FIB. Il suffit donc de remplacer les primitives de la FIB par celles-ci. C'est alors à notre algorithme d'utiliser les primitives de la FIB pour gérer les entrées additionnelles qu'il engendre. Parce que notre algorithme doit maintenir la table complète, une modification incrémentale de la RIB peut donner lieu à plusieurs modifications incrémentales de la RIB complète, et autant de modifications de la FIB.

3.2.2 Définitions et hypothèse

Dans cette section, nous rappelons les définitions du routage *next-hop* et du routage sensible à la source et nous fixons les notations des entrées des tables de routage et de leur manipulation. La seule hypothèse que nous faisons est de supposer que l'ordre imposé par la FIB est compatible avec l'ordre de spécificité sur les paires de préfixes. Nous définissons aussi les notions d'ambiguïté et de conflit utilisées dans la suite, et nous montrons qu'il suffit de considérer les entrées deux à deux pour trouver toutes les ambiguïtés. Enfin, nous définissons la notion de complétude que nous utilisons et prouvons qu'une table complète est sans ambiguïté.

3.2.2.1 Rappels, définitions et hypothèse

En routage *next-hop*, une entrée d'une table de transfert associe un *next-hop* à un préfixe. Un préfixe est défini comme un ensemble d'adresses, et une entrée (D, NH) de préfixe D et de *next-hop NH* route un paquet de destination d si $d \in D$. Lorsque plusieurs entrées routent un même paquet, l'entrée de préfixe le plus spécifique est choisie. Il s'agit du minimum pour l'ordre de spécificité $<$. Cet ordre coïncide avec l'inclusion : le préfixe contenant le moins d'adresses est choisi. Formellement, si D_1 et D_2 sont deux préfixes, $D_1 < D_2$ si et seulement si $D_1 \subset D_2$.

En routage sensible à la source, une entrée d'une table de transfert associe un *next-hop* à une paire de préfixes. Une entrée $((D, S), NH)$ route un paquet (d, s) si $d \in D$ et $s \in S$. Comme en routage *next-hop*, plusieurs entrées peuvent router un même paquet. L'ordre de spécificité s'étend naturellement aux paires de préfixes : $(D_1, S_1) <= (D_2, S_2)$ si et seulement si $D_1 <= D_2$ et $S_1 <= S_2$. Mais l'ordre de spécificité ne suffit plus : le consensus est de choisir l'entrée qui a le préfixe destination le plus spécifique et à défaut le préfixe source le plus spécifique. Il s'agit de l'ordre lexicographique par la destination d'abord $\prec$ défini par : $(D_1, S_1) \prec (D_2, S_2)$ si et seulement si $D_1 < D_2$ ou $D_1 = D_2$ et $S_1 < S_2$.

Les API que nous avons utilisées n'implémentent pas toujours l'ordre lexicographique par la destination d'abord : l'utilisation de tables multiples, nécessaires dans certains cas, nous force a avoir un ordre lexicographique par la source d'abord. Or, ces deux ordres sont compatibles avec l'ordre de spécificité. Nous avons vu dans un exemple qu'en ajoutant une entrée à une table de transfert celle-ci avait un minimum pour l'ordre de spécificité : elle était routée de la même manière quel que soit l'ordre de la table.

L'ordre de spécificité est la plus grande relation compatible à la fois avec l'ordre lexicographique par la destination d'abord ($\prec$), et l'ordre lexicographique par la source d'abord ($\prec_s$) : $<=\prec \cap \prec_s$. L'algorithme que nous proposons maintient la FIB sans ambiguïté pour l'ordre de spécificité. Nous ne nous soucions donc plus de l'ordre réel de la FIB, mais seulement de l'ordre de spécificité $<$ et de l'ordre du protocole. Nous notons ce dernier $\prec$, à l'image de l'ordre lexicographique par la destination d'abord que nous utilisons pour nos exemples.

Notre algorithme de levée d'ambiguïté ne suppose donc qu'une seule chose : la FIB a un ordre compatible avec l'ordre de spécificité. Nous avons toujours trouvé cette hypothèse vérifiée (cf. section 3.1.3).

Notations des entrées Une entrée de table de routage (sensible à la source) est notée $r = ((D,S),NH)$. Elle est constituée d'un préfixe destination D, d'un préfixe source S et d'un *next-hop NH*. Toutefois, la plupart du temps, nous ne manipulons que les préfixes formant la "zone" des paquets routés, c'est-à-dire la paire (D,S). Lorsque cela n'est pas nécessaire, nous pourrons omettre le *next-hop* et écrire $r = (D,S)$. Nous écrivons $\mathrm{nh}(r)$ pour retrouver le *next hop* d'une entrée r.

Nous étendons aussi la notion d'intersection aux entrées, de sorte que, si $r_1 = ((D_1,S_1),NH_1)$ et $r_2 = ((D_2,S_2),NH_2)$ sont deux entrées, alors $r_1 \cap r_2$ est définie et vaut $(D_1 \cap D_2, S_1 \cap S_2)$. Remarquons qu'il ne s'agit plus que d'une "zone" : le *next-hop* a disparu. De même, nous étendons $\prec$ pour que $r_1 \prec r_2$ ssi $(D_1,S_1) \prec (D_2,S_2)$. Enfin, le minimum de deux entrées est l'entrée (avec le *next-hop*) de paire de préfixes minimale pour $\prec$, c'est-à-dire $\min(r_1,r_2) = r_1$ ssi $r_1 \prec r_2$; rappelons que le minimum existe et est unique si r_1 et r_2 ne sont pas disjointes.

3.2.2.2 Ambiguïtés et conflits

Nous disons qu'une table de transfert est *ambiguë* s'il existe un paquet pour lequel l'ensemble des entrées de la table routant ce paquet possède plusieurs minima pour $<$. Des entrées sont dites *en conflit*, lorsqu'elles sont incomparables deux à deux et d'intersection non vide. L'ensemble des paquets qui sont routés par les entrées en conflit s'appelle *zone de conflit* : il s'agit de leur intersection.

Par exemple, les routes r_1 et r_2 de la figure 3.7 (tirée de la section 3.1.2) sont en conflit, puisque nous n'avons ni $r_1 <= r_2$ ni $r_2 <= r_1$. La zone de conflit est $r_1 \cap r_2 = (2001{:}db8{::}/32, 2001{:}db8{::}/32)$. Il n'y a pas d'entrée plus spécifique à la fois que r_1 et r_2 pour router les paquets dans la zone de conflit : la table est ambiguë. Remarquons que $r_1 \cap r_2$ est encore une paire de préfixes et peut donc être insérée dans la table de transfert (avec un *next-hop*). Si nous le faisons, la table n'est plus ambiguë.

	destination	source	*next-hop*
(r_1)	::/0	2001:db8::/32	$A \rightarrow$ "Internet"
(r_2)	2001:db8::/32	::/0	$A \rightarrow B$
($r_1 \cap r_2$)	2001:db8::/32	2001:db8::/32	...

FIGURE 3.7 – Table ambiguë.

Deux entrées $r_1 = (D_1,S_1)$ et $r_2 = (D_2,S_2)$, sont en conflit, noté $r_1 \# r_2$, si et seulement si elles ne sont ni disjointes, ni comparables. Parce que deux préfixes sont soit l'un inclus dans l'autre, soit disjoints, on vérifie que r_1 et r_2 sont en conflit si et seulement si $D_1 < D_2$ et $S_1 > S_2$ ou $D_1 > D_2$ et $S_1 < S_2$. Il suffit alors de considérer les entrées deux à deux pour considérer toutes les zones de conflit. Pour preuve, considérons trois entrées r_1, r_2 et r_3 en conflit. Supposons que $D_1 < D_2 < D_3$ (ce qui est vrai à renommage près) ; alors, par définition du conflit, $S_1 > S_2 > S_3$, et donc $r_1 \cap r_2 \cap r_3 = r_1 \cap r_3$. En conclusion, il suffit des deux entrées r_1 et r_3 pour trouver la zone de conflit.

Enfin, la borne inférieure de deux paires de préfixes existe et est encore une paire de préfixes, puisqu'il s'agit de leur intersection. Elle peut donc être insérée dans la table de transfert, comme nous l'avions fait remarquer dans notre exemple.

3.2.2.3 Complétude d'une table

Nous développons dans cette partie une notion de complétude d'une table. Cette notion est importante, car une table complète est non ambiguë : notre algorithme maintient la FIB complète en ajoutant des entrées

dites de *complétion*. Par ailleurs, nous montrons aussi que compléter la table ne crée pas de conflits : il suffit de considérer tous les conflits présents sans les entrées de complétion et de les résoudre par complétion pour avoir une table complète et donc sans ambiguïté.

Complétude faible Une table de transfert est *faiblement complète* lorsque chaque zone de conflit est couverte par des entrées plus spécifiques. Plus formellement, une table sensible à la source T est faiblement complète si $\forall r_1, r_2 \in T, r_1 \cap r_2 = \bigcup\{r \in T \mid r \leqslant r_1 \cap r_2\}$.

Theorème 1 *Une table de transfert est sans ambiguïté si et seulement si elle est faiblement complète.*

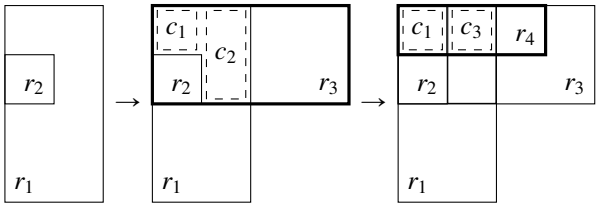
Démonstration 1 *Soit $U_x^y = \bigcup\{r \in T \mid r \leqslant x \cap y\}$. Montrons que T est non ambiguë si et seulement si $\forall r_1, r_2 \in T, r_1 \cap r_2 = U_{r_1}^{r_2}$.*

($\Leftarrow$) Supposons T faiblement complète, et considérons deux entrées $x, y \in T$ et un paquet p routé par ces deux entrées. Par complétude faible, on a $U_x^y = x \cap y$, donc il existe une entrée $r \in \{r \in T \mid r \leqslant x \cap y\}$ routant p. Comme $r \leqslant x \cap y$, on a $r \leqslant x$ et $r \leqslant y$. Ceci étant vrai pour toute paire d'entrées et pour tout paquet, on en déduit qu'il existe bien un minimum pour chaque ensemble d'entrées routant un paquet : la table n'est pas ambiguë.

($\Rightarrow$) Supposons que T ne soit pas faiblement complète. Alors il existe deux entrées $x, y \in T$ en conflit telles que $x \cap y \neq U_x^y$. Considérons un paquet $p \in x \cap y \setminus U_x^y$, et une entrée $r \in T$ routant p. Clairement, $\neg(r \subseteq x \cap y)$, et donc soit $r \# x$, soit $r \# y$, soit $r \geqslant x$ et $r \geqslant y$. Dans tous les cas, r n'est pas plus spécifique que x et que y, et il n'y a donc pas de minimum pour l'ensemble des entrées routant p. On en déduit que T est ambiguë. Par contraposée, si T est non ambiguë, alors elle est faiblement complète.

La levée d'ambiguïté utilisant la complétude faible n'est pas pratique, car d'une part elle nécessite d'ajouter plusieurs entrées pour résoudre un unique conflit, et d'autre part l'ajout d'entrées peut générer de nouveaux conflits. Supposons par exemple (comme illustré ci-après) que la FIB contienne deux entrées $r_1 > r_2$, et que nous ajoutons $r_3 > r_2$, en conflit avec r_1. Étant donné que $r_2 < r_3$, il n'y a pas de conflit avec r_2, mais nous avons besoin des entrées additionnelles c_1 et c_2. La FIB est maintenant faiblement complète.

Supposons maintenant que nous ajoutons $r_4 < r_3$ en conflit avec r_1 et l'entrée additionnelle c_2. Nous devons installer une nouvelle entrée additionnelle c_3. De plus, l'entrée c_1 doit être modifiée dans le cas où elle porte le *next-hop* de r_3 : puisque $r_4 < r_3$, on s'attend à ce que le *next-hop* de r_4 soit alors utilisé pour router les paquets dans c_1.



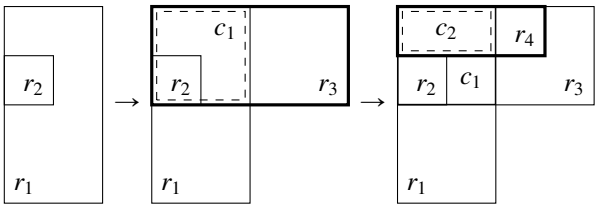
Une certaine part de cette complexité peut être éliminée avec une notion plus forte de complétude.

Complétude Une table de transfert est *(fortement) complète* si chaque zone de conflit est entièrement couverte par une seule entrée. Plus formellement, T est complète si $\forall r_1, r_2 \in T, r_1 \cap r_2 \in T$. Cela implique clairement la complétude faible : une table de transfert complète n'est donc pas ambiguë. Notre algorithme maintient la complétude d'une table de transfert ; nous appelons *entrées de complétion* les entrées additionnelles insérées par l'algorithme.

Theorème 2 *L'ajout d'entrées dans la table de transfert en vue de la complétude ne conduit pas à un* nouveau *conflit.*

Démonstration 2 *Supposons que $r_1 = (D_1, S_1)$ et $r_2 = (D_2, S_2)$ sont deux entrées en conflit telles que $D_1 < D_2$ et $S_1 > S_2$. Considérons l'entrée de complétion $r_{sol} = (D_1, S_2)$ qui résout ce conflit. Supposons maintenant que r_{sol} est en conflit avec une autre entrée $r_3 = (D_3, S_3)$. Nous avons soit $D_1 < D_3$ et $S_1 > S_2 > S_3$, en quel cas $r_3 \# r_1$; soit $D_2 > D_1 > D_3$ et $S_2 < S_3$, en quel cas $r_3 \# r_2$. Dans les deux cas, le conflit existait déjà auparavant : il a déjà dû être résolu.*

Considérons encore l'exemple précédent, cette fois avec la complétude forte (illustré ci-après). Lors de l'ajout de r_3, nous ajoutons une entrée de complétion pour couvrir $c_1 = r_1 \cap r_3$. Comme r_2 est plus spécifique, la nouvelle entrée ne change pas la décision de transfert pour les paquets dans r_2. Lorsque nous ajoutons r_4, celle-ci est en conflit avec r_1 et c_1, mais pour la même zone de conflit $r_4 \cap r_1$. L'entrée de complétion ne donne ainsi pas lieu à un nouveau conflit.



3.2.3 Algorithme de levée d'ambiguïtés

Nous décrivons dans cette section l'algorithme lui-même. Nous notons $<$ l'ordre choisi par le protocole de routage. L'ordre de spécificité $<$ n'apparaît pas clairement, car il est caché dans les notions de conflit et d'intersection (borne inférieure) définies précédemment.

Avant d'énoncer l'algorithme de levée d'ambiguïté lui-même, nous commençons par une partie de préliminaires : définition des fonctions auxiliaires, notations pour les fonctions de la FIB et la définition d'une classe d'équivalence de conflits. À l'aide de cette classe, nous définissons un ensemble C' qui contient exactement les conflits que nous voulons traiter dans notre algorithme.

Dans une deuxième partie, nous décrivons l'algorithme lui-même. Notre algorithme décrit comment ajouter, supprimer ou modifier des entrées dans la FIB. Il parcourt pour cela une table T qui correspond aux entrées de la RIB qui ont été installées. Notre algorithme définit le moment où la route à ajouter ou supprimer peut effectivement l'être dans T. Aucune entrée de complétion n'est retenue dans T : elles sont simplement installées dans la FIB.

3.2.3.1 Préliminaires

Nous définissons dans cette partie les primitives de manipulation de la FIB que nous utilisons, deux fonctions auxiliaires de recherche dans T, et une relation d'équivalence entre deux routes pour éviter de considérer plusieurs fois la même zone de conflit. Nous utilisons cette relation d'équivalence pour extraire dans l'ensemble C' les entrées que nous devons considérer pour maintenir la FIB complète lors de l'ajout, la suppression ou la modification d'une entrée r.

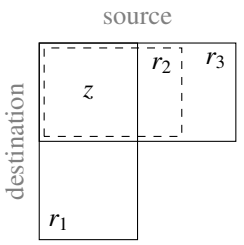
Primitives de la FIB Les FIB se manipulent par des primitives incrémentales. L'API minimale permet d'ajouter et de supprimer une entrée, mais il est parfois possible de modifier une entrée. Notre algorithme fonctionne pour les trois opérations.

Chaque opération prend en compte les préfixes source et destination ainsi qu'un (ou plusieurs pour la modification) *next-hop*. Là encore, nous voulons pouvoir spécifier les préfixes en termes de zones définies par des entrées : nous dissocions donc les routes de leur *next-hop*. Soit $r = ((D, S), NH)$ une entrée, et NH', NH'' deux *next-hop*. Alors :

— install(r, NH') ajoute l'entrée $((D,S), NH')$,
— uninstall(r, NH') supprime l'entrée $((D,S), NH')$, et
— switch(r, NH', NH'') remplace l'entrée $((D,S), NH')$ de la FIB par $((D,S), NH'')$.

L'appel à switch(r, NH', NH'') est un équivalent atomique aux appels successifs de uninstall(r, NH') et install(r, NH'').

Fonctions auxiliaires Nous définissons deux fonctions auxiliaires. La fonction min_conflict$(zone, r)$ (Algorithme 5) renvoie, si elle existe, l'entrée minimale (pour $<$) en conflit avec r pour la zone de conflit *zone*. Considérons par exemple la table décrite en figure 3.8 : r_1 est en conflit à la fois avec r_2 et avec r_3 pour la même zone $z = r_1 \cap r_2 = r_1 \cap r_3$. Or $r_2 < r_3$, donc min_conflict$(z, r_1) = r_2$. Cette fonction nous sert abondamment, entre autres pour éviter de traiter deux fois le même conflit.



$$\text{min\_conflict}(z, r_1) = \min_{<}(r_2, r_3) = r_2$$

$$\text{conflict\_solution}(z) = \min(r_1, r_2, r_3) = r_2$$

FIGURE 3.8 – Exemple : table à 3 entrées.

La fonction conflict_solution$(zone)$ (Algorithme 6) renvoie, si elle existe, l'entrée minimale (pour $\prec$) participant à un conflit pour la zone *zone*. Nous appelons cette entrée *solution* du conflit, car c'est le *next-hop* de cette entrée qui doit être utilisé pour résoudre le conflit. Considérons la zone z de l'exemple précédent (figure 3.8). Trois entrées participent au conflit : r_1, r_2, et r_3. L'entrée r_1 a une destination moins spécifique que r_2 et r_3, et r_2 est plus spécifique que r_3, d'où $r_2 \prec r_3 \prec r_1$. Nous avons donc conflict_solution$(zone) = r_2$. Cette fonction ne sert qu'à détecter si une entrée résout un conflit lors de son ajout ou de sa suppression. Dans le cas de l'ajout, il ne faut alors pas *ajouter* cette entrée mais *modifier* l'entrée de complétion existante par la nouvelle entrée. De même, dans le cas de la suppression, il ne faut pas supprimer l'entrée mais la modifier par une entrée de complétion.

```
1 Function min_conflict(zone, r)
2     min ← ⊥
3     for all r1 ∈ T
4     s.t. r ≠ r1 and r ∩ r1 = zone
5         min ← min(r1, min)
6     return min
```

Algorithme 5 : Recherche de l'entrée minimale en conflit avec r pour *zone*.

```
1 Function conflict_solution(zone)
2     min ← ⊥
3     for all r1, r2 ∈ T
4     s.t. r1 ≠ r2 and r1 ∩ r2 = zone and
        r1 ≺ r2
5         min ← min(r1, min)
6     return min
```

Algorithme 6 : Recherche d'une solution à un conflit.

Conflits pertinents Considérons une entrée r, et un ensemble E d'entrées en conflit avec r pour la même zone de conflit. Remarquons que tous ces conflits ont la même solution. Si la solution est dans E, alors il s'agit nécessairement du minimum de E. Le minimum existe puisque les éléments de E ont soit la même destination, soit la même source, et routent au moins une adresse dans r.

Par exemple, sur la figure 3.9, r est en conflit avec $r_1, r_2, ...$ pour la zone de conflit $r \cap r_1 = r \cap r_2 = ...$; nous avons $E = \{r_i\}$. Or, nous voyons que $r_1 < r_2 < ...$, soit $\min(E) = r_1$.

Puisque tous les r_i ont la même intersection avec r_1, il est suffisant, pour capturer tous les conflits, de ne considérer qu'un seul des r_i. Or, le seul des r_i qui peut nous intéresser, pour en extraire le *next-hop*,

est le plus spécifique : r_1. En effet, dans notre exemple, l'entrée de complétion doit bien avoir $\mathrm{nh}(r_1)$ pour *next-hop*.

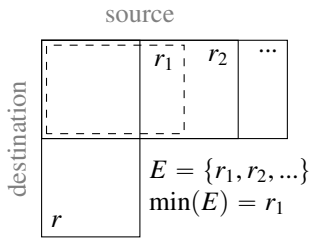


FIGURE 3.9 – Conflit minimum.

Étant donné une entrée r, nous définissons l'équivalence $\sim_r$ par $r_1 \sim_r r_2 \Leftrightarrow r_1 \cap r = r_2 \cap r$: deux entrées sont équivalentes pour $\sim_r$ si elles ont la même intersection avec r. Cela signifie que, si deux entrées sont en conflit avec r, alors elles ont la même zone de conflit. Tous les r_i de l'exemple précédent appartiennent à la même classe d'équivalence pour $\sim_r$.

Dans la suite, r est l'entrée à ajouter, supprimer ou modifier. Quotienter un ensemble d'entrées en conflit avec r par cette équivalence, et prendre le minimum des classes d'équivalences obtenues nous donne ainsi exactement les entrées qui nous intéressent. Nous notons C l'ensemble des entrées en conflit avec r pour lesquelles il n'y a pas de solution naturelle, i.e. $C = \{r' \in T \mid r' \# r \text{ and } r' \cap r \notin T\}$. Sur cet ensemble, nous ne gardons que les conflits minimaux $C' = \{\min(E) \mid E \in C/_{\sim_r}\}$. Il nous suffit de comparer r aux entrées de C' pour capturer tous les conflits engendrés par r.

Cas possibles de conflits Nous énumérons ici tous les cas possibles de conflits que nous rencontrons dans l'algorithme, et donnons leur solution. Nous considérons pour cela deux entrées r et r_1 en conflit tel que r_1 soit la plus petite entrée en conflit avec r pour la zone $r \cap r_1$. En fait, r correspond à l'entrée que nous cherchons à ajouter, supprimer ou modifier et r_1 à une entrée de C'.

Nous illustrons notre analyse de cas sur la figure 3.10, où nous voyons apparaître une organisation en sous-cas. Nous distinguons quatre cas et sous-cas imbriqués décrits ci-dessous. Nous les appelons *u* (unique), *dd* (divers disqualifiés), *dcs* (divers candidats spécifiques) et *dcd* (divers candidats disqualifiés).

Dans le premier cas (*u*), r est nécessaire à la formation de la zone de conflit (c'est-à-dire qu'il n'existe pas de r_2 tel que $r_2 \# r_1$ et $r_2 \cap r_1 = r \cap r_1$). Si r est ajoutée ou supprimée, il en est de même pour l'entrée de complétion. Dans les trois autres cas, où le conflit pour $r \cap r_1$ existe sans r, l'entrée de complétion est modifiée (ou inchangée).

Dans le second cas (*dd*), $r_1 \prec r$: l'ajout, la suppression ou la modification de r n'a aucun impact sur la résolution de ce conflit. En effet, le *next-hop* correspondant est $\mathrm{nh}(r_1)$, indépendamment des autres entrées avec lesquelles r_1 puisse être en conflit pour cette zone. Dans les deux autres cas, $r \prec r_1$: le *next-hop* de r est susceptible d'être sélectionné pour lever l'ambiguïté. Il est donc nécessaire de trouver r_2, l'entrée minimale en conflit avec r_1 pour $r \cap r_1$ différente de r, et de la comparer à r.

Dans le troisième cas (*dcs*), $r \prec r_2$: l'entrée r est le minimum des entrées en conflit avec r_1 pour $r \cap r_1$. Son *next-hop* ($\mathrm{nh}(r)$) est utilisé pour l'entrée de complétion. Dans ce cas, l'ajout, la suppression ou la modification de r implique une modification de l'entrée de complétion : sans r, celle-ci a le *next-hop* de r_2.

Enfin, dans le quatrième cas (*dcd*), $r \succ r_2$. Comme pour le deuxième cas, r n'a aucune influence sur l'entrée de complétion qui porte le *next-hop* de r_2.

Remarque : le cas de conflit où deux entrées r et r_1 sont en conflit mais où il existe $r_2 = r \cap r_1$ est déjà pris en compte à travers C'. Ce n'est donc pas un cas que nous rencontrons.

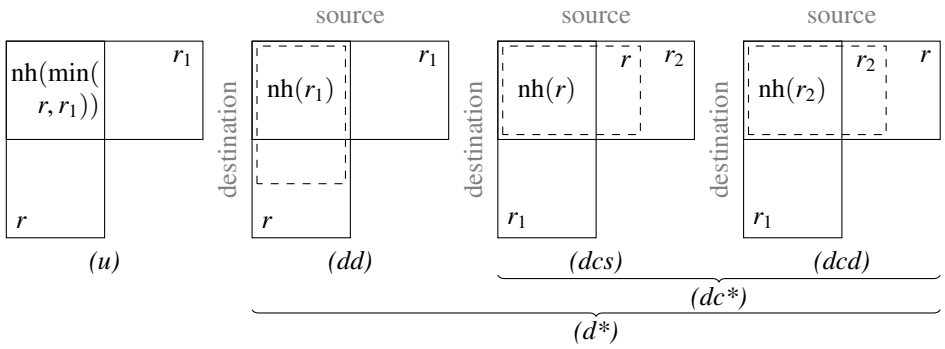


FIGURE 3.10 – Cas possibles de conflits.
(u) : seuls r et r_1 sont en conflit pour cette zone ; (d) : r_1 est en conflit avec une autre entrée r_2 pour cette zone ; (dd) : $r_1 \prec r$; (dc*) : $r \prec r_1$; (dcs) : $r \prec r_2$; (dcd) : $r_2 \prec r$.*

3.2.3.2 Algorithme

Nous proposons en fait trois algorithmes. Chacun correspond à une opération incrémentale de modification de la FIB. D'un point de vue du protocole de routage, il s'agit de l'ajout, de la suppression et de la modification d'une entrée r à la table T (qui, rappelons-le, est l'ensemble des entrées de la RIB qui ont été installées). Chaque algorithme décrit quand l'entrée r peut être ajoutée, supprimée ou modifiée dans T. Chacun maintient la FIB complète tout au long de sa procédure (en supposant qu'elle est complète au début).

Nos algorithmes utilisent les notations et fonctions auxiliaires utilisées dans les préliminaires. Nous utilisons aussi C' comme défini dans les préliminaires : C' est l'ensemble des entrées minimales en conflit avec r pour lesquelles il n'y a pas de solution naturelle. Enfin, nous continuons de nous référer aux cas décrits dans les préliminaires (figure 3.10).

Ajout d'une entrée (Algorithme 7) Nous avons montré (section 3.2.2.3) qu'ajouter des entrées de complétion ne conduit pas à un nouveau conflit. Il nous suffit donc de nous assurer qu'une entrée de complétion avec le bon *next-hop* couvre chaque conflit entre r et les entrées déjà dans T. Nous parcourons pour cela C' (lignes 2 à 4), et testons tous les cas possibles de conflits énumérés aux préliminaires. Une fois tous les conflits résolus, nous ajoutons r.

Afin de garder la FIB sans ambiguïté tout au long du processus, nous installons les entrées les plus spécifiques en premier. En particulier, nous commençons par installer les entrées de complétion (lignes 2 à 9) avant l'entrée elle-même (lignes 10 à 14).

Pour chaque entrée $r_1 \in C'$ (en commençant par l'entrée la plus spécifique), nous cherchons d'abord l'entrée minimale r_2 en conflit avec r_1 pour $r \cap r_1$ (ligne 5). Ceci nous permet de nous rattacher aux quatre cas de conflits (figure 3.10) :

— *(u)* : r_2 n'existe pas (ligne 6). Il n'y a pas (encore) de conflit pour cette zone : nous devons ajouter à la FIB une entrée pour la zone $(r_1 \cap r)$ avec le *next-hop* de la plus petite des deux entrées r et r_1 (ligne 7).

— *(dcs)* : r_2 existe et r est inférieure à r_2 et à r_1 (ligne 8). L'entrée r devient la nouvelle solution de la zone de conflit, à la place de r_2. Le *next-hop* installé pour la zone de conflit étant celui de r_2, nous devons remplacer $((r_1 \cap r), NH_2)$ par $((r_1 \cap r), NH)$ (ligne 9).

— *(dd, dcd)* : dans ces deux cas, une entrée de complétude avec le bon *next-hop* est déjà installée : il n'y a qu'à passer au conflit suivant.

Enfin, avant d'installer r, nous devons chercher s'il existe deux entrées en conflit pour la zone de r (ligne 10). Dans ce cas, une entrée de complétion est installée, et r doit la remplacer (ligne 12). Autrement, r peut

être ajoutée normalement (ligne 14). Nous terminons la procédure en ajoutant r à la RIB (ligne 15).

```
1  Function add_route(r)
2      for all r1 ∈ T
3      s.t. r # r1 and r ∩ r1 ∉ T
4      and r1 = min_conflict(r ∩ r1, r)
5          r2 ← min_conflict(r ∩ r1, r1)
6          if r2 = ⊥
7              install(r ∩ r1, nh(min(r, r1)))
8          else if r ≺ r2 and r ≺ r1
9              switch(r ∩ r1, nh(r2), nh(r))
10     r1 ← conflict_solution(r)
11     if r1 = ⊥
12         install(r, nh(r))
13     else
14         switch(r, nh(r1), nh(r))
15     T ← T ∪ {r}
```

Algorithme 7 : Ajout d'une entrée.

```
1  Function delete_route(r)
2      T ← T ∖ {r}
3      r2 ← conflict_solution(r)
4      if r2 = ⊥
5          uninstall(r, nh(r))
6      else
7          switch(r, nh(r), nh(r2))
8      for all r1 ∈ T
9      s.t. r # r1 and r ∩ r1 ∉ T
10     and r1 = min_conflict(r ∩ r1, r)
11         r2 ← min_conflict(r ∩ r1, r1)
12         if r2 = ⊥
13             uninstall(r∩r1, nh(min(r, r1)))
14         else if r ≺ r2 and r ≺ r1
15             switch(r ∩ r1, nh(r), nh(r2))
```

Algorithme 8 : Suppression d'une entrée.

À présent, toute nouvelle zone de conflit est complétée, et toute zone de conflit où r est solution est couverte par une entrée de complétion avec le *next-hop* de r. La FIB est complète et a le même comportement que T.

Suppression d'une entrée (Algorithme 8) La suppression d'une entrée se fait inversement à son ajout. Nous commençons par supprimer r de T (ligne 2). Comme pour l'ajout, il se peut que r résolve un conflit. Si ce n'est pas le cas, nous supprimons simplement r de la FIB (ligne 5). Autrement, nous cherchons l'entrée r_2 qui résout le conflit de zone r (ligne 3) et changeons r par une entrée de complétion ayant le *next-hop* de r_2 (ligne 7).

Nous nous attaquons ensuite aux entrées de complétion en commençant par les moins spécifiques, toujours pour garder la FIB sans ambiguïtés. Comme précédemment, nous parcourons C' (lignes 8 à 10). Pour chaque entrée $r_1 \in C'$ (en commençant par les moins spécifiques), nous cherchons, comme pour l'ajout, l'entrée minimale r_2 en conflit avec r_1 pour $r \cap r_1$ (ligne 11). Si r_2 n'existe pas (cas *u*), nous supprimons $((r_1 \cap r), NH)$ de la FIB (ligne 13). Autrement, pour les mêmes raisons que ci-dessus, si r est inférieure à r_1 et r_2 (cas *dcs*), alors nous remplaçons $((r_1 \cap r), NH)$ par $((r_1 \cap r), NH_2)$ dans la FIB (ligne 15). Il n'y a qu'à passer au conflit suivant dans les autres cas (*dd* et *dcd*).

Modification d'une entrée (Algorithme 9) Nous décrivons le cas où r est remplacée par une entrée que nous appelons r_{new}. Il s'agit du cas le plus simple car les entrées de complétion sont maintenues. Elles ne sont modifiées que lorsque l'entrée que nous voulons modifier est choisie pour résoudre une ambiguïté (cas *dcs*).

L'ordre dans lequel les entrées sont modifiées n'a aucune importance : la FIB reste toujours complète. Ici, nous choisissons de remplacer d'abord r par r_{new} (ligne 2).

Nous parcourons ensuite C' comme précédemment (lignes 3 à 5). Seuls le cas *(u)* lorsque $r \prec r_1$ et le cas *(dcs)* nous intéressent. Dans ces deux cas (et contrairement aux autres), $r \prec r_1$ et r est l'entrée minimale en conflit avec r_1 pour $r \cap r_1$. C'est pourquoi nous parcourons C' à la recherche des entrées r_1 satisfaisant ces conditions (ligne 6). Nous remplaçons alors le *next-hop NH* de l'entrée de complétion correspondante par le nouveau *next-hop* NH_{new}.

1 **Function** change_route(r, r_{new})
2 switch(r, nh(r), nh(r_{new}))
3 **for all** $r_1 \in T$
4 *s.t.* $r \# r_1$ *and* $r \cap r_1 \notin T$
5 *and* $r_1 = \text{min\_conflict}(r \cap r_1, r)$
6 *and* $r \prec r_1$ *and*
$r = \text{min\_conflict}(r \cap r_1, r_1)$
7 switch($r \cap r_1$, nh(r), nh(r_{new}))

Algorithme 9 : Modification d'une entrée.

3.2.4 Comparaison de la levée d'ambiguïté avec la vérification des pare-feux

Nous avons réalisé ces travaux dans le cadre du routage, avec pour première application les réseaux superposés (*overlay*) puis pour principale application les réseaux *multihomés*. Il s'avère que des travaux similaires ont été menés dans le cadre de la vérification des bases de données des pare-feux (*firewalls*) [HSP00, LS05]. Dans cette partie, nous décrivons ces travaux et montrons à la fois l'étonnante similitude de nos approches et les différences de nos problématiques. En effet, leur but n'est pas de résoudre automatiquement les conflits, mais de les détecter pour en avertir l'administrateur. Leurs techniques ne suffisent donc pas à résoudre notre problème.

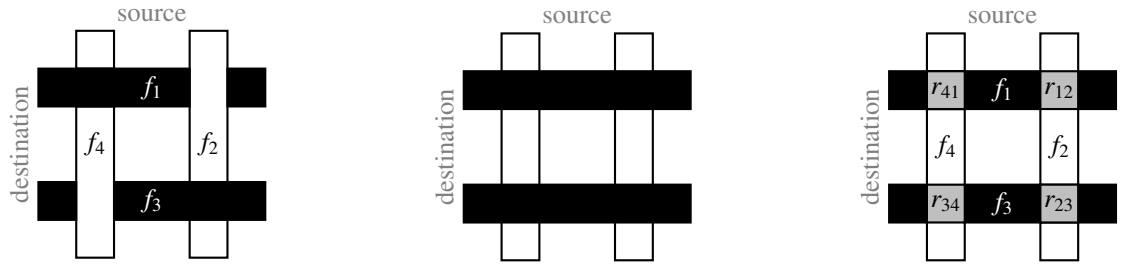
Tables de règles et filtres de paquets Les pare-feux utilisent des tables dont chaque *règle (rule)* associe une *action* à un ensemble de paquets IP, à l'image d'une entrée d'une table de transfert. Ces tables, appelées tables de règles (*rule table*, ou table de routeur — *router table*) sont donc des ensembles de règles (f, a) où f est un ensemble de paquets et a une action. Les ensembles de paquets sont appelés dans ce contexte *filtres de paquets (packet filter)*. Un filtre est un n-uplet dont chaque composante correspond à un champ d'un paquet ; un paquet correspond à un filtre si chaque champ du paquet correspond à la composante du filtre associée.

Par exemple, si une table associe une action à une paire de préfixes destination et source, un filtre est de la forme $f = (D, S)$ avec D un préfixe destination et S un préfixe source. L'entrée $((dst = 2001{:}db8{:}1{::}/48, src = 2001{:}db8{:}2{::}/48), a = \text{“}deny\text{''})$ détruit les paquets à destination de 2001:db8:1::/48 et en provenance de 2001:db8:2::/48.

Un filtre peut concerner divers champs de la couche réseau mais aussi transport. Il est assez fréquent de considérer les adresses IP source et destination, le type de protocole de couche transport (TCP ou UDP), et les ports source et destination de couche transport : on parle communément du 5-uplet (adresse destination, adresse source, protocole, port destination et port source). Les ensemble d'adresses destination et source sont naturellement exprimés en terme de préfixes car ceux-ci représentent les sous-réseaux auxquels ils sont attribués.

Ambiguïté dans la classification des paquets Le problème de la *classification des paquets (packet classification)* consiste, étant donné une table de règles, à déterminer quelle règle doit être appliquée à un paquet. Hari et al. [HSP00] soulèvent le problème que deux filtres de paquets peuvent conduire à une *ambiguïté* pour la classification des paquets. C'est le cas lorsque les filtres s'expriment avec des préfixes qui se chevauchent : il est possible qu'un paquet corresponde à plusieurs filtres. On parle alors de *conflit* entre les deux filtres.

Il existe plusieurs méthodes arbitraires pour résoudre ces conflits. Celles décrites par Hari et al. [HSP00] sont similaires aux algorithmes proposés dans la section 3.1 (figure 3.3). La première est de sélectionner le premier filtre satisfaisant de la table (algorithme 3) — ou de manière équivalente d'affecter explicitement des priorités aux filtres. La deuxième est d'affecter des priorités aux champs des filtres, ce qui est équivalent



(a) Disposition possible dans un pare-feu.

(b) Ordonnées par la destination d'abord.

(c) Ajout des filtres de résolution.

FIGURE 3.11 – Quatre filtres de paires de préfixes destination-source.

à linéariser les ordres sur les préfixes par la destination d'abord ou par la source d'abord (algorithmes 1 et 2).

Toutefois, dans le contexte de leur article, ces méthodes arbitraires ne sont pas toujours satisfaisantes. La figure 3.11*a* illustre le problème qu'ils cherchent à résoudre. Quatre filtres $f_1 \dots f_4$ sont tels que [1] f_i et f_{i+1} sont en conflit et que le filtre f_{i+1} est choisi pour résoudre le conflit entre f_i et f_{i+1}. Ordonner les règles permet de satisfaire tous les conflits sauf un : par exemple $f_1 < f_2 < f_3 < f_4$, mais pas $f_4 < f_1$. Affecter des priorités aux champs ne permet pas non plus d'obtenir le résultat escompté. La figure 3.11*b* montre qu'en prioritisant le préfixe destination, on obtient $f_4 < f_1$ et $f_2 < f_3$, ce qui est désiré, mais aussi $f_1 > f_2$ et $f_3 > f_4$, ce qui ne l'est pas.

Afin de pouvoir résoudre ces conflits, Hari [HSP00] introduit la notion de *filtre de résolution (resolve filter)*. Ces filtres sont les intersections de deux filtres en conflit. Dans notre exemple, il faut rajouter les quatre filtres de résolution représentés sur la figure 3.11*c* : $r_{12} = f_1 \cap f_2$, r_{23}, r_{34}, et r_{41}.

Différences avec notre problème Dans le cas de la classification des paquets, seul l'administrateur sait comment devrait se comporter sa table de règles. Il n'est donc pas possible de choisir automatiquement une action pour un filtre donné. L'algorithme d'ajout de filtres de résolution donné par Hari et al. [HSP00] ne parle d'ailleurs pas de l'action à choisir pour les filtres : celle-ci est simplement ignorée.

En fait, la réelle motivation de ces travaux est de pouvoir *détecter* les conflits et ainsi de potentielles failles de *sécurité*. Aussi, Hari et al. fournissent un algorithme avancé (FastDetect) pour la détection des conflits. L'algorithme renvoie un ensemble de filtres de résolution qui devrait être utilisé pour compléter la table. À sa suite, d'autres algorithmes ont été développés pour la détection des conflits, dont celui de Lu et al. [LS05]. Tous les résultats expérimentaux que nous avons trouvé ne traitent que du problème de la détection des conflits.

Au contraire, dans notre approche, les entrées de la RIB sont ordonnées par un ordre fixé par le protocole de routage. La levée d'ambiguïté a pour but de forcer une FIB munie d'un ordre différent à se comporter comme la RIB pour le transfert des paquets. La levée d'ambiguïté est donc une opération entièrement automatisable car la RIB indique sans ambiguïté quel *next-hop* doit être utilisé pour chaque entrée de complétion. L'algorithme que nous fournissons maintient complète la table en choisissant le *next-hop* donné par l'ordre de la RIB.

Par ailleurs, les algorithmes de détection ne sont pas créés pour remplir automatiquement la table des règles. Souvent, seule l'opération *d'ajout* de filtres de résolution est renseignée ; pas celle de suppression ni celle de modification. L'action à associer à un filtre de résolution n'est pas déterminée, et l'ordre dans lequel les filtres de résolution doivent être ajoutés pour maintenir la table cohérente n'est pas précisé. Enfin, les algorithmes de calcul des filtres de résolution peuvent ajouter plusieurs fois le même filtre (comme c'est

1. Indices modulo 4.

le cas avec l'algorithme de Hari) et même contenir un filtre déjà présent dans la table des règles. L'union mathématique efface naturellement ces doublons.

Notons aussi que les algorithmes de détection des conflits utilisent des structures de données supplémentaires (typiquement deux *tries* — ou *arbres préfixe*). Notre algorithme n'utilise aucune structure de données supplémentaire, mais calcule au fur et à mesure les entrées de complétion à ajouter, supprimer ou modifier. Il maintient la FIB complète tout au long de l'opération en choisissant d'ajouter les entrées les plus spécifiques d'abord (et au contraire en supprimant les entrées les moins spécifiques).

Similitudes avec notre problème Le problème de l'ambiguïté des tables de transfert auquel nous nous intéressons est similaire à celui de l'ambiguïté des tables de règles que nous venons de décrire. D'une manière encore plus étonnante, la terminologie utilisée par Hari et al. [HSP00] est quasiment la même que la nôtre. Nous avons la même notion d'ambiguïté, la même notion de conflit, la même manière de représenter les paires de préfixes par des rectangles [2], nous considérons tous deux les préfixes comme des ensembles d'adresses. Enfin, nous nous plaçons sous la même hypothèse : une entrée (ou règle) inclue dans (ou plus spécifique que) une autre est prioritaire sur celle-ci.

Aussi, nous constatons tous deux que deux paires de préfixes (D_1, S_1) et (D_2, S_2) sont en conflit si et seulement si $D_1 < D_2$ et $S_1 > S_2$ ou l'inverse. Enfin, pour résoudre les conflits, nous optons tous deux pour la même solution : ajouter, pour chaque conflit entre deux entrées r_1 et r_2, une entrée additionnelle à l'intersection des deux entrées : $r_1 \cap r_2$. Pour qualifier cette entrée additionnelle, Hari et al. parlent de *filtre de résolution (resolve filter)* tandis que nous parlons *d'entrée de complétion*. Nous utilisons ce terme du fait que notre solution se base sur une notion de complétude de la table.

Dans leur article, Lu et al. [LS05] introduisent la notion de *filtre de résolution essentiel*. Les filtres essentiels sont les filtres de résolution dont la présence dans la table est nécessaire pour qu'elle ne soit pas ambiguë. Si deux filtres sont en conflit, mais que tous les paquets capturés par ces deux règles sont aussi capturés par un filtre plus spécifique, alors il n'est pas nécessaire d'ajouter un filtre de résolution pour ce conflit. Cette notion de filtre essentiel vient rejoindre notre notion de complétude faible d'une table : l'ajout des filtres essentiels rend la table faiblement complète.

Adaptabilité au routage Nos algorithmes, pour maintenir la table complète, détectent les conflits. Les algorithmes de détection des conflits sont donc adaptables à notre problème. En particulier, il doit être possible de déterminer le *next-hop* à utiliser pour résoudre les conflits. À l'image de l'algorithme de Lu et al., nous pourrions aussi nous limiter au maintient des entrées additionnelles.

La plupart des algorithmes que nous avons trouvé utilisent des structures additionnelles qui augmentent la quantité de mémoire utilisée par l'application, contrairement à notre algorithme. En contrepartie, nous pensons qu'ils sont adaptables sans perte de performance malgré les tâches supplémentaires à fournir. Par exemple, alors que notre algorithme s'exécute en $O(n^2)$, l'algorithme de Lu et al. [LS05] détecte les conflits en $O(n \log(n))$. Quant à l'utilisation de la mémoire, leurs résultats expérimentaux laissent à penser que la mémoire utilisée par leur algorithme prendrait environ la même taille que la RIB de Babel (chapitre 4) : en estimant une entrée de Babel à 136 octets [3], nous obtenons le tableau suivant.

nombre d'entrées	RIB Babel	PlaneSweep	(Notre algorithme)
1000	136 kB	+120 kB	+0 B
30000	4,1 MB	+3,5 MB	+0 B

Toutefois, dans le cadre du *multihoming*, un algorithme en $O(n^2)$ suffit car le nombre d'entrées spécifiques à la source est très limité. Il est possible, avec une répartition intelligente des entrées spécifiques

2. Encore plus étonnant, nous plaçons tous deux le préfixe destination en ordonnée et le préfixe source en abscisse.
3. Somme d'une `struct source` et `struct route`.

à la source dans la RIB, de détecter les conflits très rapidement : une entrée non spécifique ne peut être en conflit qu'avec une entrée spécifique. La complexité de la recherche de conflit devient donc $O(n \cdot m)$, avec n le nombre d'entrées non spécifiques et m le nombre d'entrées spécifiques. Enfin, dans le cadre du *multihoming*, les préfixes source sont disjoints : les entrées additionnelles sont toutes "essentielles".

En conclusion, la solution que nous avons implémentée n'utilise pas de mémoire supplémentaire et est suffisante pour nos besoins. Par ailleurs, à long terme, nous espérons que toutes FIB seront naturellement sensibles à la source avec ordre par la destination d'abord. Aucun algorithme ne sera alors nécessaire pour lever les ambiguïtés.

3.3 Conception d'un protocole sensible à la source

Nous avons vu que, en routage *next-hop*, des protocoles de routage étaient utilisés pour peupler les FIB des routeurs du réseau (section 2.1.3), et nous avons décrit les bases des protocoles à vecteur de distances. Dans cette section, nous adaptons l'algorithme à vecteur de distances au routage sensible à la source et nous montrons comment étendre de manière compatible un protocole à vecteur de distances au routage sensible à la source.

3.3.1 Vecteur de distances sensible à la source

Les protocoles à vecteur de distances sont l'une des deux grandes familles de protocoles de routage. Nous comparons dans cette partie les protocoles *next-hop* et sensibles à la source. Nous concluons que les deux types de protocoles appartiennent à une famille plus générale et qu'il n'y a pas de différence opérationnelle entre les deux. Obtenir un protocole sensible à la source à partir d'un protocole de routage *next-hop* ne nécessite pas de changement de fond.

3.3.1.1 Adaptation de l'algorithme à vecteur de distances

L'algorithme à vecteur de distances que nous avons présenté en section 2.1.3.3 s'applique au routage *next-hop* — nous y manipulons des préfixes destination. Nous avons adapté cet algorithme au routage sensible à la source.

Notations Nous définissons la RIB comme un dictionnaire de triplets $((D,S),m,n)$ indexé par (D,S) avec (D,S), m et n qui sont respectivement une paire de préfixes destination et source, une métrique et un *next-hop*. Nous écrivons :

— $(D,S) \in RIB$ pour tester l'existence d'une entrée indexée par (D,S) ;
— $(m,n) \leftarrow RIB[(D,S)]$ pour retrouver la métrique et le *next-hop* associés à (D,S) ;
— $RIB[(D,S)] \leftarrow (m,n)$ pour ajouter ou remplacer le triplet $((D,S),m,n)$;
— $RIB[(D,S)] \leftarrow \perp$ pour supprimer l'entrée indexée par (D,S) ;
— $((D,S),m,n) \in RIB$ pour itérer sur les entrées de la RIB.

Nous définissons la FIB de manière analogue comme un dictionnaire de paires $((D,S),n)$ indexé par (D,S) avec (D,S) et n qui sont respectivement une paire de préfixes destination et source et un *next-hop*. La modification de la FIB peut nécessiter un algorithme de levée d'ambiguïté tel que celui que nous avons défini dans la section 3.2.

Initialisation

— La RIB contient chaque entrée redistribuée avec une métrique nulle (le *next-hop* n'est pas néces-

saire) ;

```
1 RIB ← {}
2 for all ((D, S), _) ∈ FIB
3 |  RIB[(D, S)] ← (0, ⊥)
```

Transmission

— Régulièrement, chaque entrée de la RIB (sans le *next-hop*) est annoncée à tous ses voisins ;

```
1 for all ((D, S), m, _) ∈ RIB
2 |  send((D, S), m)
```

Réception d'une annonce

Soit $((D,S),m)$ la paire de préfixes et la métrique reçues du voisin n par un lien de coût c :

— la métrique de la route depuis le nœud local est calculée ;

```
1 m' = c + m
```

— si la route est de métrique finie et si la RIB n'a pas d'entrée meilleure pour cette paire de préfixes, l'entrée est ajoutée à la RIB avec n comme *next-hop* ;

```
1 (m_RIB, n_RIB) ← RIB[(D, S)]
2 if m' < ∞ and ((D, S) ∉ RIB or m' < m_RIB or n = n_RIB)
3 |  RIB[(D, S)] ← (m', n)
4 |  FIB[(D, S)] ← (n)
```

— si la route est de métrique finie et qu'une entrée pour cette paire de préfixes est dans la RIB avec n pour *next-hop*, alors la perte de la route est propagée et l'entrée de la RIB est supprimée ;

```
1 (m_RIB, n_RIB) ← RIB[(D, S)]
2 if m' ⩾ ∞ and (D, S) ∈ RIB and n = n_RIB)
3 |  send((D, S), ∞)
4 |  FIB[(D, S)] ← ⊥
5 |  RIB[(D, S)] ← ⊥
```

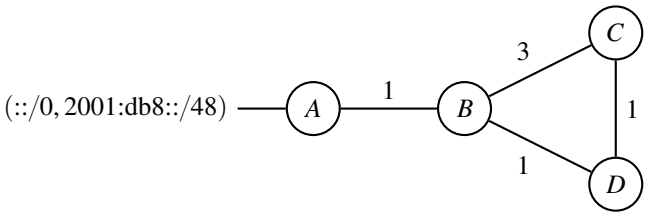
Perte de voisin Si la couche lien indique la perte d'un lien connecté à un voisin n, ou si aucun message n'est reçu d'un voisin n depuis un certain temps (*timeout*), le voisin est déclaré perdu.

Dans ce cas, toutes les entrées qui ont n pour *next-hop* sont supprimées de la RIB et des rétractions sont envoyées pour les préfixes associés à ces entrées.

```
1 for all ((D, S), _, n) ∈ RIB
2 |  if n_perdu = n
3 |  |  send((D, S), ∞)
4 |  |  FIB[(D, S)] ← ⊥
5 |  |  RIB[(D, S)] ← ⊥
```

Les seules différences entre cet algorithme et l'algorithme de base sont dans l'indexation des structures de données, et non sur la partie opérationnelle de l'algorithme. La RIB, la FIB et les fonctions d'envoi d'annonces et de réception d'annonces doivent être paramétrées par une paire de préfixes au lieu d'un préfixe.

La figure 3.12 représente l'exécution de l'algorithme à vecteur de distances sensible à la source pour la paire de préfixes destination et source $(::/0, 2001{:}db8{::}/48)$ redistribuée par A. Elle est strictement identique à la figure 2.13 qui représente l'algorithme à vecteur de distances classique pour le préfixe destination $::/0$: il n'y a que l'intitulé qui change. Nous pouvons donc faire les mêmes observations. Notons $P = (::/0, 2001{:}db8{::}/48)$. Initialement, seul A connaît P. Il annonce P avec une métrique de 0 à B, qui ajoute le coût du lien à la métrique qu'il reçoit pour obtenir une nouvelle métrique de 1. Puisqu'il n'a pas de route pour P,



A	B	C	D
0, **FIB**	$\perp$	$\perp$	$\perp$
0, **FIB**	$1, B \rightarrow A$	$\perp$	$\perp$
0, **FIB**	$1, B \rightarrow A$	$4, C \rightarrow B$	$2, D \rightarrow B$
0, **FIB**	$1, B \rightarrow A$	$3, C \rightarrow D$	$2, D \rightarrow B$

Calcul des métriques et next-hop *pour la paire de préfixes* $(::/0, 2001{:}db8{::}/48)$ *redistribuée par A.*

FIGURE 3.12 – Calcul des RIB du réseau avec l'algorithme à vecteur de distances.

Itération	Nœud	destination	source	métrique	*next-hop*
1	A	::/0	2001:db8::/48	0	**FIB**
	B			0	
	C			0	
	D			0	
2	A	::/0	2001:db8::/48	0	**FIB**
	B	::/0	2001:db8::/48	1	$B \rightarrow A$
	C			0	
	D			0	
3	A	::/0	2001:db8::/48	0	**FIB**
	B	::/0	2001:db8::/48	1	$B \rightarrow A$
	C	::/0	2001:db8::/48	4	$C \rightarrow B$
	D	::/0	2001:db8::/48	2	$C \rightarrow B$
4	A	::/0	2001:db8::/48	0	**FIB**
	B	::/0	2001:db8::/48	1	$B \rightarrow A$
	C	::/0	2001:db8::/48	4	$C \rightarrow D$
	D	::/0	2001:db8::/48	3	$D \rightarrow B$

FIGURE 3.13 – RIB complètes de tous les routeurs pendant la convergence.

il choisit $B \rightarrow A$ pour *next-hop* et ajoute l'entrée $(P, 1, B \rightarrow A)$ dans sa RIB, avant de l'annoncer à ses voisins. Le routeur A ignore l'annonce de B car la métrique de l'annonce $(1 + 1)$ est supérieure à celle de la route qu'il a déjà dans sa RIB (0), tandis que C et D ajoutent une entrée pour cette route. Enfin, quand D annonce sa route et que C reçoit l'annonce, C a une entrée $(P, 4, C \rightarrow B)$ dans sa RIB, mais comme la route proposée par D a une métrique de $(2 + 1)$, C choisit de passer par D. Le réseau est à présent stable et les meilleurs chemins ont été calculés. Lorsqu'un nœud reçoit une annonce, il a une entrée dans sa RIB pour la paire de préfixes associé à cette annonce avec une métrique meilleure (ou égale) à celle de l'annonce. La figure 3.13 représente de manière détaillée l'état de chaque RIB du réseau.

En remplaçant les paires de préfixes (D, S) de cet algorithme par P, et les préfixes D de l'algorithme classique de la section 2.1.3.3 par P, nous obtenons rigoureusement le même algorithme. Dans les deux cas, P représente l'ensemble des paquets acceptés par la route. L'algorithme à vecteur de distances appartient a une famille plus générale où l'indexation des entrées est celle d'un ensemble de paquets P. Celui-ci peut être l'ensemble des paquets correspondant à un préfixe destination, comme en routage *next-hop*, un ensemble de paquets correspondant à une paire de préfixes destination et source, comme en routage sensible à la source, ou tout autre ensemble de paquets, comme par exemple une paire d'un préfixe destination et du *ToS* [CC17].

Nous avons validé expérimentalement cet algorithme en réalisant notre implémentation du routage sen-

sible à la source dans un protocole à vecteur de distances (chapitre 4).

3.3.2 Interopérabilité entre routages *next-hop* et sensible à la source

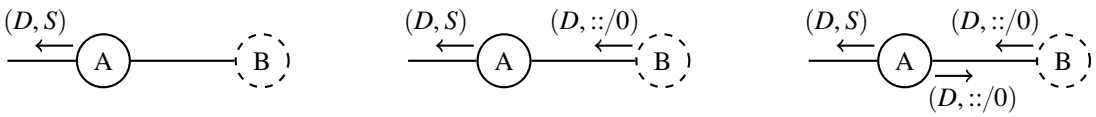
Lors du déploiement d'un nouveau protocole ou d'une nouvelle fonctionnalité, il n'est pas toujours possible ou souhaitable de changer tous les routeurs d'un réseau en même temps. C'est par exemple le cas si l'administration du réseau est décentralisée, ou si certains routeurs ne peuvent être mis à jour. En ce qui concerne le routage sensible à la source, nous attachons de l'importance à l'interopérabilité des routeurs au sens de leur capacité à s'envoyer des annonces de route et autres informations nécessaires au protocole tout en préservant l'absence de boucles et l'arrivée des paquets à leur destination (en l'absence de trou noir).

Conditions de l'interopérabilité L'interopérabilité entre des routeurs classiques et sensibles à la source est possible. Rappelons-nous tout d'abord (section 3.1) qu'une route classique est équivalente à une route spécifique ayant un préfixe source de longueur nulle : les mêmes paquets sont acceptés par les deux routes puisque dans les deux cas toutes les adresses sources sont acceptées. De ce fait, un routeur sensible à la source peut annoncer une route ayant un préfixe source de longueur nulle comme une route non spécifique de la même manière que le protocole de base. Réciproquement, un routeur sensible à la source peut considérer l'annonce d'une route du protocole de base comme l'annonce d'une route spécifique au préfixe source de longueur nulle. Les routeurs classiques et sensibles à la source peuvent ainsi interopérer pour toutes les routes du protocole de base.

En revanche, pour les routes sensibles à la source, aucune interopérabilité n'est possible. Il est évident que les routeurs classiques ne peuvent pas comprendre le préfixe source, propre au routage sensible à la source. Or, la compréhension partielle d'une route sensible à la source est incorrecte : ils doivent aussi ignorer le préfixe destination. En effet, un routeur classique qui ignore le préfixe source et non le préfixe destination d'une annonce sensible à la source peut engendrer des boucles de routage persistantes.

La première cause de boucle est que si un routeur classique ignore uniquement le préfixe source d'une route (D, S), il annonce une nouvelle route $(D, ::/0)$, moins spécifique que la route d'origine. Les routeurs sensibles à la source voient cette annonce comme celle d'une route distincte de celle d'origine. En l'installant, ils routent des paquets supplémentaires vers le routeur classique qui annonce la route, tandis que ce routeur les route à son tour vers un routeur spécifique à la source. Une boucle de routage est installée pour les paquets qui ont leur adresse destination dans D et leur source hors de S.

Considérons par exemple la figure 3.14 : le nœud A est un routeur sensible à la source ayant une route (D, S), qu'il annonce, et le nœud B est un routeur classique. Lorsque B reçoit l'annonce de A, il ignore le préfixe source, installe la route comme étant non spécifique ($(D, ::/0)$), puis l'annonce. Le routeur A reçoit l'annonce de B pour $(D, ::/0)$, différente de la route (D, S) que possède A. Le routeur A installe donc $(D, ::/0)$: les paquets destinés à D ayant leur source hors de S sont transmis de B vers A, et de A vers B. Un routeur classique ne peut donc pas ignorer *uniquement* le préfixe source, mais doit ignorer toute l'annonce.



1. A annonce (D, S) *2. B installe et annonce* $(D, ::/0)$ *3. A installe* $(D, ::/0)$

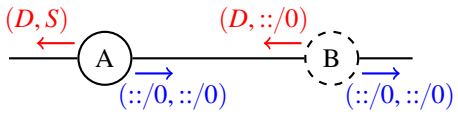
FIGURE 3.14 – Boucle de routage : création d'une nouvelle route.
A est un routeur sensible à la source, B un routeur classique.

Certains protocoles, comme BGP, supportent des attributs "transitifs". Un attribut transitif est un attribut qui peut être ignoré pour la sélection et l'installation de la route, mais qui est répété dans l'annonce de la

route. Dans notre cas, si un routeur classique considérait le préfixe source comme un attribut transitif, il pourrait installer une route (D, S) comme étant $(D, ::/0)$ tout en réannonçant (D, S).

On pourrait tenter de contourner le problème précédent en considérant le préfixe source comme un attribut "transitif". Mais, même dans ce cas, une boucle de routage peut s'installer.

Considérons par exemple la figure 3.15 : les nœuds A et B sont respectivement des routeurs classiques et sensibles à la source et le routage a convergé. Les deux routeurs ont les deux routes $r_1 = (D, S)$ et $r_2 = (::/0, ::/0)$ dans leur table de routage, mais le routeur B les considère toutes deux comme non spécifique : il installe $(D, ::/0)$ mais annonce (D, S). Or, un paquet à destination de D n'ayant pas sa source dans S est routé par $(D, ::/0)$ mais pas par (D, S) : les routeurs A et B utilisent respectivement r_2 et r_1 pour router ce paquet. Il y a clairement une boucle de routage persistante pour les paquets à destination de D n'ayant pas leur source dans S. Le préfixe source ne peut donc pas être traité comme un attribut transitif.



Note : $D \neq ::/0$, sinon B ne pourrait installer qu'une des deux routes.

FIGURE 3.15 – Boucle de routage : modification de l'ensemble des paquets acceptés par une route.
A est un routeur sensible à la source, B un routeur classique.

En conclusion, les annonces de routes sensibles à la source doivent être totalement ignorées par les routeurs classiques. Mais les annonces de routes non sensibles à la source (routes classiques ou routes avec un préfixe source de taille nulle) peuvent être prises en compte et réannoncées. Sur la figure 3.16, tous les routeurs installent la route non spécifique (D), mais seuls les routeurs sensibles à la source installent la route spécifique à la source (D, S) en contournant le routeur classique D.

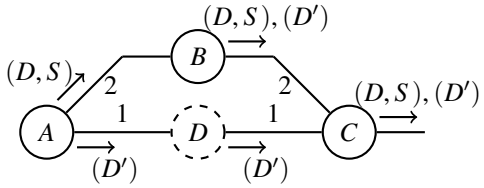
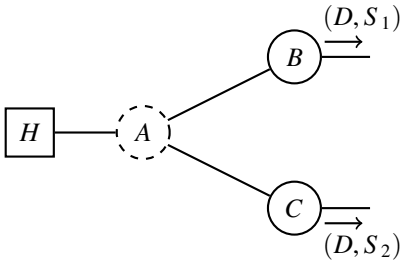


FIGURE 3.16 – Interopérabilité : la route sensible à la source contourne le routeur classique.

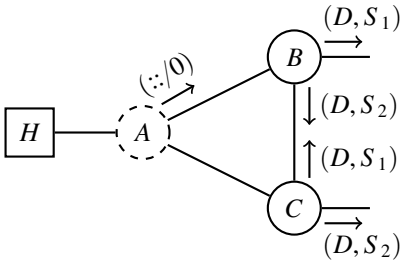
Cas particulier des réseaux multihomés Dans le cas du *multihoming* avec plusieurs adresses, chaque FAI fournit un préfixe d'adresses et une route par défaut spécifique à ces adresses. Il n'y a donc que des routes spécifiques pour sortir du réseau : un routeur *next-hop* ne peut pas installer de route par défaut (figure 3.17).

Pour pallier ce problème, il suffit (figure 3.17), d'une part, que les routeurs de frontière soient dans une composante connexe de routeurs sensibles à la source, formant une épine dorsale (*backbone* — figure 3.18) et, d'autre part, qu'au moins un routeur sensible à la source annonce une route par défaut (non spécifique). Les paquets concernés seront transférés par les routeurs classiques vers l'un des routeurs annonçant la route par défaut puis seront acheminés vers le bon fournisseur d'accès dès qu'ils atteindront l'épine dorsale — dont chaque routeur connaît toutes les routes par défaut spécifiques à un préfixe source. Si un paquet a une adresse source qui ne correspond à aucune des routes spécifiques à la source, il sera acheminé jusqu'à un des routeurs annonçant une route par défaut puis sera détruit.

Cette technique limite légèrement un des apports du routage sensible à la source. En routage *next-hop*, les paquets à destination de l'Internet ayant une adresse source différente de celles fournie par le FAI sont

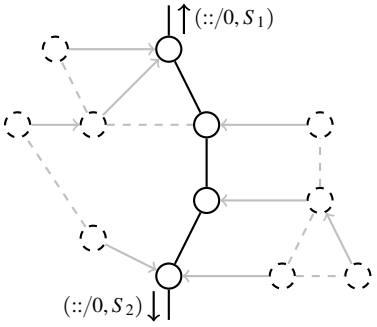


(a) B et C n'annoncent que les routes spécifiques : A n'a pas de route pour l'Internet.



(b) B et C annoncent une route par défaut : A passe par l'un des deux pour tous les paquets.

FIGURE 3.17 – Interopérabilité dans un réseau *multihomé*.
A est un routeur classique, B et C sont des routeurs de frontière sensibles à la source.



Exemple de réseau avec une épine dorsale de routeurs sensibles à la source (en trait plein). Les autres routeurs, en pointillés, sont des routeurs classiques. Une route par défaut achemine les paquets des routeurs classiques vers les routeurs sensibles à la source où ils sont alors acheminés au bon fournisseur d'accès.

FIGURE 3.18 – Épine dorsale de routeurs sensibles à la source.

envoyés au FAI par qui ils sont filtrés (voir figure 2.17*a*). En routage sensible à la source, aucune route n'existe pour ces paquets : ils sont détruits au plus tard par le premier routeur (ils peuvent ne même pas être émis par l'hôte). En annonçant une route par défaut pour les routeurs *next-hop*, ces paquets sont routés jusqu'à l'émetteur de cette route où ils sont détruits. Ces paquets sont donc bien détruits par le réseau, et non par le FAI, mais plus tard que dans le cas d'un routage sensible à la source entièrement déployé.

Chapitre 4

Implémentation du routage sensible à la source à vecteur de distances

Au chapitre 3, nous avons défini un algorithme de levée d'ambiguïté adapté aux protocoles de routage, nous avons étendu l'algorithme à vecteur de distances *next-hop* au routage sensible à la source et nous avons montré comment étendre de manière compatible un protocole de routage à vecteur de distances au routage sensible à la source.

Afin de valider notre approche et de montrer qu'il est effectivement possible d'utiliser ces algorithmes, nous avons réalisé une implémentation complète du routage sensible à la source. Il aurait été suffisant de réaliser cette implémentation pour un protocole jouet, mais il est plus satisfaisant de la faire avec un protocole de routage vraiment utilisé. Notre choix s'est porté sur le protocole Babel dont les particularités ne devraient pas être un obstacle à la généralisation de notre approche à l'ensemble des protocoles à vecteur de distances. Par ailleurs, ce protocole nous est familier et nous avons connaissance de déploiements pour lesquels il est important de préserver l'interopérabilité. Nous avons donc adapté nos idées aux spécificités de Babel, mais nous aurions pu les adapter pareillement à d'autres protocoles de routage à vecteur de distances comme RIPng ou EIGRP.

Babel [Chr11] est un protocole à vecteur de distances extensible et robuste avec de fortes propriétés d'absence de boucles. Les mécanismes utilisés autorisent Babel à utiliser une valeur arbitrairement grande pour sa métrique, sans contrepartie sur ses propriétés de convergence (il n'y a pas de comptage à l'infini). Le protocole de base évalue dynamiquement les coûts des liens en se basant sur la perte de paquets. La robustesse de Babel lui a permis d'être étendu, tout en restant compatible avec la version de base du protocole, et ainsi d'être adapté à de nombreux types de réseaux, stables ou instables. En particulier, des extensions existent pour évaluer dynamiquement d'autres paramètres de qualité de lien pour le calcul de la métrique, comme la diversité des canaux radio [Chr14] ou le RTT [JBC14, JC15].

Cette implémentation montre que nos techniques sont suffisantes pour étendre un protocole à vecteur de distances au routage sensible à la source. L'algorithme de levée d'ambiguïté de la section 3.2 s'intègre avec très peu de changements au code d'origine, nous laissant conclure qu'il est utilisable en pratique et adapté aux protocoles de routage. Enfin, nous avons pu assurer la compatibilité avec la version de base, et utiliser notre implémentation dans un réseau *multihomé* avec des routeurs spécifiques et non spécifiques.

Dans ce chapitre, nous commençons par présenter le protocole Babel. Nous présentons ensuite notre extension au routage sensible à la source, puis nous présentons quelques résultats expérimentaux.

4.1 Présentation du protocole Babel

Nous présentons dans cette section les spécificités du protocole Babel, son format de message et ses mécanismes d'extension.

4.1.1 Un protocole à vecteur de distances sans boucles

Babel [Chr11, CS17] est un protocole à vecteur de distances. Il utilise l'algorithme à vecteur de distances défini dans la section 2.1.3.3, avec quelques modifications que nous voyons dans cette section. Nous voyons notamment comment Babel prévient la formation des boucles.

Détection explicite de voisins Comme nous l'avons vu en section 2.1.2, la relation de voisinage entre deux routeurs dépend des interfaces qui les connectent. Deux routeurs peuvent avoir plusieurs relations de voisinages s'ils sont connectés de plusieurs manières différentes : on peut dire que deux routeurs sont voisins plusieurs fois. Ainsi, lorsqu'un routeur sélectionne un voisin comme *next-hop*, il choisit en fait une de ses propres interfaces et une adresse où transférer le paquet.

Un nœud a besoin de savoir quels voisins sont accessibles pour les sélectionner comme *next-hop*. L'algorithme à vecteur de distances décrit en section 2.1.3.3 considère que dès qu'une annonce de route envoyée par un voisin est reçue, ce voisin est accessible. Et inversement, lorsqu'il ne reçoit plus d'annonce d'un voisin, il considère, au bout d'un moment, que ce voisin ne l'est plus. Mais il existe des cas où l'accessibilité n'est pas bidirectionnelle : un nœud peut recevoir les messages d'un autre et pas l'inverse.

Babel utilise un mécanisme dédié pour détecter l'accessibilité de ses voisins, avec deux messages. Un nœud envoie des *Hello* pour annoncer son existence à ses voisins. En réponse à un *Hello*, un nœud répond par un *IHU (I Heard You)*. La réception d'un *IHU* confirme la que l'accessibilité est bidirectionnelle.

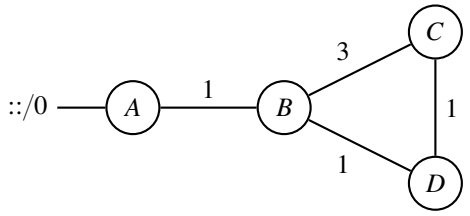
Un *Hello* est muni d'un intervalle de temps durant lequel l'émetteur du *Hello* s'engage à renvoyer un *Hello*. Si aucun *Hello* n'est reçu pendant l'intervalle, un nœud considère que le *Hello* a été perdu. Si trop de *Hello* sont perdus, un nœud considère que le voisin est en panne. Les *Hello* sont peuvent être utilisés pour mesurer le taux de perte d'un lien et ainsi estimer sa qualité.

Prévention de formation des boucles L'élimination des boucles dans un protocole à vecteur de distances est difficile, car il s'agit de déterminer des propriétés globales (une route n'a pas de boucle) à partir de propriétés locales (une entrée de la table de routage). Babel utilise une *condition de faisabilité*, qui détermine avec certitude si une route est sans boucle. Cependant, elle peut donner lieu à des faux positifs : une route sans boucle peut être déclarée infaisable.

Plusieurs conditions de faisabilité existent ; Babel utilise celle de DUAL [GLA93]. Elle repose sur la notion de *distance de faisabilité* $fd_r(A)$ d'une route r pour un routeur A, définie comme la plus petite métrique annoncée pour r à ses voisins. Une route r annoncée à A par un de ses voisins B avec une métrique $m_r(B)$ est faisable lorsque la métrique annoncée par B est strictement inférieure à la distance de faisabilité de r pour A, c'est-à-dire lorsque $m_r(B) < fd_r(A)$.

Intuitivement, la condition de faisabilité repose sur le fait que la métrique d'une route est croissante à chaque saut. Si un routeur reçoit l'annonce d'une route passant par lui-même, cette route a nécessairement une métrique supérieure à la métrique que le nœud a annoncé pour cette route. Pour rejeter cette route, il suffit donc de se souvenir de la valeur de la plus petite métrique annoncée pour cette route, et de n'accepter que les routes ayant une métrique plus petite (strictement) que cette valeur.

Le calcul des distances par Babel pour les figures 2.13 et 2.15 est illustré sur la figure 4.1. Dans la première partie (le lien entre A et B existe), la distance de faisabilité de C diminue toujours au cours du temps : à la troisième itération, D annonce une métrique de 2, inférieure à la distance de faisabilité $fd_{::/0}(C) = 4$



A	B	C	D
0, **FIB**	$\bot$	$\bot$	$\bot$
0, **FIB**	$1, B \to A$	$\bot$	$\bot$
0, **FIB**	$1, B \to A$	$4, C \to B$	$2, D \to B$
0, **FIB**	$1, B \to A$	$3, C \to D$	$2, D \to B$

(a) Avec l'algorithme à vecteur de distances classique.

A	B	C	D
$((m = 0, fd = 0)$, **FIB**$)$	$\bot$	$\bot$	$\bot$
$((0,0)$, **FIB**$)$	$((1,1), B \to A)$	$\bot$	$\bot$
$((0,0)$, **FIB**$)$	$((1,1), B \to A)$	$((4,4), C \to B)$	$((2,2), D \to B)$
$((0,0)$, **FIB**$)$	$((1,1), B \to A)$	$((3,3), C \to D)$	$((2,2), D \to B)$
$((0,0)$, **FIB**$)$	$((\bot,1), \bot)$	$((3,3), C \to D)$	$((2,2), D \to B)$
$((0,0)$, **FIB**$)$	$((\bot,1), \bot)$	$((3,3), C \to D)$	$((\bot,2), \bot)$
$((0,0)$, **FIB**$)$	$((\bot,1), \bot)$	$((\bot,3), \bot)$	$((\bot,2), \bot)$
...	...	...	...

(b) Avec le protocole Babel : pendant la converge, puis quand le lien entre A et B tombe en panne.

FIGURE *4.1 – Calcul des* next-hop *pour la route redistribuée par A.*

de C pour la même route, et C peut donc passer par D. Dans la deuxième partie, B, perdant sa route $(1, D)$, rejette les annonces de C et de D qui ont des métriques supérieures à 1 (respectivement 3 et 2). À leur tour, D puis C perdent leur route sans créer de boucles.

Famine Comme la distance de faisabilité d'une route est strictement décroissante en fonction du temps, il est possible, si les métriques du réseau augmentent (par exemple parce que les bons liens cassent), de se retrouver en situation de famine : aucune route existante n'est jugée faisable. Ce serait le cas dans notre exemple si nous réactivons le lien entre A et B avec un coût de 4. Le nœud B annoncerait alors une route avec une métrique de 4, ce qui est strictement supérieur aux distances de faisabilité de B et C (respectivement de 2 et 3).

Afin de sortir de la famine, Babel étiquette ses routes avec des *numéros de séquence*. La distance de faisabilité est alors une paire $(\text{numéro de séquence}, \text{métrique})$. Une route est faisable lorsque son numéro de séquence est plus récent que celui de la distance de faisabilité, ou que les deux numéros de séquence sont égaux mais que la métrique de la route est strictement inférieure à celle de la distance de faisabilité. Plus formellement, si B annonce une route r avec une métrique m_B et un numéro de séquence s_B et si $fd_r(A) = (m, s)$ alors la route annoncée par B est faisable pour A ssi $s_B > s$ ou $s_B = s$ et $m_B < m$.

En principe, les numéros de séquence sont incrémentés sur demande, par un mécanisme de requêtes. Lorsqu'un routeur est sur le point d'avoir une famine, il peut demander un nouveau numéro de séquence en envoyant une requête (*Seqno Request*) à l'émetteur de la route. En routage *next-hop*, les requêtes contiennent le préfixe destination associé à la route ainsi que l'identifiant de l'émetteur de la route.

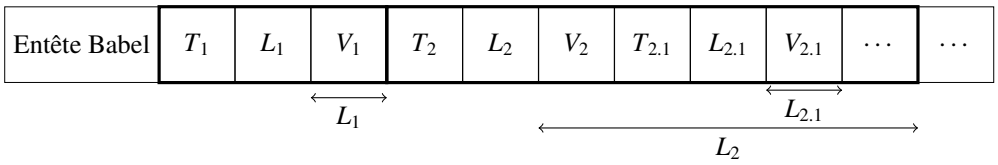
Expiration d'une route Afin d'empêcher la persistance de routes obsolètes dans le réseau, chaque route est munie d'une durée de vie. Un routeur ne recevant pas de mise à jour pour cette route doit la retirer, et en informer ses voisins en leur envoyant une rétraction de route.

Pour prévenir la rétraction de routes encore valides, un routeur avec une route sur le point d'expirer demande au voisin, de qui il tient cette route, de renvoyer immédiatement une annonce pour cette route. Le message utilisé pour demander une nouvelle annonce est appelé *requête de route (Route Request)*. Cette requête est destinée à un voisin, et n'est pas à confondre avec la requête de numéro de séquence qui est

destinée à l'émetteur de la route. Contrairement à cette dernière, la requête de route n'est pas relayée.

4.1.2 Structure du format de message de Babel

Un paquet Babel (figure 4.2) est formé d'une suite de TLV précédée d'un entête. Un *TLV (Type-Length-Value)* est un message constitué de trois champs : son type, sa taille et sa valeur. Le *type* d'un TLV détermine l'encodage de l'information contenue dans sa *valeur*. Par exemple, le TLV *Update* de la figue 4.3*a* a 8 pour type, *Length* pour taille, et l'ensemble des champs *AE*, *Flags*, *Plen*, *Omitted*, *Interval*, *Seqno*, *Metric* et *Prefix* pour valeur.



Le premier TLV (T_1, L_1, V_1) est sans sous-TLV : la taille de V_1 est exactement L_1. Le deuxième TLV (T_2, L_2, V_2) possède plusieurs sous-TLV ($T_{2.1}, L_{2.1}, V_{2.1}, \ldots$) : la taille L_2 tient compte de ceux-ci.

FIGURE 4.2 – Un paquet Babel : une suite de TLV qui peuvent contenir des sous-TLV.

Le protocole Babel autorise l'usage de sous-TLV (illustré sur la figure 4.2). La *taille* d'un TLV Babel n'est pas nécessaire pour analyser son contenu de base. Augmenter le champ *taille* d'un TLV libère un espace entre le contenu de base et la fin du TLV. Cet espace est rempli par une suite de TLV, appelés sous-TLV. Les champs de la valeur des sous-TLV s'ajoutent aux champs du TLV de base.

Une annonce de route Babel est composée de 1 à 3 TLV, mais, pour les besoins de cette discussion, nous pouvons considérer qu'une annonce coïncide avec un TLV *Update* (figure 4.3*a*). Ce TLV contient un préfixe destination défini par les champs *Plen* et *Prefix*. Le premier est la taille du préfixe et le second est le préfixe lui-même. Deux autres TLV contiennent un préfixe destination : le TLV *Seqno Request* (figure 4.3*c*), utilisé en cas de famine, et le TLV *Route Request* (figure 4.3*b*), pour prévenir l'expiration d'une route.

4.1.3 Mécanismes d'extension de Babel

Les TLV sont extensibles par nature : une implémentation ne connaissant pas le type d'un TLV peut passer au suivant grâce à sa taille en ignorant totalement sa valeur. De nouveaux TLV peuvent donc être définis et ajoutés au protocole sans casser les implémentations existantes [Chr15]. De la même manière, il est possible de définir de nouveaux sous-TLV : lorsqu'un TLV possède des sous-TLV de type inconnu, seuls ceux-ci sont ignorés.

La nouvelle version de Babel introduit un autre type de sous-TLV, motivé par notre travail ainsi que par le travail de Chouasne [CC17] : le sous-TLV obligatoire. Un sous-TLV est obligatoire si son bit de poids fort est 1. Lorsqu'un TLV contient un sous-TLV obligatoire inconnu, le TLV tout entier est ignoré. Cela nous permet de définir le préfixe source que nous voulons ajouter en tant que sous-TLV et d'éviter qu'une implémentation ne garde que le préfixe destination, ce que nous avons montré nécessaire à la section 3.3.1.

4.2 Extension sensible à la source de Babel

Nous décrivons dans cette section notre extension de Babel au routage sensible à la source.

4.2.1 Extension de Babel au routage sensible à la source

Parmi les TLV du protocole Babel d'origine, trois contiennent un préfixe destination. Ce sont les seuls TLV qui sont inadaptés au routage sensible à la source. Ces trois TLV, représentés en figure 4.3, permettent d'envoyer une annonce de route (*Update*), de demander à un voisin d'envoyer une annonce de route (*Route Request*) et de demander au nœud à l'origine d'une route d'incrémenter son numéro de séquence (*Seqno Request*).

L'extension que nous avons définie pour la nouvelle version de Babel [1], qui supporte les TLV obligatoires, comporte un seul sous-TLV obligatoire qui s'applique aux trois TLV de base. Ce sous-TLV contient seulement un préfixe source (figure 4.4). Ce TLV est utilisé pour les routes spécifiques à un préfixe source et omis pour les routes non spécifiques. Puisque le sous-TLV est obligatoire, une implémentation classique ignore les TLV contenant un préfixe source.

Cette extension permet l'interopérabilité avec le protocole de base. Celle-ci est obtenue en utilisant les TLV du protocole de base pour les routes non spécifiques à la source.

4.2.2 Levée d'ambiguïtés et manipulation de la FIB

L'implémentation de Babel sur laquelle nous travaillons permet de manipuler la FIB par un module séparé offrant les trois fonctions d'ajout, de modification et de suppression d'une entrée de la FIB. Nous avons implémenté notre algorithme de levée d'ambiguïté (défini en section 3.2) dans un module indépendant n'impactant pas le code existant. Il utilise les fonctions du module de communication avec le noyau et propose trois fonctions similaires en façade.

Pour intégrer l'algorithme de levée d'ambiguïté au reste du code, il a suffit de remplacer les appels aux fonctions du module de communication avec le noyau par des appels aux fonctions équivalentes proposées par notre algorithme. L'algorithme de levée d'ambiguïté se situe donc bien comme une couche intermédiaire entre la RIB et la FIB.

Bien sûr, nous avons dû aussi modifier le module de communication avec le noyau pour qu'il puisse manipuler des entrées avec un préfixe source. Nous avons réalisé cette implémentation pour le noyau Linux en utilisant les deux API que nous avons présentées en section 3.1.3.

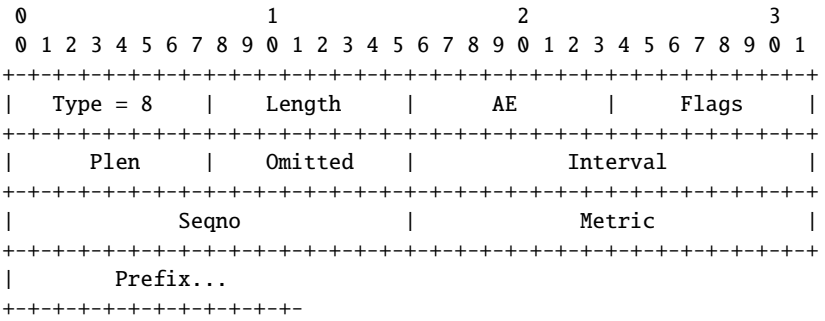
Les versions récentes du noyau Linux supportent nativement le routage sensible à la source IPv6. L'interface *Netlink* de communication avec le noyau utilise un format de message à base de TLV. Un de ces TLV correspond au préfixe source des entrées à manipuler. Il suffit de paramétrer ce TLV pour manipuler des entrées spécifiques à la source. Comme nous utilisions déjà *Netlink* pour communiquer avec le noyau, le support du routage sensible à la source avec l'interface native était peu coûteux.

Dans les autres cas, notamment en IPv4, il est nécessaire d'utiliser des tables multiples. Toute la gestion des tables doit être effectuée par notre implémentation : une règle de cette table est un triplet contenant la *priorité*, qui définit l'ordre d'évaluation des tables, le *préfixe source* qui définit l'ensemble des paquets auquel cette règle s'applique et la *table de transfert* à utiliser pour router ces paquets. Nous avons donc défini des fonctions pour ajouter des règles dont l'ordre des priorités préserve celui des préfixes sources : un préfixe source plus spécifique qu'un autre doit avoir un numéro de priorité plus petit pour être sélectionné en premier.

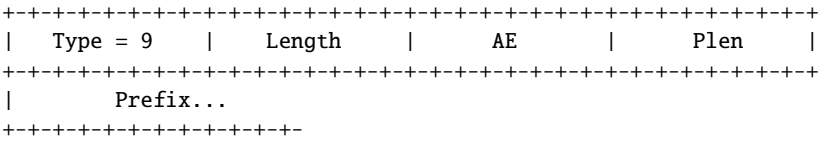
4.3 Résultats expérimentaux

Nous avons réalisé plusieurs expériences dans notre réseau expérimental qui nous ont permis de voir le routage sensible à la source en action au niveau du réseau et des couches supérieures. Dans la première

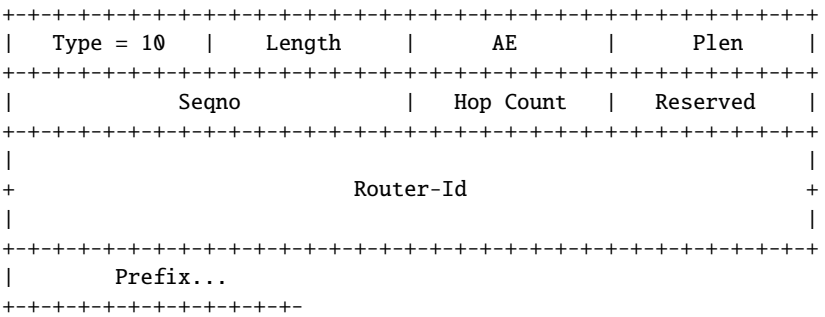
1. Nous avons aussi défini une extension pour l'ancienne version, qui comportait trois nouveaux TLV, copies des trois TLV *Update*, *Route Request* et *Seqno Request* augmentés d'un préfixe source.



(a) Update

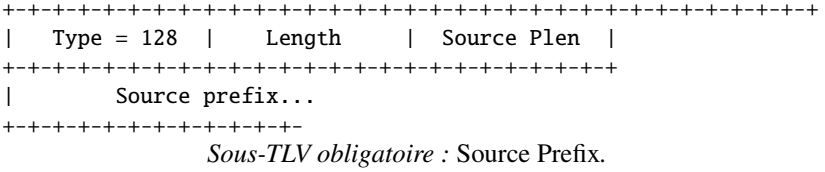


(b) Route Request



(c) Seqno Request

FIGURE 4.3 – TLV du protocole de base.



Sous-TLV obligatoire : Source Prefix.

FIGURE 4.4 – Extension sensible à la source pour la version standard de Babel.

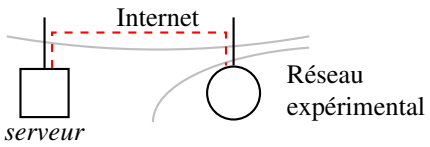


FIGURE 4.5 – Réseau connecté à un serveur par VPN.

partie de cette section, nous montrons diverses utilisations du routage sensible à la source que nous avons utilisées. Les principales conclusions que nous pouvons tirer est que le routage sensible à la source :

— s'adapte bien à plusieurs topologies ;
— répond réellement au besoin en routage de ces topologies sans problèmes particuliers à pallier avec d'autres techniques.

Dans la deuxième partie de cette section, nous testons divers applications classiques dans le cadre d'un réseau *multihomé* avec routage sensible à la source. Nos expériences montrent que le routage sensible à la source répond à nos attentes :

— des paquets avec des adresses source différentes mais la même adresse destination suivent des routes différentes ;
— ces routes peuvent être très différentes ;
— un hôte peut être connecté simultanément à tous les fournisseurs, et le routage lui offre différentes routes.

Ce troisième et dernier point est une transition pour le chapitre suivant où nous nous intéressons plus particulièrement aux possibilités offertes par le routage sensible à la source aux couches supérieures.

4.3.1 Configuration d'un réseau sensible à la source

Nous avons utilisé le routage sensible à la source pour recevoir la connectivité du réseau par un serveur distant relié au réseau par VPN, mais aussi pour expérimenter avec le *multihoming* (chapitre 5), ou encore pour expérimenter avec la suite de protocoles de *Homenet*. Notre réseau expérimental est un réseau maillé (*mesh*) constitué d'une dizaine de routeurs wifi répartis dans notre laboratoire. Il est connecté à l'Internet par deux routeurs de frontière. Ces deux routeurs sont directement connectés au réseau de l'université, distinct de notre réseau expérimental.

Notre réseau a été connecté de diverses manières à l'Internet : par l'utilisation de NAT aux routeurs de frontière, par VPN connectés à un serveur hors de l'université, par des tunnels IPv6, ou encore par des routes IPv6 annoncées par le réseau de l'université.

Trafic sortant par VPN Dans cette topologie, décrite en figure 4.5, le réseau expérimental reçoit sa connexion à l'Internet d'un serveur distant. Ce serveur est connecté au réseau par un tunnel : il fait partie intégrante du réseau, et utilise le protocole de routage du réseau (Babel) pour annoncer sa route par défaut.

Cette configuration est problématique : le routeur de frontière qui établit le tunnel avec le serveur possède deux routes par défaut ; celle annoncée par le serveur, qui passe dans le VPN (en pointillés), et celle, native et propre au routeur de frontière, qui sert entre autres à établir le VPN (en trait plein). Lorsque le protocole de routage installe la route du serveur dans la table de transfert du système, tous les paquets passent dans le VPN, y compris les paquets encapsulés. Ainsi, lorsque la route du VPN est installée, le tunnel entre dans lui-même : la connexion s'écroule. Babel détecte la rupture au bout de quelques secondes et retire la route du VPN, ce qui rétablit la connexion, donc l'installation de la route, donc l'effondrement du tunnel, et ainsi de suite.

```
1  # vtysh -c "configure terminal" \
2          -c "ipv6 route $prefix $router_univ" \
3          -c "table 5"
4  # ip -6 rule show
5  0:      from all lookup local
6  [[some rules for Babel source-sensitive routing]]
7  500:    from all to 2001:660:3301:9208::/64 lookup main
8  501:    from 2001:660:3301:9208::/64 lookup 5
9  32766:  from all lookup main
10 # cat /etc/babeld.conf
11 import-table 254
12 import-table 5
13 redistribute ip ::/0 eq 0 src-prefix 2001:660:3301:9208::/64
```

FIGURE 4.6 – Configuration du réseau *multihomé*.

Le routage sensible à la source apporte une solution élégante à ce problème en distinguant les deux routes. Au niveau du serveur, la route par défaut doit être redistribuée comme spécifique au préfixe du réseau expérimental et, au niveau du routeur de frontière, le tunnel doit être configuré pour utiliser l'adresse de l'interface directement connectée à l'Internet. Ainsi, les paquets du réseau expérimental sont bien encapsulés et les paquets encapsulés passent bien par la route native.

```
redistribute ip 0.0.0.0/0 eq 0 src-prefix 192.168.4.0/24
```

En fournissant directement une route spécifique à la source, aucune configuration n'est nécessaire sur les routeurs du réseau pour obtenir cette route spécifique.

Route apprise d'un protocole classique Notre université nous fournit quelques préfixes IPv6, et annonce une route par défaut non spécifique par RIPng. Un routeur de frontière apprend cette route et l'installe comme spécifique au préfixe source correspondant. Puis, il la redistribue dans Babel qui l'annonce au réseau expérimental. Le routeur de frontière doit de l'autre côté annoncer par RIPng le préfixe d'adresses dont il est responsable.

La configuration d'un tel routeur est illustrée en figure 4.6, où `prefix` désigne le préfixe attribué au réseau par ce routeur, et `router_univ` désigne l'adresse du routeur de frontière attribuée par l'université. Nous utilisons la version de RIPng de *quagga*, une suite pour le routage : les lignes de 1 à 3 servent à la configurer. La ligne 2 ajoute une route statique indiquant que notre routeur est responsable du routage de ce préfixe. Ainsi, le routeur de l'université fait suivre à notre routeur les paquets à destination du préfixe annoncé.

La route par défaut reçue par RIPng n'est pas naturellement spécifique à la source. Nous devons donc demander à *quagga* de l'installer dans une table particulière (ligne 3), et nous devons installer manuellement une règle associée à cette table pour que la route par défaut installée par RIPng soit spécifique au préfixe demandé (ligne 8). La règle que nous ajoutons a une priorité (501) inférieure à celle de la règle par défaut (32766) car nous voulons que la route par défaut spécifique à la source installée par RIPng soit considérée avant une route par défaut non spécifique. Cependant, ces règles étant installées manuellement, il convient d'installer des règles de levée d'ambiguïtés également manuellement. Ici, il suffit de dire que tous les paquets à destination de notre préfixe doivent être routés par la table par défaut (ligne 7).

Enfin, comme nous le voyons, la configuration de Babel est relativement simple : il faut lui dire d'importer à la fois les routes de la table par défaut (ligne 11) et de la table 5 (ligne 12), et d'ajouter un préfixe source à la route par défaut redistribuée (ligne 13).

Notons tout de même que cette configuration peut être incomplète : la levée d'ambiguïté étant partiellement faite manuellement, des erreurs peuvent se produire. En particulier, si le routeur a des tables de

transfert sensibles à la source natives, et que le réseau a plusieurs préfixes source, il faut ajouter davantage de règles semblables à celle de la ligne 7 : une par préfixe source. Considérons par exemple un paquet destiné au réseau et ayant une adresse source dans 2001:660:3301:9208::/64. S'il n'y a pas de table native, une entrée a été ajoutée dans la table des règles pour ce préfixe source et le paquet est routé par la table de transfert correspondante (ligne 6) ; celle-ci est complète : le paquet est correctement routé. Mais si les tables de transfert sont nativement sensibles à la source, la première règle à laquelle répond le paquet est celle de la ligne 8 : la table de transfert correspondante contient une route par défaut et le paquet est routé en dehors du réseau.

Homenet Les problèmes de configuration semi-automatique du paragraphe précédent peuvent être résolus en utilisant un protocole de configuration adapté. Le groupe de travail *Homenet* de l'IETF a en particulier mis au point un protocole d'attribution de préfixes pour les réseaux domestiques, HNCP [SBP16]. Les implémentations de ce protocole installent dans le noyau des routes sensibles à la source natives (un support pour ces routes est alors nécessaire), et configurent le protocole de routage pour qu'il annonce ces routes. Il n'y a alors aucune configuration à faire.

HNCP a été déployé dans notre réseau par Dorine Chagnon [Chr16] au cours d'un stage hors cursus. Le déploiement d'HNCP est une réussite : il configure les adresses des hôtes du réseau et fonctionne parfaitement avec notre implémentation sensible à la source de Babel (elle aussi faisant partie de la suite *Homenet*).

4.3.2 Routage sensible à la source et accessibilité des FAI

La plupart des applications classiques (*ping*, *traceroute*...) offrent déjà la possibilité de choisir l'adresse qu'elles utilisent comme source pour les paquets. Nous utilisons ces applications pour vérifier que le routage sensible à la source répond bien à nos attentes.

Tests avec mytraceroute L'outil *traceroute* liste l'ensemble des nœuds entre l'hôte où il est exécuté jusqu'à la destination spécifiée. L'adresse source utilisée peut aussi être spécifiée. Nous utilisons cet outil (en fait, une variante de cet outil — *mtr*) en faisant varier l'adresse source pour deux raisons :

— vérifier que le routage sensible à la source route bien les paquets différemment en fonction de la source ;
— comparer à quel point les différents chemins sont différents.

Le résultat de l'expérience est représenté en figure 4.7. Nous voyons que l'utilisation d'une adresse source différente donne bien des chemins très différents : seuls les trois derniers routeurs des deux chemins semblent être dans le même site.

OpenVPN Cette expérience est une mise en œuvre de la configuration du VPN décrite en section 4.3.1 et correspond à l'application aux réseaux superposés que nous avons décrite au début de ce manuscrit, en section 2.2.1.2. Cette expérience nous permet d'affirmer que le routage sensible à la source répond bien à ce cas d'usage.

Pour que le VPN ne rentre pas dans lui-même, nous configurons l'application pour qu'elle utilise l'adresse source fournie par le réseau de l'université (qui nous permet de joindre le serveur) — et non une adresse du réseau superposé. Comme pour *traceroute*, l'application *OpenVPN* a une option de configuration pour être liée à une adresse spécifiée. Celle-ci nous permet de choisir le chemin emprunté par les paquets encapsulés. Il nous a suffit de rajouter au fichier de configuration une ligne similaire à celle-ci :

```
local <adresse>
```

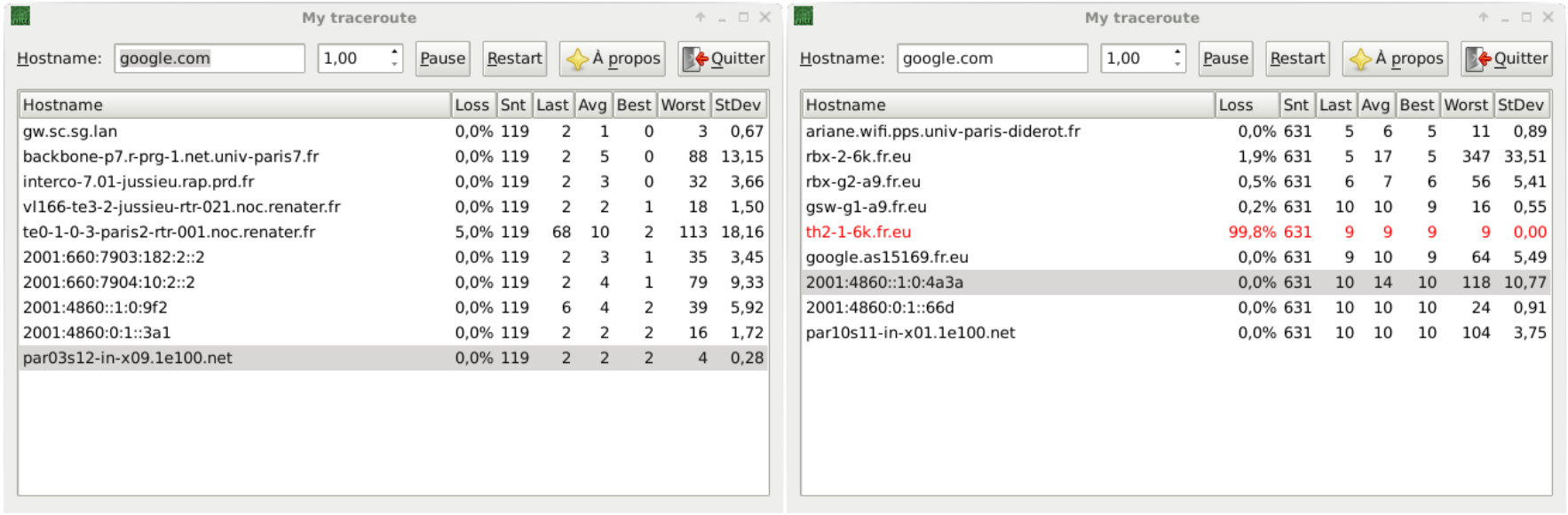


FIGURE 4.7 – Tests avec *MyTraceroute* et deux adresses IP source différentes.

Nous constatons que le routage sensible à la source transfère bien systématiquement les paquets du réseau superposé à travers le VPN, et seulement ceux-ci. La sélection de l'adresse source des paquets a suffit à configurer le VPN, sans avoir besoin de manipuler le routage du réseau.

Couches supérieures Le routage sensible à la source est une extension qui a des conséquences importantes pour les couches supérieures : le choix de l'adresse source des paquets correspond au choix d'un chemin dans le réseau. Les tests de cette section avec *mytraceroute* le montrent. Or, il existe déjà des couches supérieures capables de tirer profit de ce choix. Nous consacrons le chapitre 5 au lien entre les couches supérieures et le routage sensible à la source. En particulier, la section 5.2.1 contient des résultats empiriques qui montrent que le routage sensible à la source est une brique manquante dans une architecture plus vaste.

4.4 Ouverture

Nous pensions que le routage sensible à la source pouvait se généraliser à d'autres protocoles, ce qui a été notamment confirmé par le travail de Lamparter dans le protocole IS-IS [BL18], travail d'autant plus intéressant qu'il s'agit d'un protocole à état de lien, et donc très différent de Babel. Nous sommes convaincus que le travail que nous avons fait peut être réutilisé pour étendre d'autres protocoles à vecteur de distances mais aussi, pour ce qui est de la levée d'ambiguïté, à tout autre protocole de routage.

Plus encore, en ce qui concerne les protocoles de routage, nous avons montré que router les paquets en fonction de leur préfixe source ne modifiait pas les algorithmes utilisés, mais seulement l'indexation des structures de données et le format de messages. Cela est dû au fait que toute la complexité est gérée par le transfert — et par la levée d'ambiguïté. Deux propriétés ont aussi fait que le routage sensible à la source peut être déployé aisément dans un réseau :

— l'entête IP n'est pas modifié — aucun surcoût n'est nécessaire,
— la propriété fondamentale du routage *next-hop* est inchangée : un nœud du réseau ne décide toujours que du *next-hop* du paquet.

Or, un paquet IP a encore d'autres champs qui pourraient être utilisés pour le routage des paquets. Nous pensons naturellement au ToS [CC17].

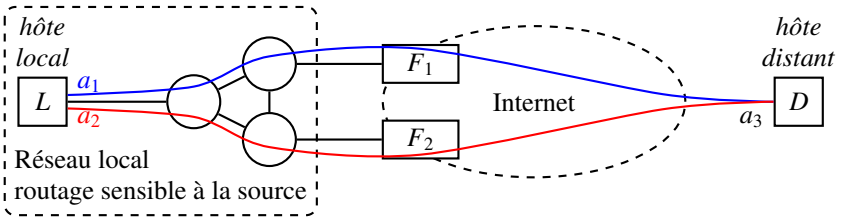
Chapitre 5

Couches supérieures

Dans les chapitres précédents, nous nous sommes intéressés au routage sensible à la source et nous avons défini tous les éléments nécessaires à la réalisation et à l'extension d'un protocole de routage sensible à la source à vecteur de distances. Nous avons réalisé une telle extension pour le protocole Babel.

Le choix de l'adresse source des paquets devient, avec le routage sensible à la source, un choix primordial. En effet, nous avons vu au chapitre 2 que le routage sensible à la source est une solution au routage des paquets dans les réseaux *multihomés* avec plusieurs adresses (section 2.2.1.1), mais nous avons aussi vu qu'il ne suffisait pas à assurer la fiabilité des connexions des hôtes (section 2.3). Le routage sensible à la source fournit à chaque hôte une route pour chaque fournisseur d'accès, si bien que si une panne intervient au niveau d'un des fournisseurs d'accès, l'hôte a toujours une route pour l'Internet par un autre fournisseur d'accès. Le choix du FAI par l'hôte coïncide avec le choix de l'adresse source des paquets qu'il envoie. Un hôte peut donc assurer la fiabilité de ses connexions en choisissant pour ses paquets une adresse source correspondant à un fournisseur opérationnel.

En réalité, en même temps que l'adresse source devient un choix primordial, celui de l'adresse destination l'est également. De même qu'un hôte *multihomé* choisit le FAI de ses paquets sortant par le choix de l'adresse source de ses paquets, un hôte distant choisit le chemin des paquets entrant dans le réseau *multihomé* par le choix de l'adresse destination de ses paquets (figure 5.1). Le choix du chemin d'un paquet dans le réseau coïncide donc avec le choix de la paire d'adresses source et destination du paquet.



L'hôte L a deux chemins distincts pour D, choisis par l'adresse source : les paquets avec a_1 et a_2 pour adresse source passent respectivement par F_1 et F_2. De même, D a deux chemins pour L, choisis par l'adresse destination.

FIGURE 5.1 – Plusieurs chemins : choix de l'hôte local et distant.

Dans ce chapitre, nous nous intéressons aux possibilités offertes par le choix des adresses source et destination d'un paquet. Une première contribution de ce chapitre est de montrer que le routage sensible à la source est une brique manquante à une construction bien plus vaste. En effet, plusieurs protocoles multichemin existent, basés sur la notion qu'un chemin correspond à une paire d'adresses. Pour notre étude, nous nous sommes particulièrement intéressés à MPTCP (section 5.2.1).

MPTCP est une extension compatible de TCP au *multihoming* : toute application utilisant TCP de-

vient multichemin dès lors que les noyaux (des deux pairs en communication) sur lesquels elle s'exécute supportent MPTCP. Or, plusieurs applications évitent d'utiliser TCP pour implémenter leurs propres mécanismes de transport au-dessus d'UDP. Comme autres contributions du chapitre, nous cherchons à voir dans quelle mesure ces applications gagnent à utiliser leurs propres mécanismes multichemin dans le cadre du *multihoming* avec routage sensible à la source.

5.1 Sélection d'adresses et multichemin

La plupart des applications choisissent une adresse destination parmi une liste d'adresses possibles (retournée par une API comme `getaddrinfo()`) et laissent la couche réseau choisir l'adresse source. La RFC 6724 [TDMC12] définit deux algorithmes standard pour le choix d'une paire d'adresses source et destination. Le premier algorithme cherche la meilleure adresse source possible pour une adresse destination donnée, et le deuxième algorithme trie un ensemble d'adresses destination en fonction des adresses source associées à ces préfixes par le premier algorithme. Le second algorithme est censé être utilisé par les API qui renvoient la liste des adresses d'un hôte distant (telles que `getaddrinfo()`), de sorte qu'une application n'a qu'à prendre la première adresse destination retournée — ou en cas d'échec, les essayer dans l'ordre tour à tour.

Pour choisir une paire d'adresses, la RFC 6724 utilise un certain nombre de paramètres locaux à l'hôte. Un de ces paramètres concerne la taille du préfixe commun aux deux adresses : plus le préfixe commun est grand, et plus il y a de chances que les deux hôtes appartiennent au même sous réseau. Cependant, aucune mesure effective de la qualité du chemin induit par la sélection de la paire d'adresses n'est réalisée.

Or, il arrive que la première paire d'adresses retournée par les algorithmes de la RFC 6724 donne lieu à un chemin infaisable ou simplement moins performant qu'un autre chemin. Cette situation arrive en particulier dans les déploiements où les hôtes ont à la fois une adresse IPv6 et une adresse IPv4. L'hôte a alors au moins deux chemins pour contacter un serveur [1], un par pile IP, qui peuvent être très différents.

Happy Eyeballs [WY12] est une technique proposée par Wing qui ne remplace pas la RFC 6724 mais qui tente d'établir simultanément la connexion avec la première paire d'adresses de chaque pile (IPv4 et IPv6) retournée par les algorithmes de la RFC 6724. La connexion la plus rapide à s'établir est gardée et définit l'adresse à utiliser pour le reste de la connexion. La norme de Happy Eyeballs a été actualisée par Schinazi [SP17] pour utiliser non pas une adresse de chaque pile mais toutes les adresses destination possibles. Happy Eyeballs constitue donc une technique multichemin où les différents chemins sont sondés simultanément avant d'en choisir un pour toute la connexion.

La principale limitation de Happy Eyeballs est de ne concerner que l'initialisation de la communication. Le meilleur chemin est utilisé au début de la connexion, mais si les performances du chemin induit par les adresses choisies se dégrade, ou même s'il vient à tomber en panne, le même chemin continue d'être utilisé. La connexion peut donc se retrouver coupée alors qu'un autre chemin est disponible. Par ailleurs, tous les chemins possibles ne sont pas exploités : les algorithmes de la RFC 6724 font correspondre exactement une adresse source par adresse destination. Un hôte avec deux adresses IPv6 voulant contacter un pair ayant une adresse IPv6 n'aura qu'un seul chemin possible au lieu de deux.

5.2 Techniques multichemin de couche transport

Un des buts du *multihoming* est d'obtenir un réseau fiable. La perte d'un FAI devrait être tolérable et ne devrait pas mener à la perte des connexions. Happy Eyeballs ne concerne que l'initialisation de la communication. Si une panne se produit sur le chemin initialement choisi par Happy Eyeballs, la connexion

1. Sous réserve que le serveur aussi ait des adresses IPv4 et IPv6.

est coupée. Happy Eyeballs peut alors être réutilisé pour recommencer une nouvelle connexion, mais il ne permet pas la fiabilité de la même connexion. Plusieurs protocoles multichemin existent qui garantissent la fiabilité d'une connexion, voire l'augmentation de performances, notamment Shim6, SCTP et MPTCP. Nous avons spécialisé notre étude à MPTCP qui, par certains aspects, est plus raffiné que les autres protocoles.

5.2.1 MPTCP

MPTCP [RPB+12] est une extension compatible de TCP pour le multichemin qui optimise le débit en répartissant la charge sur tous les chemins disponibles en fonction de leurs capacités. Les chemins sont continuellement évalués tout au long de la connexion et le protocole adapte sa stratégie à mesure que les performances des chemins changent. La panne d'un fournisseur d'accès est détectée presque instantanément et la charge est alors répartie sur le ou les chemins restants.

Quelques détails sur MPTCP MPTCP est un protocole multichemin de couche transport qui a plusieurs avantages : compatibilité avec TCP, fiabilité des connexions et répartition de charge.

TCP est le protocole de couche transport le plus utilisé. Il simule un flot de données entre deux hôtes ; la connexion TCP est identifiée par la paire d'adresses IP et par la paire de numéros de port utilisées. Le changement de l'adresse IP utilisée par une connexion TCP conduit donc à la perte de la connexion.

TCP cherche à maximiser le débit tout en évitant la congestion dans le réseau. Le débit que s'autorise TCP dépend en partie de la taille de sa fenêtre de congestion. Celle-ci contient les données envoyées non acquittées : plus elle est grande, et plus de données peuvent être envoyées simultanément.

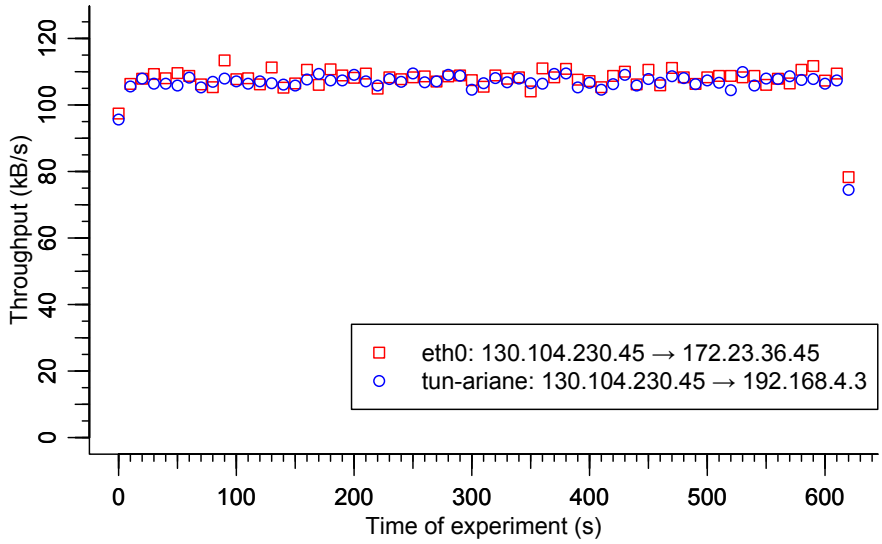
MPTCP est une extension compatible de TCP. Implémenté directement dans le noyau, toute les applications utilisant TCP sont automatiquement portées au multichemin, sans aucun besoin de modifier l'application. Cette dernière bénéficie de la fiabilité de ses connexions dans un réseau *multihomé* mais aussi de l'augmentation de son débit grâce aux capacités de répartition de charge de MPTCP.

MPTCP considère que chaque paire d'adresses induit un chemin différent. Le protocole inclut un mécanisme d'échange d'adresses pour que les deux hôtes puissent découvrir autant de chemins que possible entre eux. MPTCP tente d'établir un sous-flot pour chaque chemin possible ; une fois établis, MPTCP peut faire passer le trafic par l'un ou l'autre de ses sous-flots : si l'un d'eux tombe en panne, l'autre est toujours utilisable.

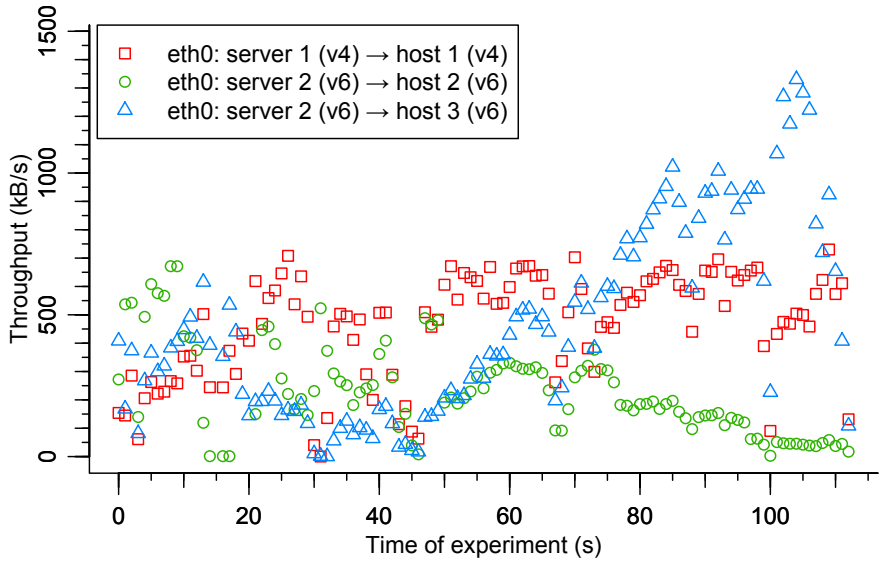
Enfin, MPTCP est capable d'optimiser le débit en répartissant la charge sur l'ensemble de ses sous-flots. Pour cela, il attribue à chaque sous-flot une fenêtre de congestion qui lui est propre et dont il ajuste dynamiquement la taille : les flots rapides reçoivent plus de données que les flots lents.

MPTCP et routage sensible à la source La notion de chemin dans MPTCP correspond exactement à celle que fournit le routage sensible à la source : MPTCP est capable d'établir des sous-flots basés sur les paires d'adresses de l'hôte local et de l'hôte distant. Nous avons réalisé plusieurs expériences mettant en évidence le bon fonctionnement de MPTCP avec le routage sensible à la source, et en présentons deux. Il s'agit dans les deux cas d'un téléchargement d'un fichier de 110 MB depuis un site Web qui a deux adresses : une IPv4 et une IPv6. Notre hôte local se trouve dans un réseau *multihomé* avec deux routeurs de frontière.

Dans le premier test, nous disposons de deux adresses *IPv4*, et limitons le téléchargement des deux routeurs de frontière à 100 kB/s. La figure 5.2*a* représente le résultat de l'expérience : chacun des deux chemins est utilisé au mieux. Il n'y a besoin d'aucune configuration supplémentaire des applications : le routage sensible à la source donne à MPTCP tout ce dont il a besoin pour exploiter les différents chemins.



(a) avec limitation de débit ($100\,\mathrm{MB}/s$*)*



(b) sans limitations

FIGURE 5.2 – Téléchargements avec MPTCP.

Dans le second test, un ordinateur qui a une interface wifi et trois adresses (une v4, et deux v6) est placé au milieu du réseau pour réaliser le téléchargement, sans limite de débit. La figure 5.2*b* représente le résultat de l'expérience : MPTCP répartit le trafic sur les trois chemins et adapte dynamiquement leur débit.

Ces expériences sont particulièrement intéressantes car elles montrent que le routage sensible à la source n'est pas une technologie isolée mais une brique qui manquait dans une architecture plus vaste. Le routage sensible à la source est l'élément manquant de la couche réseau qui permet le multichemin et fiabilité les réseaux *multihomés*.

5.2.2 Comparaison de MPTCP avec d'autres protocoles

Plusieurs solutions de couche réseau sont apparues [NC11], visant à apporter une solution générique pour fiabiliser toutes les couches supérieures. L'une de ces solutions est Shim6 [GMBVB10, NB09]. Shim6 est un protocole qui s'intercale entre la couche réseau et la couche transport. Il réécrit les adresses des paquets entrant et sortant de sorte que les paquets utilisent des adresses correspondant à un chemin faisable sans que les couches supérieures ne remarquent les changements d'adresses. Ainsi, les connexions sont maintenues — le protocole TCP, notamment, peut continuer d'utiliser les mêmes adresses.

Pour cela, quand une connexion à un hôte distant est établie par des couches supérieures, Shim6 collecte les adresses des deux hôtes de part et d'autre de la connexion. Quand la connexion tombe en panne (Shim6 détecte une interruption de traffic), Shim6 trouve un chemin alternatif fonctionnel en sondant toutes les paires d'adresses source et destination. La meilleure paire d'adresses est gardée et les paquets sont renumérotés.

Le principal intérêt de Shim6 par rapport à MPTCP est de pouvoir assurer la fiabilité de n'importe quel protocole de couche transport. Mais contrairement à MPTCP, Shim6 ne permet pas l'usage de plusieurs chemins simultanément et ne réagit pas s'il n'y a pas de panne : le chemin est gardé si ses performances se détériorent.

SCTP [Ste07] est un protocole de couche transport fiable, ordonné ou non, orienté message et orienté connexion qui, outre de nombreuses fonctionnalités, a des capacités multichemin pour les hôtes *multihomés*. Lorsqu'une connexion SCTP (*association*) est établie, un chemin primaire (*primary path*) est sélectionné par chaque extrémité et utilisé pour envoyer des paquets. Un certain nombre de mécanismes autorisent les extrémités à échanger leurs adresses et à construire des chemins alternatifs. Ces chemins sont régulièrement sondés à intervalles fixes *(heartbeats)* pour vérifier leur viabilité. Lorsqu'une panne est détectée sur le chemin primaire, l'un des chemins alternatifs fonctionnels est sélectionné pour être le nouveau chemin primaire.

La description du protocole ne donne aucune indication sur la stratégie à adopter lorsque le chemin primaire est actif mais moins efficace qu'un autre chemin. L'implémentation Linux ne compare les chemins qu'en fonction de leur état suivant l'ordre de préférence : actif, inconnu, partiellement perdu, puis inactif.

Le principal intérêt de MPTCP par rapport à SCTP est d'être une extension compatible à TCP, le protocole de transport le plus utilisé : n'importe quelle application déjà existante utilisant TCP devient multichemin. De plus, la répartition de charge est partie intégrante de MPTCP, tandis que seule la fiabilité est directement prévue par SCTP, comme le reflète l'implémentation du noyau Linux.

5.3 Techniques multichemin de couche application

TCP est un protocole qui fournit beaucoup de garanties : arrivée de toutes les données envoyées, dans l'ordre, et avec une gestion automatique de la perte d'une connexion. UDP au contraire ne donne aucune garantie sur le paquet envoyé. Bien que TCP soit désirable dans beaucoup de cas, certaines applications arrivent à de meilleures performances en définissant leurs propres mécanismes de transport au-dessus d'UDP.

Nous voyons à travers ces applications que la connaissance des données transportées joue un rôle dans la conception du transport et qu'elle permet de l'optimiser. Par exemple, TCP est adapté au téléchargement, où un paquet perdu doit être réémis puisqu'il faut l'intégralité du fichier ; au contraire, TCP est moins adapté à la voix sur IP, où attendre la réémission d'un fragment de voix perdu induirait un délai qui dégrade plus l'expérience utilisateur que s'il est simplement ignoré.

Le routage sensible à la source et le choix des chemins par les hôtes permet l'utilisation de transports multichemin comme MPTCP. Dès lors que le noyau supporte MPTCP, toutes les applications utilisant TCP bénéficient des avantages de MPTCP et sont donc adaptées au multichemin. Au contraire, les applications qui définissent leur propre transport (avec UDP) doivent être adaptées séparément pour supporter le multichemin. Nous nous demandons dans quelle mesure ces applications sont adaptables au multichemin, si la connaissance de l'application permet encore d'optimiser le transport et s'il y a un intérêt de faire du multichemin à la couche application. Nous nous demandons aussi quels mécanismes sont nécessaires ou profitables.

MPTCP est inadapté aux applications qui n'utilisent pas TCP. En effet, il arrive que MPTCP n'utilise qu'un seul chemin à la fois et se comporte alors comme TCP. C'est le cas lorsqu'il ne reste plus qu'un seul FAI de disponible, ou encore lorsque l'application génère suffisamment peu de trafic pour que MPTCP puisse tout faire passer sur un seul chemin. Dans ce deuxième cas, MPTCP apporte tout de même la fiabilité, au détriment des mécanismes d'optimisation propres à l'application.

Pour la suite de notre étude, nous avons pris le contre-pied de MPTCP en nous intéressant aux applications légères (qui génèrent un trafic modéré) et interactives (qui cherchent à minimiser la latence). Pour que nos recherches reposent sur un cas concret, nous avons choisi *mosh* [WB12], une application qui répond à ces critères, qui est sous une licence libre, et dont le code est bien compartimenté, ce qui nous aide à ne toucher à rien d'autre qu'aux mécanismes de transport et à ainsi pouvoir rester général bien qu'en modifiant une application particulière. Ce point important est d'autant plus vrai que le transport de mosh est compilé dans une bibliothèque séparée. Par ailleurs, nous avons eu le plaisir de voir que le code était bien écrit et agréable à modifier.

La première contribution que nous en retirons est de montrer qu'il est possible d'étendre une application basée sur UDP au multichemin en utilisant les API standard tout en restant à la couche application. Nous montrons qu'il est possible d'intégrer des mécanismes multichemin de manière asynchrone à une boucle à événement complexe, avec peu ou pas de modification à cette boucle à événement. Le comportement initial de l'application peut rester inchangé : en particulier, nous n'avons pas besoin d'interrompre la boucle à événement pour envoyer des paquets supplémentaires (sondes). Pourtant, nous sommes capables d'évaluer les performances des différents chemins tout au long de l'exécution de l'application, de détecter les pannes et de sélectionner le meilleur chemin pour chaque paquet.

La deuxième contribution est de rechercher des techniques d'optimisation d'une application légère et interactive en milieu multichemin. Nous montrons que la connaissance des besoins de l'application nous permet de tirer profit d'une manière particulière des différents chemins pour, au final, augmenter les performances de l'application. Dans notre cas (application légère et interactive), il est possible de dupliquer les paquets sans saturer les chemins. Nous nous servons de la duplication pour deux mécanismes d'optimisation : le premier accélère la convergence lors d'une panne sur un chemin, et le second limite efficacement les pertes de paquets.

5.4 Multichemin pour une application légère et interactive

Dans cette section, nous établissons des principes multichemin que nous pensons adaptés à une application légère et interactive. Nous les adapterons par la suite à mosh (section 5.5).

Intérêt de la couche application Nous avons choisi de nous intéresser à la couche application pour plusieurs raisons :

— la couche application est plus flexible au sens où il est possible d'adapter un code d'une application à une autre application en y apportant des changements spécifiques à l'application ;
— le déploiement d'une application est plus rapide que le déploiement d'une nouvelle fonctionnalité dans tous les noyaux où l'application doit s'exécuter ;
— la réutilisabilité du code est toujours possible par l'intermédiaire d'une bibliothèque ;
— il est possible de faire du multichemin à la couche application avec les API standard.

Limitations des applications à un seul chemin Il arrive que les applications basées sur UDP résistent naturellement au changement d'adresses locales. Il suffit pour cela que le protocole n'identifie pas les connexions par leurs adresses IP, que l'application laisse au noyau le choix de l'adresse source et que le pair distant réponde à la dernière adresse utilisée. Lorsque l'adresse utilisée pour la connexion est perdue, le noyau choisit automatiquement une nouvelle adresse, que le pair distant utilise en réponse : la connexion est maintenue.

Dans le cas du *multihoming*, il n'y a pas de raison que les adresses soient retirées des hôtes en cas de panne. En effet, l'hôte est toujours connecté aux routeurs du réseau — qui distribuent les adresses. Le noyau de l'hôte n'a donc en principe aucune information sur la panne et devrait continuer d'utiliser une adresse associée à un chemin en panne. Une application basée sur UDP peut donc résister aux pannes dans des cas particuliers, mais pas dans le cas général.

Par ailleurs, dans le cas où plusieurs chemins sont possibles, il est possible qu'un chemin de mauvaise qualité soit choisi. Il n'est pas non plus possible de tirer profit de l'utilisation simultanée de plusieurs chemins.

Sélection du meilleur chemin Pour sélectionner le meilleur chemin, il faut découvrir tous les chemins potentiels et les évaluer. Puisqu'un chemin coïncide avec une paire d'adresses, découvrir les chemins revient à trouver toutes les paires d'adresses locales et distantes. Les adresses locales peuvent s'obtenir par les API standard du système. Pour obtenir les adresses du pair distant, il est possible d'avoir déjà un premier ensemble d'adresses (par exemple en utilisant le DNS). Il est aussi possible d'ajouter un mécanisme d'échange des adresses dans le protocole de l'application. Dans ce cas, il faut aussi veiller à ce que les changements d'adresses soient transmis au pair distant.

Il y a plusieurs façons d'évaluer la qualité des chemins. Tout d'abord, les paquets contenant les données de l'application peuvent inclure les informations nécessaires à la mesure de la qualité des chemins, par exemple des horodatages (*timestamp*) pour évaluer le RTT ou un numéro de séquence pour détecter les pertes de paquets. Mais tous les chemins ne sont pas nécessairement utilisés pour transporter des données. Pour mesurer en continu la qualité de ces autres chemins sans perturber l'envoi des données, il est possible de définir dans le protocole de l'application un type de message particulier : les sondes.

Plusieurs protocoles utilisent des sondes pour mesurer la qualité d'un chemin, ou simplement vérifier s'il est fonctionnel. Par exemple, Shim6 sonde tous les chemins lorsqu'il détecte une panne, SCTP envoie des sondes à intervalles réguliers, et MPTCP aussi, d'une certaine manière, envoie des sondes avec les *keepalive*. Pour MPTCP, la qualité des différents sous-flots dépend plus généralement de chaque paquet et de la taille des fenêtres de congestion.

Notre proposition est une légère variante de ce qui existe. Chaque paquet contribue à l'estimation d'un lien : les paquets contenant des données évaluent la qualité du meilleur chemin, et des sondes sont envoyées sur les autres chemins (et seulement ceux-ci). Nous proposons d'envoyer les sondes à intervalles d'un RTT du chemin concerné : cela nous permet à la fois de ne pas dépendre d'une constante arbitraire et de rester proche de ce que l'utilisateur peut attendre.

Comme pour TCP, nous pouvons estimer le temps que doit mettre le serveur à répondre (RTO), pour chaque chemin. Si un chemin ne répond pas, c'est qu'il est en panne, qu'un paquet a été perdu ou que le RTT du chemin a augmenté.

Pour survivre à la panne supposée d'un chemin qui ne répond pas, nous augmentons le RTT de ce chemin par le temps d'attente de la réponse. Il suffit alors d'attendre que le RTT de ce chemin dépasse le RTT d'un autre chemin pour que ce dernier soit utilisé : la fiabilité est assurée. Remarquons qu'un chemin en panne est toujours sondé : dès que le chemin est rétabli, les sondes peuvent passer et le chemin peut être à nouveau utilisé.

Dans le cas d'une augmentation ponctuelle de RTT ou d'une perte occasionnelle de paquet, nous ne voulons pas changer la valeur d'origine du RTT, ni surestimer le temps d'attente. D'une part, nous avons donc deux variables séparées : une pour le RTT et une pour le temps d'attente. D'autre part, pour estimer le temps d'attente de manière asynchrone, c'est-à-dire sans avoir besoin de réagir immédiatement à l'expiration d'un RTO, nous ajoutons un RTO au temps d'attente — ce qui coïncide avec l'estimation du cas synchrone.

Optimisations Les mécanismes de base que nous venons de voir n'utilisent qu'un seul chemin pour les données — les autres chemins ne sont que sondés. Or, l'expérience que nous avons eue avec l'implémentation de ces mécanismes dans mpmosh (section 5.5) a révélé que l'application met plus de temps à réagir dans deux cas : lorsque le chemin de moindre latence tombe en panne, et lorsque ce chemin est fortement sujet aux pertes de paquets.

Nous améliorons le comportement des applications légères confrontés à ces deux cas en dupliquant certains paquets sur différents chemins. Nous allons voir que la duplication des paquets est relativement peu exploitée dans les protocoles existants — qui cherchent généralement à optimiser le débit. Ces mécanismes sont efficaces pour le cas de mpmosh (section 5.6).

Duplication des paquets dans les protocoles existants La duplication des paquets de données n'est que peu utilisée dans les protocoles que nous connaissons. Avec Happy Eyeballs, le client duplique les demandes de connexions pour accélérer la vitesse de connexion au serveur. Cette duplication n'apparaît qu'à l'initialisation de la connexion, et aucun paquet contenant des données de l'application n'est dupliqué.

MPTCP, en principe, ne duplique pas les paquets envoyés. Toutefois, pendant que les fenêtres s'ajustent, les paquets envoyés sur un chemin lent ayant une trop grosse fenêtre peuvent être dupliqués sur un chemin rapide ayant suffisamment de place dans sa fenêtre de congestion. Cela évite à MPTCP d'attendre un paquet bloqué par un flot plus lent — TCP garantit que les paquets arrivent dans l'ordre. Par ailleurs, lorsque MPTCP considère un chemin comme *potentiellement en panne*, tous les segments non acquittés envoyés sur ce chemin sont réémis sur d'autres chemins. Ces duplications ne sont pas adaptées à une application légère qui utilise déjà le chemin le plus rapide pour tous ses paquets.

M/TCP est un autre TCP multichemin pour lequel a été proposé la duplication des acquittements et des segments réémis pour améliorer son comportement sur des chemins sujets aux pertes [RA04]. Les acquittements sont systématiquement dupliqués tandis que les réémissions ne sont dupliquées que s'il y a assez de place dans la fenêtre de congestion. La seule duplication des acquittements suffit à sensiblement améliorer le comportement de M/TCP. Là encore, ces mécanismes ne sont pas adaptés à notre cas puisque nous cherchons à dupliquer les données, et que seules les données retransmises sont dupliquées.

Duplication des paquets pour une application légère et interactive Notre problème est très différent : nous cherchons à avoir le meilleur temps de réponse — et non le plus gros débit. Premièrement, pour optimiser la vitesse de convergence lors d'une panne, nous dupliquons le paquet de données sur le premier chemin actif. Comme nous l'avons vu, un chemin est détecté inactif lorsqu'il n'a pas répondu rapidement,

mais il faut attendre que son RTT soit supérieur au RTT d'un autre chemin pour que cet autre chemin soit utilisé. Dupliquer sur le premier chemin actif a deux intérêts : d'une part, si le chemin est bien inactif, alors un nouveau chemin prend immédiatement le relais, d'autre part, si le chemin est faussement détecté inactif (par exemple à cause d'une perte occasionnelle de paquets), alors il n'est pas disqualifié — nous continuons d'utiliser le meilleur chemin.

Deuxièmement, dupliquer le paquet suivant un paquet perdu ne lutte pas efficacement contre les pertes de paquets : c'est le paquet perdu qu'il aurait fallu dupliquer. Nous cherchons donc à limiter l'effet des pertes de paquets en dupliquant systématiquement les paquets sur plusieurs chemins. Pour ne faire cela qu'en cas d'utilité (quand le chemin est effectivement sujet aux pertes), nous mesurons le taux de pertes des chemins. Ceux-ci sont alors triés (toujours par RTT croissant) avant d'envoyer une donnée : les chemins les plus rapides sont utilisés jusqu'à ce que la probabilité que le paquet ne soit pas perdu soit "suffisante".

Nous implémentons (section 5.5) et évaluons (section 5.6) ces deux mécanismes dans le cadre de mpmosh et nous montrons qu'ils participent à une meilleure expérience de l'utilisateur.

5.5 Implémentation dans mpmosh

Nous décrivons dans cette section les détails techniques de nos mécanismes et leur intégration dans mosh. Nous motivons notre étude en exhibant les limitations de mosh dans un réseau *multihomé*. Nous introduisons ensuite mosh et détaillons le fonctionnement des couches basses de son transport. Ensuite, nous montrons comment mpmosh détecte et construit les chemins, comment il les évalue et enfin comment il utilise plusieurs chemins pour optimiser le comportement de l'application dans certains cas. Enfin, nous donnons quelques détails sur la modification du protocole et sur les API utilisées.

5.5.1 Motivations pour un mosh multichemin

Mosh (*The mobile shell*) est un émulateur de terminal distant. Comparé à ssh, mosh augmente l'expérience utilisateur de plusieurs manières. Grâce à son l'implémentation de son transport au-dessus d'UDP, il reste utilisable sur les réseaux à fort taux de pertes, il résiste aux changements d'adresses IP du client et il peut garder la session ouverte même sans connexion : dès qu'une connexion sera disponible, la session sera réutilisable. De plus, mosh peut être déployé sans modification du noyau : il utilise UDP et il est entièrement implémenté en espace utilisateur — il ne nécessite donc pas de droits administrateurs pour s'exécuter.

Toutefois, mosh est conçu pour des réseaux où un seul chemin existe. Il n'est pas capable de distinguer différents chemins et est susceptible d'utiliser le moins performant des chemins. Le client mosh est capable de s'adapter si l'hôte a une seule adresse et qu'il en change. Dans le cas où l'hôte a plusieurs adresses, le client mosh ne change pas d'adresse tant qu'elle reste attribuée à l'hôte, même en cas de panne sur le chemin correspondant. Le serveur, lui, ne change jamais d'adresse. Enfin, puisque l'adresse du serveur ne change pas, mosh ne peut passer d'IPv4 à IPv6 ou inversement. Il existe donc des cas pour lesquels mosh ne survit pas aux pannes.

Les solutions existantes telles que Shim6 ou MPTCP ne sont pas adaptées aux besoins de mosh pour des questions de déploiement et d'optimisation. Shim6 permet à mosh de garder les mêmes mécanismes tout en répondant au problème des pannes. En revanche, en absence de panne, le chemin utilisé est gardé : un meilleur chemin ne sera pas emprunté. De plus, nous ne connaissons pas de déploiement de Shim6. En présence d'un seul chemin, MPTCP se comporte comme TCP, et mosh évite TCP pour pouvoir définir ses propres mécanismes de transport qui lui valent les avantages que nous venons de mentionner. Il est clair que nous ne voulons pas utiliser MPTCP pour mosh.

1 bit	direction	} 64 bits nonce
63 bits	sequence number	
16 bits	timestamp	
16 bits	timestamp reply	

FIGURE 5.3 – Entête de la couche réseau de mosh.

5.5.2 Introduction à mosh

Mosh, *The mobile shell*, est un émulateur de terminal distant, léger et réactif qui optimise la survivabilité des sessions. Il diffère de SSH par deux aspects. D'une part, il prédit ce qui devrait s'afficher sur le terminal en attendant une réponse du serveur, ce qui augmente l'expérience utilisateur sur les liens de mauvaises qualités. D'autre part il est construit au-dessus d'UDP, avec un algorithme limitant l'impact de la perte de paquets et du réordonnancement.

Mosh est structuré en plusieurs couches : la couche *façade (front-end)*, la couche *transport* et la couche *réseau*. La couche façade interagit avec l'utilisateur, côté client, et avec l'hôte distant, côté serveur. Elle implémente l'algorithme de prédiction, et communique avec le pair par la couche transport.

La couche transport s'occupe de maintenir la synchronisation d'état entre les deux pairs, mais aussi de la segmentation des messages. Chaque modification reçue de la couche utilisateur donne lieu à un nouvel état, avec un numéro de séquence associé. La couche transport conserve l'ensemble des états non acquittés par le pair, et les envoie tous à chaque itération. Ainsi, même si un paquet se perd, le suivant contient ses informations, et aucune retransmission n'est requise. Afin de savoir si les paquets doivent être segmentés, la couche transport a besoin de connaître combien d'octets elle peut envoyer en un paquet : il s'agit du MTU — ou plus précisément du PMTU (*Path Maximum Transmission Unit*). Le MTU est obtenu auprès de la couche réseau avant chaque envoi.

La couche réseau reçoit et envoie les paquets sur le réseau, mais elle fournit aussi des informations aux couches supérieures : le MTU, ou l'état de la connexion (si elle est perdue ou non). Pour détecter une perte de connexion, mosh utilise le calcul classique du RTO (Retransmission TimeOut) de TCP. Il s'agit du temps au bout duquel le serveur aurait dû répondre. Le calcul du RTO requiert une estimation du temps d'un aller-retour : le RTT (Round Trip Time).

MPmosh (multipath mosh) est mon adaptation de mosh aux réseaux *multihomés*. Les modifications que nous apportons à mosh se confinent à la couche réseau de mosh. Celle-ci implémente déjà un mécanisme classique d'estimation du RTT que nous réutilisons. Avant de montrer les modifications que nous introduisons, nous décrivons la couche réseau de mosh.

5.5.3 Couche réseau de mosh

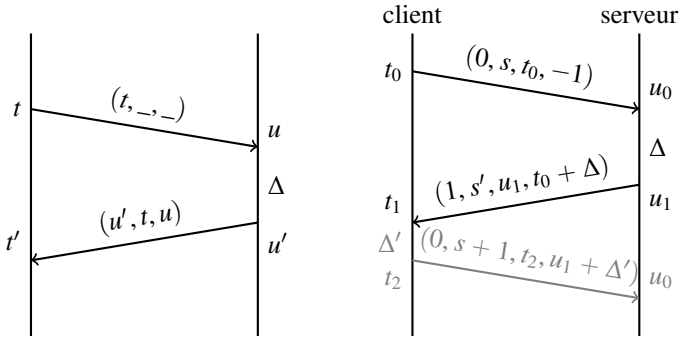
Dans cette section, nous voyons quels sont les mécanismes de couche réseau de mosh (sans mes changements). L'entête de protocole réseau de mosh (classique) est décrit en figure 5.3 : il est constitué de 12 octets (96 bits). Les premiers 64 bits contiennent un nonce cryptographique. Deux paquets de contenu différent doivent impérativement avoir des nonces différents. Chaque nonce est constitué d'un bit indiquant la direction du paquet (client vers serveur ou l'inverse), et d'un numéro de séquence, propre à chaque pair, strictement croissant. Les 32 bits restants sont deux horodatages de 16 bits utilisés pour le calcul du RTT.

Mesure du RTT La mesure du RTT dans mosh utilise les techniques classiques utilisées par Mills dans le protocole HELLO [Mil83] et dans le protocole de synchronisation d'horloge NTP [MMBK10] [2]. Comme illustré en figure 5.4*a*, le client envoie un message avec l'horodatage de l'envoi du paquet *t* que le serveur

2. Babel l'utilise aussi dans une de ses extensions [JBC14].

reçoit au temps u de son horloge. Le serveur répond, de manière asynchrone, au temps u'. La réponse contient les horodatages t, u et u'. Lorsque le client reçoit le paquet, il peut calculer le RTT comme ci-dessous.

$$RTT = (t' - t) - (u' - u)$$



(a) Algorithme de Mills *(b) Application à mosh*

FIGURE 5.4 – Un échange à la couche réseau de mosh.

Mosh utilise une version légèrement différente, mais équivalente : le serveur, au lieu d'envoyer à la fois t, u et u', envoie t décalé de $u' - u$ *(timestamp reply)*, et u' *(timestamp)*. Le client calcule alors le RTT comme la différence entre le temps t' de réception du message et le *timestamp reply* : $RTT = t' - (t + (u' - u))$.

La figure 5.4*b* représente ce processus de manière plus précise, à travers un échange complet de la couche réseau de mosh. Le client envoie un paquet destiné au serveur (direction = 0), avec un numéro de séquence s, un *timestamp* t_0 et un *timestamp reply* -1. La valeur de -1 est utilisée car il s'agit du premier paquet envoyé : le client n'a pas encore reçu d'horodatage du serveur. Le serveur reçoit le paquet au temps u_0 : il sauvegarde à la fois t_0 et u_0 comme dernier *timestamp* reçu du client. Plus tard, au temps u_1, il décide de répondre au client et lui envoie des données : il envoie un paquet à destination du client (direction = 1), avec son prochain numéro de séquence s', le *timestamp* u_1, et le *timestamp reply* $t'_0 = t_0 + \Delta = t_0 + (u_1 - u_0)$. À la réception, le client calcule la valeur du RTT : $RTT = t_1 - t'_0$; il sauvegarde aussi les deux horodatages pour les prochains messages envoyés.

Le RTT peut varier fortement d'un paquet à l'autre. Il suffit pour cela qu'un paquet, même isolé, soit retardé : le calcul du RTT donne lieu à une augmentation immédiate. Pour obtenir une certaine stabilité tout en convergeant rapidement vers la valeur courante du RTT, les valeurs des RTT sont moyennées à l'aide d'une moyenne exponentielle. Mosh calcule aussi la moyenne de variance du RTT, relativement au SRTT. La moyenne du RTT, sa variance, et la moyenne de sa variance s'appellent respectivement SRTT (*Smoothed RTT*), RTTVAR et SRTTVAR :

$$\begin{aligned} RTTVAR_{n+1} &= |SRTT_n - RTT_{n+1}| \\ SRTTVAR_{n+1} &= (1-\alpha) * SRTTVAR_n + \beta * RTTVAR_{n+1} \\ SRTT_{n+1} &= (1-\alpha) * SRTT_n + \alpha * RTT_{n+1} \end{aligned}$$

avec α et β des constantes (dans l'implémentation courante, $\alpha = \frac{1}{8}$, et $\beta = \frac{1}{4}$).

Enfin, mosh prévient les attaques par duplication. Une telle attaque consiste à envoyer un duplicata exact d'un paquet précédent. Le duplicata est alors correctement déchiffré et authentifié par mosh, mais contient des horodatages obsolètes. Pour remédier à cela, mosh ne considère que les messages arrivés dans l'ordre [3],

3. Seulement pour ce qui concerne la couche réseau de mosh ; les données de couches supérieures sont toujours transmises.

c'est-à-dire les messages ayant un numéro de séquence strictement plus grand que le plus grand numéro de séquence jusqu'alors reçu.

Perte de connexion Le calcul du RTT est souvent utilisé pour détecter les pannes ou la perte d'un paquet car il faut au moins attendre un RTT après l'envoi d'un paquet pour espérer recevoir une réponse. Le protocole TCP, en l'absence de réponse, retransmet le paquet. Il calcule pour cela le RTO (le temps de retransmission — *Retransmission Timeout*), temps au bout duquel une réponse aurait dû arriver.

$$RTO = SRTT + 4 \times SRTTVAR$$

Mosh calcule aussi le RTO tel qu'il est défini dans TCP. Toutefois, le RTO n'est pas utilisé par la couche réseau de mosh, mais par les couches supérieures. En effet, la couche réseau de mosh est passive en émission : tant que les couches supérieures n'envoient pas de données, la couche réseau de mosh n'envoie pas de données non plus. C'est donc aux couches supérieures de décider quand envoyer des données et s'il faut envoyer les mêmes données. Elles peuvent aussi se servir du RTO pour avertir l'utilisateur d'une éventuelle coupure de la connexion.

Résistance aux coupures Plusieurs causes possibles peuvent être à l'origine d'une perte de connexion : problèmes dans le réseau, lien débranché, changement d'adresse IP, ou expiration du NAT. Mosh fait de son mieux pour garder la connexion, et n'abandonne jamais sans requête explicite de l'utilisateur.

Si la connexion expire, des paquets de données continuent d'être envoyés au serveur, mais à des intervalles de temps plus espacés. Lorsque la connectivité est retrouvée, mosh reconverge presque immédiatement.

Mosh résiste aux changements d'adresses IP car il utilise l'appel système `sendto`, sans être lié à une adresse : pour chaque paquet, la responsabilité de trouver une adresse est laissée au système (celui-ci utilise l'algorithme par défaut de sélection d'adresses que nous avons évoqué). Lorsque le système change d'adresse IP, les paquets sont alors automatiquement envoyés avec la nouvelle adresse. Il est donc possible d'utiliser la même session de mosh à plusieurs endroits différents (par exemple après avoir transporté un ordinateur portable).

5.5.4 MPmosh : détection et construction des chemins

Nous avons vu qu'un chemin est caractérisé par une paire d'adresses source et destination. Pour que notre application bénéficie de l'utilisation de plusieurs chemins, elle doit d'abord les découvrir. Nous voyons dans cette partie comment le client et le serveur mpmosh ont des rôles asymétriques dans cette découverte, et comment les différentes paires d'adresses sont découvertes. Nous voyons aussi que certaines associations sont impossibles et donc évitées, et discutons des limitations de l'implémentation [4].

Une application client-serveur Les rôles du client et du serveur de mosh sont asymétriques. Dans la version classique de mosh, seul le client peut changer d'adresse ou de port. Le serveur ne fait que répondre au client, par le dernier port et la dernière adresse connus. Lorsqu'une perte de connexion est détectée, le client persiste à envoyer régulièrement des messages au serveur au cas où la connexion reviendrait, alors que le serveur devient inactif. Autrement dit, le client prend des initiatives, alors que le serveur ne fait que répondre et s'adapter au client.

Nous conservons cette distinction dans mpmosh : seul le client est à l'origine d'un échange de sondes, et seul le client peut décider d'établir un nouveau chemin. Le serveur quant à lui se contente d'utiliser les

4. https ://github.com/boutier/mosh — licence GPLv3+

chemins que le client utilise. Il n'a donc pas besoin, en particulier, d'établir la liste des paires d'adresses possibles : il les apprend à la réception d'un paquet en provenance du client.

Puisque le client est à l'origine des différents chemins, il doit collecter ses propres adresses ainsi que celles du serveur.

Collecte des adresses Généralement, lorsqu'un serveur a plusieurs adresses, celles-ci sont stockées dans le DNS. Le client peut alors retrouver une liste d'adresses accessibles depuis l'emplacement actuel du client. Un programme peut retrouver les adresses d'un serveur depuis le DNS à partir de son nom de domaine avec la fonction `getaddrinfo`. S'il ne connaît pas le nom du serveur, mais une de ses adresses, il peut retrouver le nom du serveur avec `getnameinfo`, avant de retenter `getaddrinfo` pour obtenir les adresses alternatives.

Par ailleurs, un programme peut retrouver les adresses locales du nœud sur lequel il s'exécute par l'API standard du système, malheureusement celles-ci diffèrent en fonction des systèmes. La plupart des Unix disposent de la fonction `getifaddrs`. Ainsi, à la fois le client et le serveur de mpmosh connaissent la liste de leurs adresses. Celle-ci est mise à jour régulièrement.

Au démarrage de mpmosh, le client possède déjà une adresse du serveur, fournie en argument et qui a déjà été utilisée par ssh pour se connecter au serveur : un chemin valide existe avec cette adresse. Le client construit un premier jeu de chemins avec cette adresse du serveur et ses propres adresses locales.

Au cours de l'exécution de mpmosh, le client demande régulièrement au serveur quelles sont ses adresses. En particulier, aussitôt qu'il a constitué son premier jeu de chemin, il envoie une requête d'adresses sur tous ses chemins. Ainsi, il obtient rapidement un ensemble complet des chemins possibles. À la fois la requête et la réponse d'un message contenant des adresses sont étiquetées avec le bit `ADDR_FLAG`.

Filtrage des adresses Notre implémentation ne fait qu'un filtrage très limité des paires d'adresses valides. D'une part, nous vérifions que les deux adresses sont bien de la même famille (IPv4 ou IPv6). D'autre part, nous nous assurons que les deux adresses sont des adresses *loopback* ou que les deux adresses sont des adresses locales au lien, ou qu'aucune des deux adresses n'est *loopback* ou locale au lien.

Certaines heuristiques existent pour limiter le nombre de mauvaises associations, en particulier en se basant sur l'appartenance à un préfixe commun. Nous n'avons pas cherché dans cette direction, et nous nous contentons d'avoir un filtrage naïf, afin de ne pas avoir de faux négatif. Nous voyons aussi en section 5.5.5 (figure 5.5) que le surcoût est, à notre avis, acceptable.

5.5.5 Évaluation des chemins

L'évaluation des chemins se fait par l'envoi périodique de petits messages appelés *sondes* et seulement constitués de l'entête de couche réseau de mpmosh. L'utilisation de sondes par l'application permet de mesurer la métrique que l'application veut optimiser sur l'intégralité du chemin, de bout en bout. Dans mpmosh, nous cherchons d'abord à optimiser, comme nous l'avons dit, le RTT. Dans cette section, nous présentons la manière dont nous avons intégré nos mécanismes à la boucle à événement principale de (mp)mosh, l'évaluation du RTT de chaque chemin par mpmosh, le surcoût engendré par les sondes, la procédure d'envoi des sondes, et enfin les problèmes liés aux instabilités (changements de chemins).

Intégration à la boucle à événement La boucle à événement de mosh appelle la couche réseau soit lorsqu'elle veut envoyer des données, soit lorsqu'un paquet est arrivé. Il revient aux couches supérieures de mosh de renvoyer des données, de pallier les pertes de paquets et de choisir la fréquence d'envoi des paquets.

La couche réseau de mpmosh ne change pas ce mécanisme et garde la distinction des rôles. En particulier, mpmosh ne sollicite pas de manière artificielle la boucle à événement, comme ça pourrait être le

cas pour l'envoi des sondes à l'image des *heartbeats* SCTP. Au contraire, la couche réseau de mpmosh attend l'émission de données pour émettre en même temps des sondes, si besoin. Nous voyons dans les paragraphes suivants comment nous évaluons les différentes constantes de temps, qui sont naturellement influencées par les délais induits par la boucle à événement, notamment le temps d'inactivité et la période d'envoi des sondes.

Évaluation du RTT La couche réseau de mpmosh contient, comme celle de mosh dont elle est une extension, toutes les informations nécessaires pour calculer le RTT. Le même mécanisme que celui utilisé par mosh et décrit en section 5.5.3 est utilisé pour mesurer le SRTT. L'utilisation de sondes permet donc d'obtenir dynamiquement la valeur du SRTT sur tous les chemins et de choisir le meilleur.

Cependant, ce mécanisme est incapable de détecter les pertes de connexion : lorsqu'un chemin est coupé, le RTT n'est plus mis à jour. Ce comportement est critique lorsque le meilleur chemin est coupé, car, alors, ce chemin reste considéré comme étant le meilleur. Il est donc nécessaire d'évaluer le temps d'inactivité de chaque chemin.

Le temps d'inactivité d'un chemin est difficile à évaluer. Il ne s'agit pas simplement du temps depuis le dernier message reçu, mais plutôt du temps passé à attendre un message qui n'est pas arrivé. En particulier, si aucun message n'est attendu, le chemin ne peut pas être considéré inactif. Nous verrons dans les prochains paragraphes comment nous estimons le temps d'inactivité, que nous notons *idle_time*.

Les valeurs du SRTT et du temps d'inactivité sont gardées séparément. Lorsque le chemin redevient actif, il n'y a pas de raison que son RTT change. Ainsi, la valeur du SRTT d'un chemin inactif n'est pas changée, mais la valeur utilisée comme SRTT, en particulier pour choisir un chemin, devient la somme du SRTT avec le temps d'inactivité.

Évaluation du RTO et du temps d'inactivité Pour détecter une perte d'activité d'un chemin, nous utilisons le calcul classique du RTO. Remarquons simplement que cette fois-ci, nous estimons le RTO aussi en fonction du temps d'inactivité, qui vient s'ajouter au SRTT :

$$RTO := (SRTT + idle\_time) + 4 \times SRTTVAR$$

Les causes d'un RTO écoulé sont multiples : un ralentissement du réseau, une perte isolée de paquet ou un chemin coupé. Dès qu'un RTO est écoulé sans réponse, le client de mpmosh renvoie une sonde et le temps d'inactivité est augmenté d'un RTO :

$$idle\_time := idle\_time + RTO$$

Le temps d'inactivité ne doit pas être augmenté de plus d'un RTO, même lorsque le dernier message envoyé est plus ancien. En effet, la couche réseau n'a pas le contrôle du programme lorsque le RTO expire, et il faut attendre un envoi d'un paquet par couches supérieures pour reprendre le contrôle. Prendre en compte ce temps d'attente dans le calcul du temps d'inactivité conduit à une surestimation qui peut être significative. Nous avons observé des retards de plusieurs secondes qui poussaient le client de mpmosh à choisir un chemin moins performant. Par ailleurs, augmenter le temps d'inactivité d'un RTO est suffisant : le calcul du RTO dépendant du temps d'inactivité, cela revient à doubler le RTT à chaque fois. Remarquons que dans des situations idéales (variance du RTT nulle, aucune attente), cela correspond exactement au temps d'attente.

Enfin, mosh utilise des acquittements retardés. Pour compenser, la couche réseau de mpmosh reçoit le retard maximum autorisé des couches supérieures (*max_delack*), et calcule un RTO retardé, le dRTO (*delayed RTO*). Le dRTO remplace l'usage habituel du RTO, sauf dans le calcul du temps d'inactivité : le délai ajouté par les acquittements retardés n'est pas représentatif du temps réel d'inactivité.

Période d'envoi (ms)	100	500	1000	10000
Surcoût en IPv6 (B/s)	900	180	90	9
Surcoût en IPv4 (B/s)	720	144	72	7.2

FIGURE 5.5 – Surcoût d'une sonde (coût de la trame ethernet).

Surcoût des sondes L'utilisation continue de sondes pour mesurer la qualité des chemins surcharge le réseau. Le nombre de chemins étant le produit du nombre d'adresses locales et du nombre d'adresses distantes, cette surcharge pourrait en principe croître à des valeurs importantes.

La figure 5.5 représente le surcoût induit par un chemin, en fonction de la période d'envoi. Le coût d'une trame a été vérifié empiriquement (avec *wireshark*) : une sonde est contenue dans une trame Ethernet de moins de 90 octets. Des chemins ayant pour période de sonde 10 s et 500 ms surchargent respectivement le réseau à hauteur de moins de 10 o/s et 200 o/s.

Considérant un client et un serveur ayant chacun quatre adresses compatibles entre elles, 16 chemins sont construits. Si chacun de ces chemins est sondé toutes les 500 ms, le surcoût total est de 3 kB/s, contre 144 o/s s'ils sont sondés toutes les 10 s.

Nous considérons que dans le cas de mosh, envoyer des sondes toutes les 500 ms est suffisant pour ne pas impacter l'expérience utilisateur, contrairement bien sûr à envoyer des sondes toutes les 10 s. Cependant, nous pensons que les sondes ne doivent pas être envoyées à intervalles fixés et prédéfinis. En particulier, un chemin totalement inactif ne devrait pas être sondé souvent.

Procédure d'envoi et période d'envoi des sondes MPmosh n'interrompt pas la boucle à événement pour envoyer des sondes, mais attend un envoi de données des couches supérieures. À cette occasion seulement, des sondes sont envoyées sur les chemins qui n'ont pas été sondés depuis un certain temps. Pour cela, nous associons à chaque chemin le temps auquel envoyer la prochaine sonde. Si ce temps est dépassé, une sonde est envoyée sur le chemin correspondant et ce temps est mis à jour.

Plus précisément, lorsque les couches supérieures décident d'envoyer un message, les chemins sont triés par SRTT croissants. Le premier chemin (qui a la meilleure latence) est utilisé pour transmettre ce message tandis que les suivants sont utilisés pour l'envoi d'une sonde si le temps d'envoi de sonde associé au chemin est dépassé.

Au moment où une sonde est envoyée, le temps d'envoi de la prochaine sonde est calculée. Afin de limiter la période d'envoi des sondes, celle-ci n'est pas fixée, mais dépendante de la latence du chemin : nous pensons que les sondes devraient être envoyées à la période d'un RTT. En effet, nous pensons que, d'un point de vue de l'utilisateur, attendre une réponse du serveur ou la mise à jour du RTT d'un chemin est similaire. Comme précédemment, nous utilisons pour cela la valeur du SRTT ajoutée à celle du temps d'inactivité. Ainsi, l'envoi de sondes sur les chemins inactifs est limité.

Enfin, nous bornons la période d'envoi de sondes. D'une part, il n'est pas nécessaire de sonder un chemin trop souvent, car l'expérience utilisateur n'en est pas améliorée, et d'autre part il faut sonder au moins régulièrement les chemins inactifs pour être sûr de détecter la réactivation d'un chemin. Dans notre implémentation, nous fixons la borne inférieure à 500 ms, et la borne supérieure à 10 s.

$$probe\_interval := \min(\max(dRTO, 500\,\text{ms}), 10\,\text{s})$$

5.5.6 Optimisations des performances de mpmosh

Jusqu'ici, nous avons présenté une version non optimisée de mpmosh. L'application est capable d'évaluer la latence des chemins, et de choisir dynamiquement celui qui a la meilleure latence, avec un surcoût acceptable. Cette version a deux limitations. D'une part, le temps de convergence d'un chemin vers un autre

peut être suffisamment long pour être ressenti. D'autre part, l'utilisation d'un chemin à faible latence mais fortement sujet aux pertes peut donner lieu à des résultats médiocres, voire moins bons qu'un chemin à plus forte latence non sujet aux pertes.

Une solution très simple suffit à résoudre ce problème : dupliquer systématiquement les paquets sur tous les chemins (et ne plus utiliser de sondes). Cette solution est applicable dans le cas de mosh car peu de données sont transmises, mais d'une part elle n'est pas généralisable à des applications un peu plus gourmandes, et d'autre part elle n'est pas très satisfaisante, surtout lorsque la duplication n'est pas nécessaire — ce qui est le cas lorsque le chemin à plus faible latence est sans perte. Nous proposons donc ici une solution plus modérée constituée de deux mécanismes indépendants pour résoudre ces problèmes.

5.5.6.1 Duplication sur le premier chemin actif

La duplication sur le premier chemin actif sert à résoudre le problème de la convergence. Nous avons vu qu'un chemin est détecté inactif dès lors qu'il excède un RTO : un chemin coupé est détecté inactif très vite. Lorsque le chemin primaire est détecté inactif, nous dupliquons le paquet de données sur le chemin actif de meilleure latence. Ainsi, lorsque le meilleur chemin A est coupé, un autre chemin B est utilisé en parallèle pour les paquets de données, jusqu'à ce que A soit considéré comme moins bon que B.

Cette solution n'apporte donc aucun surcoût lorsque le meilleur chemin est actif, et un surcoût très limité d'une seule duplication par envoi dans le cas contraire. Cette duplication est temporaire : elle s'arrête dès que le nouveau chemin devient le meilleur. Si le chemin est détecté inactif à tort, simplement à cause d'un retard ou d'une perte exceptionnelle de paquets, la duplication n'est que ponctuelle.

Cependant, cette solution n'est pas suffisante lorsque les chemins disponibles sont tous sujets aux pertes. D'une part, si un paquet se perd, il est possible que l'envoi suivant soit dupliqué, mais ce n'est alors pas le paquet perdu qui est dupliqué : la duplication a été effectuée trop tard. D'autre part, si tous les chemins sont sujets aux pertes, il est possible qu'à un moment donné ils soient tous détectés inactifs, en quel cas aucun duplicata n'est envoyé.

5.5.6.2 Utilisation de plusieurs chemins simultanément

Afin d'optimiser les performances sur les chemins à forts taux de pertes, nous dupliquons les paquets sur différents chemins, en s'efforçant de ne le faire que si nécessaire. Pour cela, nous évaluons constamment le taux de pertes des chemins, en plus du RTT, et nous modifions la procédure d'envoi.

Évaluation du taux de pertes Le format de couche réseau de mpmosh comporte un numéro de séquence propre à chaque chemin, qui augmente de façon incrémentale à chaque envoi. Le récepteur des paquets est donc capable de détecter les paquets perdus en repérant les numéros de séquence manquants. Nous utilisons pour cela une fenêtre coulissante. L'avantage d'une fenêtre coulissante est en particulier de pouvoir différencier les paquets perdus des paquets réordonnés : un paquet réordonné n'est pas considéré comme perdu, et la mesure du taux de pertes des paquets n'en est que plus précise.

Le taux de pertes ainsi mesuré est celui des paquets entrants par un chemin, alors que celui qui nous intéresse est celui des paquets sortants. Étant donné un chemin, le taux de pertes des paquets sortants pour un pair est le taux de pertes des paquets entrants pour l'autre pair. Chaque pair communique donc à l'autre le taux de pertes des paquets entrants qu'il a calculé pour chaque chemin (8 bits sont réservés à cet effet dans le protocole de couche réseau de mpmosh).

Diminution du taux de pertes par duplication Le taux de perte qui résulte de la duplication des paquets sur plusieurs chemins indépendants est le produit du taux de pertes de ces chemins. Par exemple, nous nous

attendons à ce que l'utilisation simultanée de deux chemins perdant un paquet sur deux donne lieu à une perte moyenne d'un paquet sur quatre.

Pour maintenir le taux de pertes en dessous d'un certain seuil, nous modifions la procédure d'envoi comme illustré sur l'algorithme 10. Les chemins sont triés par meilleure latence (comme avant) et en cas d'égalité par meilleur taux de pertes. Pour chaque chemin (ligne 3), le paquet est envoyé (ligne 4) et le taux de pertes attendu est mis à jour (ligne 5), et cela jusqu'à ce que le seuil de taux de pertes acceptable soit atteint (lignes 6 et 7). Le paquet est donc dupliqué à partir du deuxième envoi.

```
1 threshold : some value between 0 and 1
2 exp_loss_ratio = 1
3 for all path
4 |   send_data(path)
5 |   exp_loss_ratio *=
  |     path.outgoing_loss_ratio
6 |   if exp_loss_ratio ⩽ threshold
7 |   |   break
```

Algorithme 10 : Envoi en fonction du taux de pertes.

Le surcoût de cette optimisation dépend bien évidemment du taux de pertes des différents chemins et du seuil choisi. Le point intéressant est de voir que si le chemin de meilleure latence est sans perte, aucun paquet n'est dupliqué. Bien sûr, si tous les chemins sont à forts taux de pertes, et avec un seuil de 0%, il est possible que cette solution soit équivalente à dupliquer les paquets sur tous les chemins, mais il s'agit alors du comportement désiré.

5.5.6.3 Limitations

Ces optimisations présentent quatre limitations : un surcoût accru, une augmentation du risque de collisions, la question du choix du MTU et une mauvaise gestion de la congestion.

Surcoût En dupliquant les paquets, un surcoût supplémentaire, plus important que celui des sondes, est introduit. Cependant, nos mécanismes essaient autant que possible de garantir que le surcoût induit n'est pas inutile. Remarquons simplement qu'en triant les chemins par taux de pertes croissants avant de les trier par RTT, nous pourrions limiter la charge, mais diminuer les performances. Bien sûr, des compromis entre les deux pourraient être cherchés.

Choix du MTU Choisir le MTU est une question difficile lorsque plusieurs chemins doivent être utilisés pour transiter un paquet. Notre implémentation donne aux couches supérieures la valeur du MTU du chemin de plus faible latence. On pourrait choisir de donner le MTU minimum, au risque de segmenter davantage et inutilement sur d'autres chemins.

Mauvaise gestion de la congestion Cette implémentation a cependant un vrai problème : elle suppose que les performances des différents chemins sont indépendantes. Pourtant, si tous les chemins passent par un même goulot d'étranglement, responsable de la perte de paquets à cause de congestion, dupliquer ne fait qu'ajouter davantage de paquets dans ce goulot, ce qui ne fait qu'augmenter la perte de paquets de tous les chemins. Il serait intéressant d'étudier comment pallier ce problème. Cependant, nous ne l'avons pas rencontré et nous ne pensons pas que ce soit un problème pertinent pour mosh.

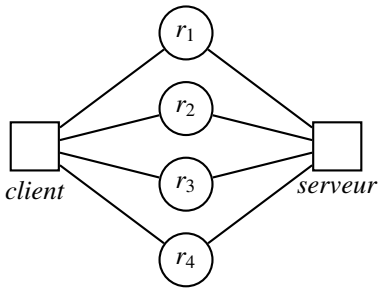


FIGURE 5.6 – Topologie émulée.

5.6 Résultats expérimentaux

Dans cette section, nous montrons dans quelle mesure l'utilisation du multichemin dans mosh nous permet de mieux résister aux pannes, de garantir la meilleure latence, et de pallier de forts taux de pertes.

Pour cela, nous testons simultanément mosh et mpmosh dans un environnement virtuel émulé avec quatre chemins, comme représenté en figure 5.6. Le réseau est constitué de deux hôtes, le client et le serveur, ainsi que de quatre routeurs, un par chemin. Les routeurs peuvent simuler la qualité des liens en ajoutant de la latence ou en perdant volontairement des paquets. L'intérêt d'un réseau émulé est de pouvoir précisément connaître le taux de pertes et la latence du chemin tout en utilisant l'application de manière réaliste.

Nous testons trois versions de mosh :

— *std* : la version standard de mosh ;
— *mp* : mpmosh avec optimisation de la convergence et optimisation du taux de pertes ;
— *mp-loss* : mpmosh avec optimisation de la convergence mais sans optimisation du taux de pertes.

Nous présentons ici quatre expériences. La première vise à montrer que mpmosh ne fait pas moins bien que mosh dans le cas où tout se présente bien, c'est-à-dire quand les chemins sont égaux et sans perte. La deuxième évalue la capacité de mpmosh à résister aux pannes. La troisième évalue la résistance aux pertes de paquets par duplication. Enfin, la quatrième est une expérience longue qui comporte quatre phases. Les caractéristiques des chemins sont plus variées que dans les trois premières expériences.

5.6.1 Chemins égaux et sans perte

Le but de cette expérience est de montrer que mpmosh n'a pas de moins bonnes performances que mosh, et c'est le cas (*réactivité moyenne* sur la figure). Dans cette expérience, chaque chemin a un surcoût identique, et aucun chemin n'est sujet aux pertes : quel que soit le chemin choisi, les performances de l'application sont identiques, et il n'y a aucun intérêt à dupliquer les paquets. En particulier, la version classique de mosh emprunte nécessairement le meilleur chemin.

Le tableau ci-dessous représente les résultats des deux expériences, chacune durant 180 secondes. La première colonne indique la latence ajoutée (dans les deux sens) et le taux de pertes des chemins : une paire $(300\,\text{ms}, 0\%)$ indique qu'un chemin retarde les paquets le traversant de 300 ms (soit un RTT de 600 ms) et perd 0% des paquets. La seconde colonne indique la version de mosh utilisée. Les trois dernières colonnes représentent respectivement le temps moyen attendu par un utilisateur avant d'avoir une réponse, le nombre d'octets envoyés par le client (à la couche application), et le nombre de paquets envoyés.

		Données et sondes		
		réactivité moyenne (ms)	données envoyées par le client (B)	nombre d'échantillons
(0 ms, 0%)	mp	12.61	102451	2266
	mp-loss	12.48	102224	2263
	std	12.65	46137	614
(300 ms, 0%)	mp	1130	73543	1518
	mp-loss	1108	73602	1501
	std	1184	43969	588

Les résultats correspondent à nos attentes : mosh et mpmosh ont des performances similaires. Mais deux détails peuvent nous surprendre :

— la valeur de la réactivité moyenne de la seconde expérience est presque deux fois supérieure au RTT ajouté. En fait, ce comportement est normal car le serveur s'autorise à retarder l'envoi de messages avec un délai de l'ordre du RTT. La valeur du RTT elle-même est correctement évaluée.

— le client mpmosh génère beaucoup plus de trafic que mosh. En fait, la quantité de données envoyées est très faible, ce qui donne aux sondes une importance relative beaucoup plus grande. Cependant, la charge totale du trafic est très faible. En effet, les plus gros paquets que nous observons dans cette expérience font 90 B, le trafic moyen généré par le client mpmosh est compris entre 130 et 170 B/s hors sondes et jusqu'à 560 B/s avec les sondes.

5.6.2 Résistance aux pannes

Nous nous intéressons ici à la capacité de mpmosh à survivre aux pannes, avec l'optimisation d'accélération de la convergence (cf. section 5.5.6.1). Dans cette expérience, nous avons deux chemins fonctionnels, l'un meilleur que l'autre. Après un régime stable de 40 secondes, le meilleur des deux chemins est coupé. Le tableau ci-dessous donne les détails de l'expérience : les chemins 3 et 4 sont initialement coupés, et le chemin 2 a 100 ms de latence ajoutée. Au bout de 40 secondes, le chemin 1 est coupé.

	Chemin 1	Chemin 2	Chemin 3	Chemin 4
Phase 1 (40 s)	(300 ms, 0%)	(100 ms, 0%)	$\perp$	$\perp$
Phase 2 (60 s)	(0 ms, 100%)	(100 ms, 0%)	$\perp$	$\perp$

La figure 5.7 représente le détail de l'expérience correspondant à la rupture du chemin 1. Au moment de la panne (entre les secondes 49 et 50 sur le graphe), mosh perd la connexion. Au contraire, mpmosh, dans ses deux versions, emprunte le second chemin avec un temps de réaction de 200 ms. La transition est presque immédiate : elle a coûté 100 ms. En effet, le premier message après la panne est reçu avec 300 ms au lieu de 200 ms.

5.6.3 Résistance aux pertes de paquets

Le but de cette expérience est de mesurer les performances de l'optimisation de résistance aux pertes de paquets utilisée dans mpmosh telle que décrite en section 5.5.6.2. Rappelons que celle-ci est principalement basée sur la mesure des pertes de paquets de chaque chemin et sur la duplication des paquets sur plusieurs chemins si nécessaire. Nous refaisons l'expérience de la section 5.6.1 avec 70% de pertes sur tous les chemins. Les résultats sont présentés dans le tableau ci-dessous, sous la même forme que précédemment.

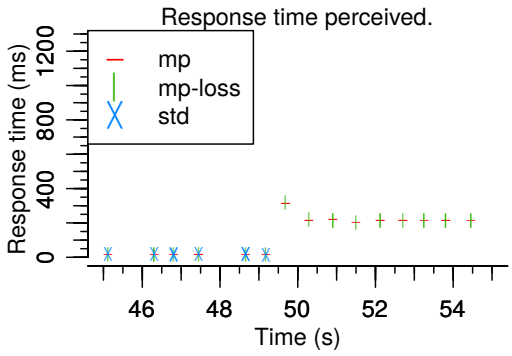


FIGURE 5.7 – Basculement vers un autre chemin après une panne.

		Données et sondes		
		réactivité moyenne (ms)	données envoyées par le client (B)	nombre d'échantillons
(0 ms, 70%)	mp	159.90	118675	1884
	mp-loss	531.30	157215	2465
	std	582.60	272332	3574
(300 ms, 70%)	mp	1242	139252	2163
	mp-loss	2163	83159	1433
	std	2286	39720	525

La duplication (*mp*) des paquets augmente sensiblement la réactivité de l'application. Dans la première expérience, nous obtenons un temps de réaction moyen de 160 ms pour la version optimisée de mpmosh (*mp*) contre 580 ms pour mosh, soit près de trois fois plus. Dans la deuxième expérience, nous obtenons un temps de réaction moyen de 1200 ms pour la version optimisée de mpmosh (*mp*) contre 2300 ms pour mosh.

Si toutes nos expériences reflètent ce gain de réactivité, nous notons tout de même quelque chose de particulièrement surprenant dans la première expérience, qui n'est pas systématique : la quantité de données émise par la version optimisée de mpmosh (*mp*) est nettement inférieure à celle des autres programmes. Pourtant, mpmosh dupliquant ses envois, nous nous attendions à l'inverse, comme c'est le cas dans la deuxième expérience, et dans la plupart des autres expériences. En fait, nous observons ici d'une part que mosh réémet des paquets de manière particulièrement agressive, et d'autre part que la taille moyenne des paquets de mosh (94 *B* de moyenne) est plus importante que la version multichemin (84 *B* de moyenne pour *mp*). Cela est dû au fait que *mosh* renvoie tout l'état non acquitté. En diminuant les pertes de paquets, la quantité d'état non acquitté est donc moindre.

Ce dernier point est particulièrement intéressant : dupliquer les paquets, en plus d'augmenter la réactivité de l'application, contribue à garder de plus petits paquets.

5.6.4 Expérience complexe

Cette expérience est composée de quatre phases. Pendant chaque phase, les propriétés des chemins sont fixées. Entre deux phases, nous faisons varier la latence additionnelle d'un chemin et son taux de pertes. Les caractéristiques des chemins durant les différentes phases sont décrites dans le tableau ci-dessous, sous forme de couples (latence, taux de pertes), comme précédemment. Les changements d'une phase sur l'autre sont indiqués en rouge.

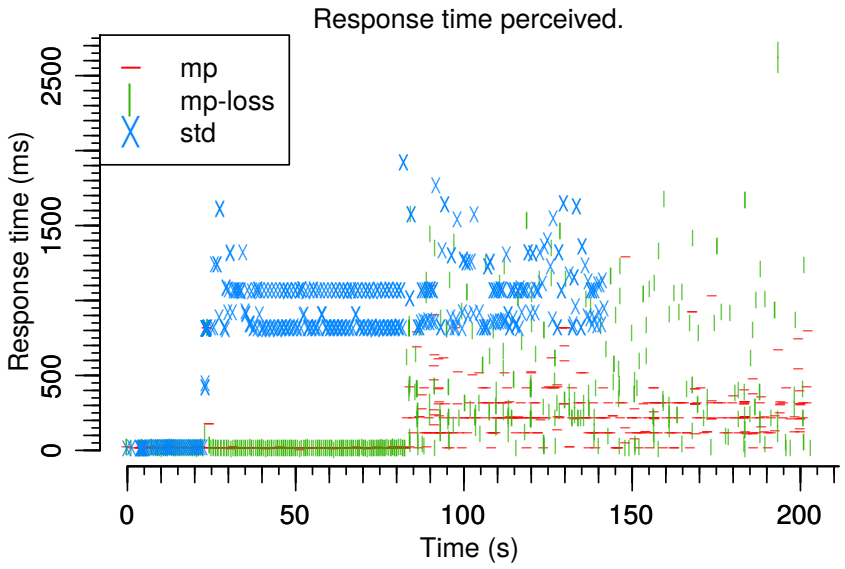


FIGURE 5.8 – Vue générale de l'expérience.

	Chemin 1	Chemin 2	Chemin 3	Chemin 4
Phase 1 (20 s)	(0 ms, 0%)	(50 ms, 0%)	(50 ms, 0%)	(50 ms, 0%)
Phase 2 (60 s)	(400 ms, 0%)	(200 ms, 0%)	(100 ms, 0%)	(0 ms, 0%)
Phase 3 (60 s)	(400 ms, 0%)	(200 ms, 20%)	(100 ms, 40%)	(0 ms, 70%)
Phase 4 (60 s)	(⊥, 100%)	(200 ms, 20%)	(100 ms, 40%)	(0 ms, 70%)

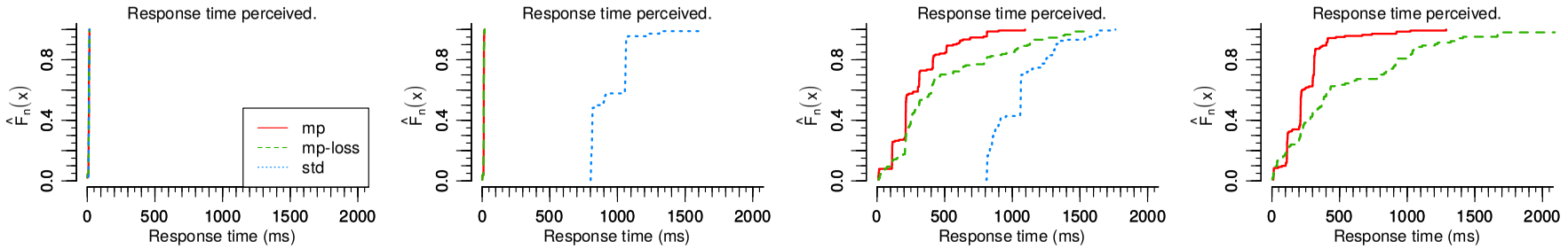
La figure 5.8 montre le temps de réponse pour chaque paquet envoyé (et arrivé) de l'ensemble de l'expérience. Nous voyons clairement les quatre phases, délimitées aux secondes 25, 85 et 145.

Nous observons que mpmosh est toujours meilleur que mosh. Les strates observables dans les phases 3 et 4 reflètent l'utilisation des différents chemins. Les points proches de 0 ms de temps de réponse représentent un aller-retour de messages passés à l'aller et au retour par le chemin sans latence ajoutée. De même, les points à 100 ms représentent un aller-retour de messages, l'un passé par le chemin à 100 ms et l'autre par le chemin sans latence ajoutée. Nous voyons aussi que lors des deux premières phases, mpmosh prend le meilleur chemin tandis que mosh garde son chemin d'origine, même en cas de rupture du chemin (phase 4).

Analyse phase par phase Les graphiques de la figure 5.9 représentent la répartition des temps de réponse des paquets pour chaque phase. Par exemple, lors de la phase 3 (figure *c*), 80% des envois ont un temps de réponse de moins de 400 ms pour la version optimisée de mpmosh (*mp*), de 900 ms pour mpmosh non optimisé (*mp-loss*) et 1300 s pour mosh (*std*). La figure 5.11 quant à elle représente le taux de pertes des messages envoyés. Lorsqu'un message est dupliqué, il suffit qu'un seul duplicata arrive pour que le message ne soit pas considéré comme perdu. Notons que les intervalles de temps sont légèrement restreints pour éviter de prendre dans les mesures de chaque phase des paquets qui appartiendraient à une autre phase.

Phase 1 (4 à 22 s) Ici, le meilleur chemin est le chemin 1 ; il n'est soumis à aucun retard ni perte de paquets. Les trois autres chemins ne subissent pas de perte de paquets, mais sont retardés de 50 ms (soit un RTT de 100 ms). À la fois mosh et mpmosh empruntent le chemin 1, tandis que les autres chemins sont simplement testés par les sondes de mpmosh.

La figure 5.9*a* montre que les trois programmes donnent lieu à des temps de réponse similaires, avec une



(a) Phase 1 — 4 *à* 22 *s* *(b) Phase 2* — 25 *à* 80 *s* *(c) Phase 3* — 85 *à* 140 *s* *(d) Phase 4* — 145 *à* 200 *s*

FIGURE 5.9 – Temps de réponse pour chaque envoi du client.

		Données seulement			Données et sondes			Proportion de sondes
		réactivité moyenne (ms)	données envoyées par le client (B)	nombre d'échantillons	réactivité moyenne (ms)	données envoyées par le client (B)	nombre d'échantillons	(% octets envoyés)
1	mp	(+2%) 79.61	(+4%) 29775	(+2%) 374	(-20%) 62.90	(+24%) 35475	(+54%) 564	16
	mp-loss	(+2%) 80.13	(+6%) 30288	(+3%) 378	(-19%) 63.42	(+26%) 35958	(+55%) 567	16
	std	(+0%) 78.25	(+0%) 28641	(+0%) 366	(+0%) 78.25	(+0%) 28641	(+0%) 366	0
2	mp	(-16%) 80.95	(+243%) 33916	(+311%) 419	(-28%) 69.90	(+278%) 37396	(+425%) 535	9
	mp-loss	(-16%) 81.61	(+243%) 33869	(+307%) 415	(-27%) 70.49	(+277%) 37289	(+419%) 529	9
	std	(+0%) 96.88	(+0%) 9882	(+0%) 102	(+0%) 96.88	(+0%) 9882	(+0%) 102	0
3	mp	(-10%) 83.98	(+1201%) 124535	(+1354%) 1483	(-13%) 81.94	(+1219%) 126275	(+1411%) 1541	1
	mp-loss	(-7%) 86.87	(+170%) 25801	(+191%) 297	(-23%) 71.81	(+203%) 29011	(+296%) 404	11
	std	(+0%) 93.83	(+0%) 9571	(+0%) 102	(+0%) 93.83	(+0%) 9571	(+0%) 102	0
4	mp	(-53%) 84.18	(+674%) 134016	(+1541%) 1592	(-54%) 82.34	(+684%) 135696	(+1599%) 1648	1
	mp-loss	(-50%) 89.18	(+6%) 18371	(+112%) 206	(-60%) 71.19	(+22%) 21071	(+205%) 296	13
	std	(+0%) 178.40	(+0%) 17308	(+0%) 97	(+0%) 178.40	(+0%) 17308	(+0%) 97	0

FIGURE 5.10 – Données de couche application envoyées par le client pour chaque phase.

médiane de 21 ms. Ils utilisent le chemin 1, le seul qui n'ait pas de latence ajoutée. En particulier, mpmosh n'emprunte pas un chemin de moins bonne latence que mosh.

La figure 5.10 nous permet de voir que le surcoût de *mp* et *mp-loss* est principalement lié aux sondes. Celles-ci représentent 16 % de la charge envoyée, avec 5700 octets en 25 secondes, soit 228 o/s octets par seconde (684 o/s à la couche lien, pour une trame ethernet en IPv6). Le surcoût des messages de données est négligeable, et s'explique par la taille de l'entête de couche réseau : celui de mpmosh est plus grand que celui de mosh de deux octets.

Phase 2 (40 à 65 s) Cette phase montre que mpmosh est capable de changer de chemin pour sélectionner celui de plus faible latence. Dans cette phase, les retards sont modifiés : le chemin 2 est sans retard, le chemin 1 a 400 ms de retard, et les chemins 3 et 4 ont respectivement 100 et 200 ms de retard.

La figure 5.9*b* montre que mosh reste sur le chemin 1, avec un temps de réponse d'environ 900 ms, alors que mpmosh bascule sur le chemin sans retard, et obtient donc les mêmes performances qu'à la phase 1.

La latence additionnelle du chemin 1 étant de 400 ms, nous nous serions attendus à avoir une latence de 800 ms pour *std*, soit 100 ms de moins. Or, comme nous l'avons observé en 5.6.1, les couches hautes peuvent introduire un certain délai, qui lui-même dépend de la latence mesurée par la couche réseau.

Le tableau 5.10 montre que lorsque mosh a une latence moins élevée, il envoie beaucoup moins de messages, mais plus volumineux. Ainsi, la taille moyenne des paquets est de 95 octets, alors qu'elle était de 78 octets à la première phase, et plus de trois fois moins de messages de données que mpmosh sont envoyés. Les sondes de *mp* et *mp-loss* ne représentent plus que 9 % de la charge envoyée.

La diminution significative des sondes est normale : rappelons que mpmosh envoie ses sondes à un intervalle proportionnel à la latence mesurée. Les chemins ayant de plus fortes latences dans cette expérience, il est normal que les sondes soient plus espacées, et représentent donc un coût moindre.

Phase 3 (70 à 95 s) Cette phase évalue le comportement de mpmosh en cas de pertes : les quatre chemins ont un taux de pertes inversement élevé par rapport à leur latence. En particulier, le chemin 1 reste avec un retard de 400 ms sans perte, et le chemin 4 reste sans retard avec 70 % de pertes.

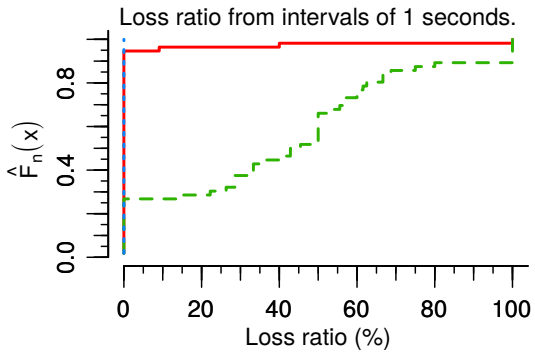
La figure 5.9*c* montre que *mp* et *mp-loss* ont les mêmes performances, avec un temps de réponse variant entre 0 et 700 ms. La figure 5.11 montre que *mp-loss* perd beaucoup de ses paquets, tandis que *mp* n'en perd aucun.

Avec une moyenne d'un peu plus d'un paquet sur deux de perdus, il serait logique que *mp-loss* ait de moins bons temps de réponse que *mp*. Cependant, parce que les couches hautes de l'application introduisent un délai, la perte d'un paquet est souvent compensée par un autre paquet arrivant. Nous observons dans nos relevés que le serveur peut mettre plus de 100 ms avant de voir apparaître une réponse.

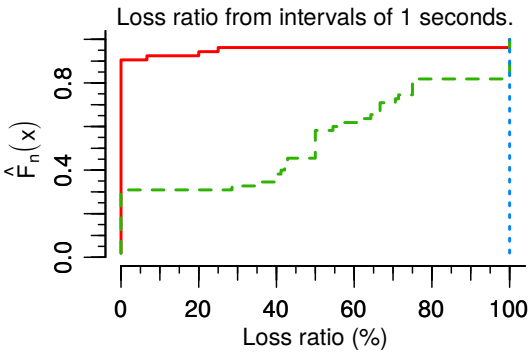
Le tableau 5.10 montre une très significative augmentation de la charge et une diminution des sondes pour mp : sa charge est quatre fois plus importante que celle de *mp-loss*. La quantité de sondes envoyées par *mp* diminue aussi à 1 %. Ces deux phénomènes s'expliquent simplement par la duplication des paquets faite par mp : les messages envoyés sont plus lourds que les sondes et sont envoyés à la place des sondes.

Phase 4 (100 à 125 s) Cette phase ne propose plus que des chemins sujets aux pertes : nous rajoutons un très fort taux de pertes (90%) au chemin 1. La version standard de mosh ne laisse alors passer que quelques paquets, tandis que mpmosh continue de fonctionner, dans ses deux formes.

Les résultats observés à partir de la figure 5.9*d*, du tableau 5.10 et de la figure 5.11 sont très similaires à la phase 3 pour ce qui est de mosh et mpmosh. Cela n'est pas surprenant, car nous constatons que le chemin 1, le seul à avoir été changé, n'est pratiquement jamais utilisé dans aucune des phases 3 et 4. Comme nous nous y attendions, *std* continue d'envoyer des paquets sur le chemin 1, et obtient donc des performances tragiques.



(c) Phase 3 — 70 *à* 95 *s*



(d) Phase 4 — 100 *à* 125 *s*

FIGURE 5.11 – Taux de pertes des paquets envoyés par le client.

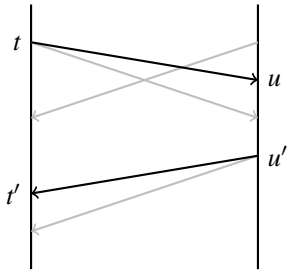


FIGURE 5.12 – Temps de réponse : $t' - t$.

Conclusion MPmosh offre une amélioration des performances et de la fiabilité : les meilleures latences sont obtenues, et des chemins alternatifs sont empruntés lorsqu'un chemin ne répond plus, que ce soit à cause de pertes systématiques ou occasionnelles de messages.

Notre optimisation pour limiter les taux de pertes est efficace, puisque presque aucun message n'est perdu. Bien qu'elle n'ait que peu d'impact sur le temps de réactivité de l'application, elle contribue pourtant à garder de petits paquets. L'impact sur le temps de réactivité peut s'expliquer par le fait que mosh peut retarder ses réponses ; il suffit alors qu'un seul paquet passe entre deux réponses pour avoir des performances similaires.

Enfin, il est intéressant de voir que l'application peut avoir un fin contrôle sur son trafic et sur les diverses performances qu'elle cherche à optimiser.

5.6.5 Détails sur les conditions de l'expérience

À chaque fois que mpmosh envoie ou reçoit un paquet, il l'écrit dans un *log*. Nous nous servons des *logs* du client et du serveur pour calculer les temps réponse et les pertes de paquets représentés sur les graphes. Le temps de réponse est calculé pour chaque paquet comme illustré sur la figure 5.12 : un message est envoyé au temps t avec potentiellement, dans le cas de mpmosh, des duplicata. Le temps d'arrivée u du paquet le plus rapide est considéré : les paquets perdus ou les duplicata en retard sont simplement ignorés. Enfin, le message de retour choisi est le premier partant après u, au temps u' ; de même, le plus rapide est sélectionné pour estimer le temps de réaction $t' - t$.

Pour être vraiment testés en parallèle et dans les mêmes conditions, ces trois programmes sont lancés simultanément sur le client, et reçoivent les mêmes entrées au même moment. L'expérience réalisée est le parcours d'une longue page de manuel (*bash*) : appuyer sur *espace* affiche la page suivante, tandis qu'appuyer sur *b* affiche la page précédente. Il y a donc peu d'informations envoyées, et il est important pour l'expérience utilisateur que les messages arrivent au plus vite pour que la bonne page s'affiche.

5.7 Ouverture

Ce travail s'ouvre sur deux suites naturelles : un travail de généralisation qui déboucherait sur la conception d'une bibliothèque, et l'évaluation d'une solution plus centralisée pour évaluer les performances des différents chemins, à la place ou en plus des programmes utilisant la bibliothèque.

Plus généralement, nous pouvons aussi penser à des cas où le multichemin, et peut être plus particulièrement la duplication de paquets, pourrait être bénéfique, comme la mobilité

Conception d'une bibliothèque Le module réseau de mpmosh est indépendant du reste du programme, et peut servir de base à une bibliothèque multichemin. Nos techniques sont indépendantes de mosh et il serait intéressant de voir dans quelle mesure elles peuvent être appliquées à d'autres applications.

En effet, mpmosh est une application interactive et légère : nous minimisons le RTT, et nous pouvons dupliquer les paquets sans problème. D'autres applications peuvent avoir des besoins différents : par exemple, une application légère mais non interactive pourrait préférer le chemin le moins sujet aux pertes et une application interactive mais plus lourde pourrait préférer éviter la duplication. On peut aussi penser à une application mixte qui pourrait vouloir utiliser certains chemins pour certaines données, et d'autres chemins pour d'autres données. Enfin, mpmosh est une application client-serveur : certains de ses traits seraient à modifier dans le contexte d'une application pair à pair.

Utilisation d'un auxiliaire L'idée d'utiliser un programme séparé pour mesurer les performances des différents chemins émane de certains utilisateurs qui utilisent plusieurs sessions mosh pour le même serveur. Bien que les sondes envoyées par chaque instance d'un mpmosh ne constituent certainement pas un surcoût important, il est raisonnable de vouloir centraliser ces mesures et éviter un trafic inutile.

Plusieurs choix de conception du programme auxiliaire sont possibles. Il peut être simplement passif, et partager les ressources entre les différents programmes s'exécutant sur un même nœud, voire seulement pour un seul utilisateur du nœud. Il peut aussi être, à l'opposé, le seul programme à communiquer avec le pair distant, via un programme parlant un même protocole : il s'agirait alors d'une sorte de nouvelle couche transport, ou de tunnel. Il peut aussi être entre les deux : à la fois échanger des données de performances, et effectuer des mesures, tout en laissant les applications communiquer directement.

Mobilité La figure 5.13 représente le cas d'un nœud mobile H initialement connecté à A. En se déplaçant, le nœud arrive à établir un lien avec B et ainsi un deuxième chemin vers le réseau. D'abord de mauvaise qualité, ce lien s'améliore en même temps que H se rapproche de B, et au contraire la liaison avec A se dégrade jusqu'à être perdue.

L'utilisation du multichemin aux couches supérieures peut permettre aux hôtes d'être mobiles, sans complexité particulière du côté du réseau. Il suffit que A et B fournissent chacun une adresse à H pour que celui-ci et les hôtes auxquels il est connecté puissent utiliser les deux chemins. Un certain nombre d'études montrent que MPTCP peut être utilisé pour assurer la mobilité des hôtes dans différentes situations [RNBH11, DLLG15].

Nous avons montré à travers notre travail que la duplication des paquets accélérait efficacement la reconvergence des applications en cas de panne, au point que la panne est presque inaperçue. Dans le cas d'un hôte mobile, nous pensons que la duplication peut apporter un gain significatif en réduisant ou en supprimant le temps de convergence et réduire le taux de pertes lorsque le nœud mobile n'a que des connexions hasardeuses. Une question ouverte serait de savoir dans quels cas un nœud pourrait prévoir qu'il risque de perdre un chemin, et dupliquer les paquets à l'avance — l'évolution de certaines performances du chemin pourrait être un indicateur.

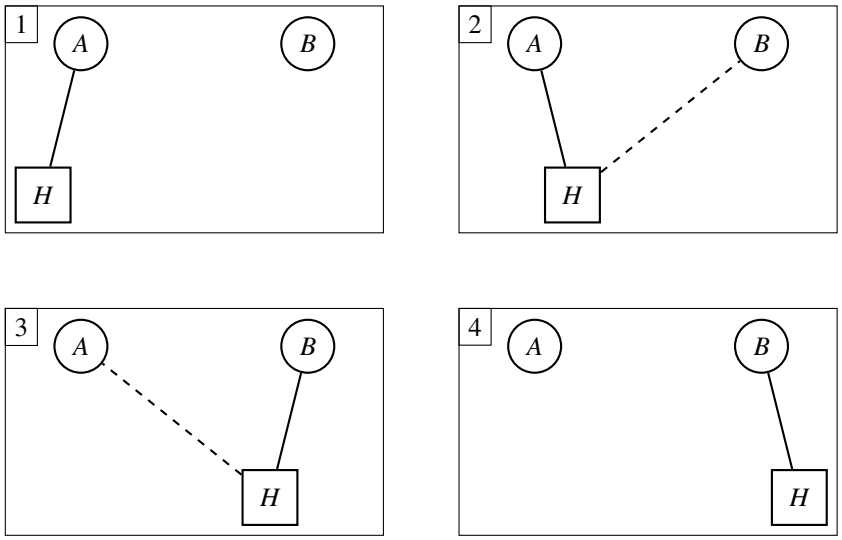


FIGURE 5.13 – Mobilité

Congestion Lorsque des paquets sont acheminés sur un chemin congestionné, ils peuvent être marqués du bit ECN (*Explicit Congestion Notification*). Dans notre travail, nous avons considéré que le trafic généré par nos applications légères était négligeable. En pratique, si de nombreux hôtes font tourner de nombreuses applications similaires sur un lien déjà congestionné, le trafic pourrait être impacté. L'utilisation du bit ECN pourrait faire évoluer la stratégie de duplication des paquets pour éviter autant que possible les chemins congestionnés.

Chapitre 6

Conclusion

Dans cette thèse, nous nous sommes intéressés au routage sensible à la source et à son impact sur les couches supérieures. Le routage sensible à la source est une extension compatible au paradigme de routage actuellement utilisé dans l'Internet : le routage *next-hop*. En routage *next-hop*, les paquets sont routés en fonction de leur adresse destination et de la FIB des routeurs. En routage sensible à la source, les paquets sont aussi routés en fonction de leur adresse source.

Routage de réseaux multihomés Le routage sensible à la source résout le problème du routage dans certains types de réseaux *multihomés*. Un réseau est *multihomé* lorsqu'il est connecté à plusieurs fournisseurs d'accès, ce qui peut être utile pour assurer la fiabilité du réseau ou l'augmentation de ses performances.

La solution classique pour connecter un réseau à plusieurs fournisseurs d'accès est d'acquérir au préalable un préfixe d'adresses indépendant des fournisseurs. Le réseau annonce son préfixe à chacun de ses fournisseurs, qui réannoncent ce préfixe à l'Internet et fournissent au réseau une route pour l'Internet. La nature dynamique des protocoles de routage suffit à assurer la fiabilité du réseau en faisant transiter les paquets par l'un ou l'autre des fournisseurs.

Cependant, tous les réseaux ne peuvent acquérir de préfixes indépendants des fournisseurs d'accès, comme par exemple les réseaux de petites entreprises ou les réseaux domestiques. Dans ce cas, chaque fournisseur d'accès fournit au réseau un préfixe d'adresses et une route par défaut. On parle de *multihoming* avec plusieurs adresses.

Or, le fournisseur d'un préfixe d'adresses filtre les paquets en provenance de son client qui n'ont pas leur adresse source dans le préfixe qu'il a fourni. Il est donc impératif que les paquets du réseau à destination de l'Internet passent par le FAI fournisseur de leur adresse source : le routage du réseau doit être capable de tenir compte de l'adresse source des paquets.

Le routage *next-hop* n'est pas assez expressif pour pouvoir différencier les paquets en fonction de leur adresse source. Toutefois, plusieurs techniques existent à base de trafic d'ingénierie et de tunnels pour pallier ce problème. Ces techniques introduisent de l'état dans le réseau et ajoutent de la complexité à la configuration du réseau.

La solution que nous avons proposée dans ce manuscrit est de remplacer le routage *next-hop* par le routage sensible à la source en étendant les protocoles de routage existants. Nous avons montré que cette solution avait plusieurs avantages :

1. Un seul protocole de routage peut être déployé de manière uniforme sur le réseau.
2. Les paquets sont directement acheminés au bon fournisseur, sans encapsulation, par le plus court chemin calculé par le protocole de routage.

3. Le filtrage des paquets par les FAI est totalement évité. En fait, les paquets qui pourraient se faire filtrer ne sont même pas routés puisque les routes fournies par les FAI sont spécifiques au préfixe qu'ils ont fourni. Si un hôte émet un paquet qui devrait être filtré par un FAI, ce paquet est donc détruit au premier routeur du réseau.
4. Le routage sensible à la source est une extension compatible au routage *next-hop* :
   — Un réseau classique avec un seul fournisseur d'accès peut passer au routage sensible à la source sans qu'aucune modification ne soit nécessaire. Toutefois, même un réseau classique gagnerait à ce que la route fournie par son FAI soit sensible à la source pour éviter le transit des paquets qui se feraient filtrer.
   — Le déploiement du routage sensible à la source peut être progressif. La cohabitation de routeurs *next-hop* et sensible à la source peut exister. Pour assurer le routage d'un réseau *multihomé*, il suffit que les routeurs de frontière forment une épine dorsale de routeurs sensibles à la source, et qu'ils annoncent une route par défaut avec un préfixe source de longueur nulle.
5. Le routage sensible à la source fournit plusieurs routes aux couches supérieures des hôtes qui peuvent optimiser leurs connexions par des protocoles multichemin. Nous reviendrons sur ce point quand nous parlerons des couches supérieures, mais notons déjà que ce choix peut être fait par les API standard et qu'il est déjà exploité par plusieurs protocoles de transport, comme MPTCP.

Réalisation du routage sensible à la source Nous avons conçu et réalisé la première implémentation complète du routage sensible à la source. Cela a nécessité de définir des mécanismes d'extension généraux qui garantissent l'interopérabilité, de définir l'extension à un protocole de routage existant et surtout de définir des algorithmes qui permettent aux tables de transfert de transférer correctement les paquets.

Levée d'ambiguïté En routage *next-hop*, les tables de transfert associent un *next-hop* à un préfixe d'adresses destination. Ces tables peuvent avoir plusieurs entrées routant un même paquet : celle qui a le préfixe le plus long (*longest match*) est sélectionnée pour transférer le paquet. Ce choix coïncide avec le choix du préfixe le plus spécifique. Plus formellement, nous pouvons donc dire que les tables de transfert *next-hop* sont munies d'une relation d'ordre sur les entrées routant un même paquet : la spécificité des préfixes.

En routage sensible à la source, les tables de transfert associent un *next-hop* à une paire d'adresses destination et source. Nous avons montré que là encore les tables de routage doivent être munies d'une relation ordonnant toutes les entrées routant un même paquet, et que cette relation doit être homogène à tout le réseau sans quoi des boucles de routage persistantes peuvent s'installer. Or, la spécificité des préfixes ne suffit plus à lever les ambiguïtés. Il y a consensus au sein de la communauté pour router les paquets en préférant les entrées spécifiques à la source ayant le préfixe destination le plus spécifique, et à défaut le préfixe source le plus spécifique.

Or, bien que la plupart des tables de transfert permettent de faire du routage sensible à la source, nous avons trouvé que dans certains cas, contrairement à ce que nous voulons, ce sont les entrées ayant le préfixe source le plus spécifique qui sont choisies. C'est le cas lorsqu'il faut implémenter le routage sensible à la source avec des règles d'ingénierie de trafic qui associent des tables de transfert *next-hop* à des préfixes source.

Pour résoudre ce problème, nous avons d'abord trouvé que toute ambiguïté entre deux routes pouvait être levée en ajoutant une troisième route plus spécifique que les deux autres. En effet, tous les ordres que nous connaissons sont compatibles avec la spécificité. La table de transfert obtenue est alors sans ambiguïté, et l'ordre supporté n'a plus d'importance.

Nous avons formalisé cette observation en définissant une notion de complétude de table de routage. Une table est fortement complète si, pour toute paire d'entrée de la table, il existe une entrée dans la table

qui route tous les paquets routés par les deux entrées. Nous avons montré qu'une table complète n'était pas ambiguë.

Enfin, nous avons mis au point un algorithme de levée d'ambiguïté, conçu pour être implémenté par les protocoles de routage. Cet algorithme :

— maintient une FIB complète, et garde donc sa non ambiguïté ;
— est incrémental, comme les primitives classiques de manipulation des FIB du noyau ;
— est sans état supplémentaire, au sens où il ne retient pas les entrées additionnelles installées dans la FIB ;
— est général, car applicable à d'autres comportements et basé sur de faibles hypothèses ;
— permet à un protocole de routage d'être utilisé sur tout système, même si l'ordre des tables de transfert n'est pas celui souhaité par le protocole : notre algorithme force l'ordre à utiliser.

Protocoles de routage sensible à la source Nous nous sommes intéressés aux protocoles à vecteur de distances pour la réalisation complète d'un protocole de routage sensible à la source dynamique. En routage *next-hop*, ces protocoles annoncent les routes installées dans leur table de transfert à leurs voisins. À la réception d'une annonce pour un préfixe destination, un nœud sait qu'il peut transférer les paquets à destination de ce préfixe au *next-hop* correspondant à l'annonce.

En routage sensible à la source, nous avons vu que les protocoles à vecteur de distances peuvent garder les mêmes algorithmes et mécanismes fondamentaux mais doivent s'adapter en ajoutant un préfixe source à leurs structures de données. En particulier, les protocoles à vecteur de distance sensibles à la source annoncent des paires de préfixes destination et source. À la réception d'une annonce, les protocoles de routage sensibles à la source réagissent de manière analogue aux protocoles *next-hop*.

Toutefois, *l'extension* d'un protocole à vecteur de distances au routage sensible à la source demande des précautions particulières pour assurer la compatibilité avec sa version de base. La compatibilité est assurée :

— en utilisant le protocole de base pour les routes spécifiques au préfixe source de longueur nulle. En effet, toutes les adresses sont dans le préfixe de longueur nulle ; une route avec un préfixe source de longueur nulle est donc équivalente à la même route sans préfixe source. Donc :
  — une route de préfixe source de longueur nulle est annoncée sans préfixe source ;
  — une annonce classique (sans préfixe source) est reçue comme si elle avait un préfixe source de longueur nulle.
— si le protocole de base ignore totalement les annonces qui contiennent un préfixe source. Le protocole de base ne peut pas juste ignorer le préfixe source et garder le préfixe destination, sans quoi des boucles de routage persistantes peuvent s'installer.

À partir de ces méthodes, nous avons défini l'extension sensible à la source du protocole de routage à vecteur de distances Babel. Nous avons implémenté cette extension, et elle est déployée en production.

Multichemin aux couches supérieures En routage *next-hop*, l'adresse destination d'un paquet est utilisée à chaque saut pour déterminer son prochain saut. Le chemin du paquet dans le réseau est donc lié à son adresse destination, et deux paquets allant d'un hôte donné à un autre avec des adresses destination différentes peuvent suivre des chemins très différents. Lorsque l'hôte de destination a plusieurs adresses différentes, l'hôte émetteur, par le choix de l'adresse destination, choisit un chemin dans le réseau.

Le routage sensible à la source complète cette flexibilité offerte aux hôtes avec le choix de l'adresse source. À chaque saut, l'adresse source d'un paquet est utilisée pour déterminer son prochain saut. Le chemin du paquet dans le réseau est donc aussi lié à son adresse source. Dans le cas où l'hôte émetteur a deux adresses IPv6, le choix de son adresse source pour une même destination coïncide avec le choix d'un chemin dans le réseau.

Cette flexibilité offerte aux hôtes prend tout son sens dans le cas des réseaux *multihomés* avec plusieurs adresses. Dans ces réseaux, un hôte a une adresse par fournisseur d'accès. L'adresse source d'un paquet émis par un hôte *multihomé* détermine le fournisseur d'accès par lequel le paquet doit sortir du réseau. Inversement, l'adresse destination d'un paquet destiné à un hôte *multihomé* détermine le FAI par lequel le paquet entre dans le réseau. Donc, le choix d'une paire d'adresses source et destination coïncide avec le choix d'un chemin dans le réseau.

Routage sensible à la source et couches supérieures Le routage sensible à la source suffit à résoudre le problème du routage des réseaux *multihomés* avec plusieurs adresses, et il fournit la fiabilité aux hôtes au sens où il y a toujours une route disponible pour l'Internet si l'un des FAI tombe en panne. Mais ce n'est pas le rôle du routage que d'assurer la fiabilité des connexions des hôtes : ce rôle leur revient.

Nous avons d'ailleurs montré un problème analogue qui ne nécessitait aucun routage : celui de l'hôte *multihomé*, directement connecté à plusieurs fournisseurs d'accès. Puisque l'hôte est directement connecté aux FAI, aucun routage n'est à faire, mais il doit déjà disposer de protocoles multichemin pour assurer la fiabilité de ses connexions.

Or, le routage sensible à la source s'est révélé être une brique manquante pour les protocoles multichemin de couches supérieures. Dans le cas de l'hôte *multihomé* comme dans le cas du réseau *multihomé* avec routage sensible à la source, les mêmes protocoles peuvent être utilisés pour assurer la fiabilité des connexions ou l'augmentation de leurs performances.

Parmi les protocoles que nous avons évoqué, nous avons notamment étudié MPTCP, une extension compatible de TCP pour le multichemin. MPTCP considère que le choix des adresses source et destination d'un paquet coïncide avec le choix d'un chemin. Nous avons mis en évidence que MPTCP est capable d'assurer la fiabilité des connexions TCP des applications et d'augmenter leur débit en répartissant la charge sur les différents chemins proposés par le routage sensible à la source.

La plupart des applications utilisent TCP et peuvent donc automatiquement bénéficier des avantages de MPTCP. Avec le routage sensible à la source le problème du *multihoming* avec plusieurs adresses a donc désormais une solution complète pour ces applications.

Multichemin à la couche application Plusieurs applications définissent leurs propres mécanismes de couche transport au dessus de UDP. Nous voyons à travers ces applications que la connaissance d'une application a un impact important sur la conception d'un transport optimisé pour cette application.

Nous avons montré qu'il en va de même pour les protocoles multichemin. En particulier, MPTCP ne peut remplacer un transport conçu pour une application spécifique car il lui arrive de n'utiliser qu'un seul chemin. Il se comporte alors comme TCP, qui est sous optimal pour cette application.

En prenant le contre-pied de MPTCP, qui cherche à optimiser le débit, nous nous sommes intéressés aux applications légères, pour lesquelles il est inutile d'augmenter le débit, et interactives, pour lesquelles nous cherchons à optimiser la latence. Là encore, nous avons vu que la connaissance de l'application nous permettait d'envisager des techniques d'optimisation particulières.

Nous avons défini des mécanismes pour sonder et évaluer les différents chemins, et choisir le chemin de meilleur latence. Nous avons développé deux techniques d'optimisation basées sur la duplication de paquets :

1. la première accélère la reconvergence après une panne : dès lors qu'un chemin est suspecté défaillant, un duplicata est envoyé sur le premier chemin valide, jusqu'à ce que la panne soit confirmée ou infirmée ;
2. la deuxième limite le taux de pertes des paquets : si le chemin de meilleur latence est sujet aux pertes, alors les données sont dupliquées sur le deuxième meilleur chemin, et ainsi de suite pour

chaque chemin.

Nous avons adapté ces mécanismes à une application particulière, mosh, ce qui nous a permis de les tester et de vérifier leur pertinence. En particulier :

1. l'accélération de la convergence améliore sensiblement l'expérience de l'utilisateur pour un surcoût occasionnel très limité ;
2. la limitation du taux de pertes est efficace et, dans le cas de mpmosh, contribue à la fluidité de l'application et à ce que chaque paquet soit de plus petite taille — la quantité d'information réémise est moindre.

Ouverture Le routage sensible à la source est aujourd'hui la solution retenue pour les réseaux domestiques *multihomés* par le groupe de travail *Homenet* de l'IETF. Ce même groupe de travail a aussi retenu le protocole de routage Babel, avec notre extension, comme le protocole de routage sur lequel travailler, tandis que plusieurs autres protocoles de routage ont aussi été étendus au routage sensible à la source. Nous avons bon espoir que ce paradigme se généralise.

En matière de routage, d'autres pistes peuvent être explorées. Nous pensons naturellement que d'autres champs du paquet IP pourraient être utilisés, notamment le ToS [CC17]. Le protocole de routage pourrait aussi router différemment les paquets en fonction des interfaces par lesquelles ils arrivent.

Nous avons vu que la conception d'un transport pouvait fortement être influencée par les besoins de l'application. La réalisation de bibliothèques multichemin requiert de comprendre le besoin des applications. Nous avons apporté une solution très différente de celle de MPTCP et que nous avons appliqué à mosh. Bien que se voulant généralisable à toute application légère et interactive, il serait intéressant de l'appliquer à d'autres applications.

Enfin, nous pensons que l'utilisation du routage sensible à la source combiné aux protocoles multichemin des couches supérieures pourrait être une solution élégante à la mobilité. Nous avons vu qu'un certain nombre d'études existent déjà dans la littérature du côté des hôtes mobiles. Le routage sensible à la source pourrait généraliser la mobilité aux réseaux tout entiers : par exemple, un train pourrait être un réseau mobile où chaque hôte se voit attribuer différentes adresses IP tout au long du trajet par l'intermédiaire des routeurs. Le routage sensible à la source acheminerait les paquets vers les différents relais tandis que les couches supérieures des hôtes du réseau assureraient la fiabilité des connexions.

Bibliographie

[84715] The 8472. Multiple-address operation for the BitTorrent DHT. BEP 45 (Informational Draft), June 2015.

[ASNN07] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095 (Proposed Standard), December 2007.

[BC13] Matthieu Boutier and Juliusz Chroboczek. Source-specific Routing. Internet-Draft draft-boutier-homenet-source-specific-routing-00, IETF Secretariat, July 2013.

[BC14] Matthieu Boutier and Juliusz Chroboczek. Source-specific Routing in Babel. Internet-Draft draft-boutier-babel-source-specific-01, IETF Secretariat, Nov 2014.

[BC15] Matthieu Boutier and Juliusz Chroboczek. Source-specific routing. In *Proceedings of the 14th IFIP Networking Conference, Networking 2015, Toulouse, France, 20-22 May, 2015*, pages 1–9, May 2015.

[BC18] Matthieu Boutier and Juliusz Chroboczek. Source-Specific Routing in Babel. Internet-Draft draft-ietf-babel-source-specific-03, Internet Engineering Task Force, January 2018. Work in Progress.

[BG92] Dimitri P. Bertsekas and Robert G. Gallager. *Data Networks, Second Edition*. Prentice Hall, 1992.

[BGmRA04] Marcelo Bagnulo, Alberto García-martínez, Juan Rodríguez, and Arturo Azcorra. The case for source address dependent routing in multihoming. In *Quality of Service in the Emerging Networking Panorama : 5th International Workshop on Quality of Future Internet Services, QofIS 2004 and First Workshop on Quality of Service Routing WQoSR 2004 and 4th International Workshop on Internet Charging and QoS Technology, ICQT 2004, Barcelona, Catalonia, Spain, September 29 - October 1, 2004, Proceedings*, pages 237–246. Springer, 2004.

[BL18] Fred Baker and David Lamparter. IPv6 Source/Destination Routing using IS-IS. Internet-Draft draft-ietf-isis-ipv6-dst-src-routing-00, Internet Engineering Task Force, January 2018. Work in Progress.

[BS04] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 (Best Current Practice), March 2004.

[CC17] Gwendoline Chouasne and Juliusz Chroboczek. TOS-Specific Routing in Babel. Internet-Draft draft-chouasne-babel-tos-specific-00, Internet Engineering Task Force, July 2017. Work in Progress.

[CDJH14] T. Clausen, C. Dearlove, P. Jacquet, and U. Herberg. The Optimized Link State Routing Protocol Version 2. RFC 7181 (Proposed Standard), April 2014. Updated by RFCs 7183, 7187, 7188, 7466.

[Chr11] J. Chroboczek. The Babel Routing Protocol. RFC 6126 (Experimental), April 2011.

[Chr14] Juliusz Chroboczek. Diversity Routing for the Babel Routing Protocol. Internet-Draft draft-chroboczek-babel-diversity-routing-00, IETF Secretariat, July 2014.

[Chr15] J. Chroboczek. Extension Mechanism for the Babel Routing Protocol. RFC 7557 (Experimental), May 2015.

[Chr16] Juliusz Chroboczek. Deployment experiences with HNCP. IETF talk, July 2016.

[CS17] Juliusz Chroboczek and David Schinazi. The Babel Routing Protocol. Internet-Draft draft-ietf-babel-rfc6126bis-04, Internet Engineering Task Force, October 2017. Work in Progress.

[DCABM05] Douglas S. J. De Couto, Daniel Aguayo, John C. Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. *Wireless Networks*, 11(4) :419–434, 2005.

[DCB17] Quentin De Coninck and Olivier Bonaventure. Multipath quic : Design and evaluation. In *13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017). http ://multipath-quic. org*, 2017.

[DG13] Seweryn Dynerowicz and Timothy G. Griffin. On the forwarding paths produced by internet routing algorithms. In *21th IEEE International Conference on Network Protocols*, Goettingen, Germany, October 2013.

[dLB06] Cédric de Launois and Marcelo Bagnulo. The paths toward IPv6 multihoming. *IEEE Communications Surveys and Tutorials*, 8(1-4) :38–51, 2006.

[dLBL03] Cédric de Launois, Olivier Bonaventure, and Marc Lobelle. The NAROS Approach for IPv6 Multihoming with Traffic Engineering. In *Quality for All, 4th COST 263 International Workshop on Quality of Future, Internet Services, QofIS 2003, Stockholm, Sweden, October 1-2, 2003, Proceedings*, pages 112–121. Springer, 2003.

[DLLG15] Pengyuan Du, Xiao Li, You Lu, and Mario Gerla. Multipath TCP over LEO Satellite Networks. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2015 International*, pages 1–6. IEEE, 2015.

[DPZ04] Richard Draves, Jitendra Padhye, and Brian Zill. Routing in multi-radio, multi-hop wireless mesh networks. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking, MOBICOM 2004, 2004, Philadelphia, PA, USA, September 26 - October 1, 2004*, pages 114–128, 2004.

[Dra99] Richard Draves. Simple Source Address Selection for IPv6. Internet-Draft draft-draves-ipngwg-simple-srcaddr-00.txt, IETF Secretariat, April 1999.

[Dra03] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484 (Proposed Standard), February 2003. Obsoleted by RFC 6724.

[FRHB09] Alan Ford, Costin Raiciu, Mark Handley, and Sébastien Barré. TCP Extensions for Multipath Operation with Multiple Addresses. Internet-Draft draft-ford-mptcp-multiaddressed-00, IETF Secretariat, May 2009.

[FS98] P. Ferguson and D. Senie. Network Ingress Filtering : Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (Informational), January 1998. Obsoleted by RFC 2827.

[FS00] P. Ferguson and D. Senie. Network Ingress Filtering : Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[GLA93] Jose J Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking (TON)*, 1(1) :130–141, 1993.

[GMBVB10] Alberto García-Martínez, Marcelo Bagnulo, and Iljitsch Van Beijnum. The Shim6 architecture for IPv6 multihoming. *IEEE Communications Magazine*, 48(9) :152–157, 2010.

[HK03] Christian Huitema and David Kessens. Simple Dual Homing Experiment. Internet-Draft draft-huitema-multi6-experiment-00, IETF Secretariat, June 2003.

[HM04] C Huitema and M Marcelo. Ingress filtering compatibility for IPv6 multihomed sites. Internet-Draft draft-huitema-multi6-ingress-filtering-00, IETF Secretariat, October 2004.

[HSP00] Adiseshu Hari, Subhash Suri, and Guru M. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, 19th Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, pages 1203–1212, 2000.

[Hui95] Christian Huitema. Multi Homed TCP. Internet-Draft draft-huitema-multi-homed-01, IETF Secretariat, June 1995.

[JBC14] Baptiste Jonglez, Matthieu Boutier, and Juliusz Chroboczek. A delay-based routing metric. *unpublished*, March 2014.

[JC15] Baptiste Jonglez and Juliusz Chroboczek. Delay-based Metric Extension for the Babel Routing Protocol. Internet-Draft draft-jonglez-babel-rtt-extension-01, Internet Engineering Task Force, May 2015. Work in Progress.

[JYY+13] Yong Jin, Takuya Yamaguchi, Nariyoshi Yamai, Kiyohiko Okayama, and Motonori Nakamura. A Site-Exit Router Selection Method Using Routing Header in IPv6 Site Multihoming. *Information and Media Technologies*, 8(3) :757–765, 2013.

[Lam15] David Lamparter. Homenet IS-IS Profile. Internet-Draft draft-lamparter-homenet-isis-profile-00, IETF Secretariat, July 2015.

[LS05] Haibin Lu and Sartaj Sahni. Conflict detection and resolution in two-dimensional prefix router tables. *IEEE/ACM Trans. Netw.*, 13(6) :1353–1363, December 2005.

[Mil83] D.L. Mills. DCN Local-Network Protocols. RFC 891 (Internet Standard), December 1983.

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4 : Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[NB09] E. Nordmark and M. Bagnulo. Shim6 : Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[NC11] Habib Naderi and Brian E Carpenter. A review of IPv6 multihoming solutions. In *10th International Conference on Networks (ICN 2011)*, pages 145–150, 2011.

[NCC11] RIPE NCC. IPv6 Address Allocation and Assignment Policy. ripe-466, April 2011.

[OO11] Kenji Ohira and Yasuo Okabe. Host-Centric Site-Exit Router Selection in IPv6 Site Multihoming Environment. In *25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2011, Biopolis, Singapore, March 22-25, 2011*, pages 696–703. IEEE, 2011.

[RA04] Kultida Rojviboonchai and Hitoshi Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE transactions on communications*, 87(9) :2699–2707, 2004.

[RNBH11] Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, and Mark James Handley. Opportunistic Mobility with Multipath TCP. In *Proceedings of the Sixth International Workshop on MobiArch*, MobiArch '11, pages 7–12, New York, NY, USA, 2011. ACM.

[RPB+12] Costin Raiciu, Christoph Paasch, Sébastien Barré, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be ? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 399–412, 2012.

[SBP16] M. Stenberg, S. Barth, and P. Pfister. Home Networking Control Protocol. RFC 7788 (Proposed Standard), April 2016.

[SNM+16] D. Savage, J. Ng, S. Moore, D. Slice, P. Paluch, and R. White. Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP). RFC 7868 (Informational), May 2016.

[SP17] David Schinazi and Tommy Pauly. Happy Eyeballs Version 2 : Better Connectivity Using Concurrency. RFC 8305, December 2017.

[SRC80] Jerome H Saltzer, David P Reed, and David D Clark. Source Routing for Campus-wide Internet Transport. In *Proc. IFIP WG 6.4 Int'l Workshop on Local Networks*, pages 1–23, 1980.

[Ste07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335.

[SX99] Randall R. Stewart and Qiaobing Xie. Multi-network Datagram Transmission Protocol. Internet-Draft draft-ietf-sigtran-mdtp-00, IETF Secretariat, February 1999.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[TC13] Ole Troan and Lorenzo Colitti. IPv6 Multihoming with Source Address Dependent Routing (SADR). Internet-Draft draft-troan-homenet-sadr-01, IETF Secretariat, sep 2013.

[TDMC12] D. Thaler, R. Draves, A. Matsumoto, and T. Chown. Default Address Selection for Internet Protocol Version 6 (IPv6). RFC 6724 (Proposed Standard), September 2012.

[TI17] Martin Thomson and Janardhan Iyengar. QUIC : A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-02, Internet Engineering Task Force, March 2017. Work in Progress.

[WB12] Keith Winstein and Hari Balakrishnan. Mosh : An Interactive Remote Shell for Mobile Clients. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 177–182, 2012.

[WY12] D. Wing and A. Yourtchenko. Happy Eyeballs : Success with Dual-Stack Hosts. RFC 6555 (Proposed Standard), April 2012.

[YWK05] Yaling Yang, Jun Wang, and Robin Kravets. Interference-aware load balancing for multihop wireless networks. Technical report, University of Illinois at Urbana-Champaign, 2005.