Cursus Master Ingénierie
Informatique
systèmes
et réseaux

UFR de mathématique et d'informatique
Université de Strasbourg

Université de Strasbourg

iCUBE

Rapport de stage

Florent HARDY

Fusion entre Segment Routing et Flowlet Switching : une évaluation de performances

Stage dans le laboratoire iCube, au sein de l'équipe Réseaux

du 3 Juin 2024 au 31 Juillet 2024

Tuteur de stage
Pascal MÉRINDOL
Enseignant Chercheur
Laboratoire iCube

Enseignant référent
Pierre DAVID
Enseignant Chercheur
Université de Strasbourg

Table des matières

1 Remerciements

Je tiens à remercier Pascal Mérindol mon maître de stage, et Pierre David, mon encadrant, de m'avoir fait confiance et de m'avoir procuré ce stage.

Mes remerciements les plus sincères vont à Quentin Bramas, Jean-Romain Luttringer et Pascal Merindol, les auteurs de l'article GOFOR qui est à la source de mon TER et un cousin distant du sujet de ce stage, de m'avoir apporté plusieurs fois leur aide et conseils.

Merci également à toute l'équipe Réseaux qui a su faire du laboratoire un lieu accueillant et chaleureux.

2 Introduction

Les réseaux modernes n'ont plus à faire leurs preuves. La variété de services est si grande qu'il faudrait des années pour les énumérer. Parmi ces services, certains sont gourmands en bande passante, typiquement le transfert de fichiers, pendant que d'autres nécessitent des faibles latences (les appels téléphoniques par exemple). Cette diversité implique que les réseaux se doivent de supporter des charges de plus en plus lourdes et de plus en plus rapides.

Si l'on se place dans la tête d'un·e opérateur·rice réseau d'un fournisseur d'accès internet, la première chose à faire est de réduire les congestions. Cela peut être fait en achetant de l'équipement de plus en plus cher, ou en utilisant l'équipement à disposition de manière plus efficace. C'est principalement la seconde option que nous allons explorer.

Ce sont deux notions : *Segment Routing* et *Flowlet Switching* qui nous intéressent pour ce stage. Après avoir décrit l'environnement de travail, ce sont les deux technologies d'ingénierie de trafic qui seront détaillées dans une partie Contexte. Ensuite, dans une section Missions, je détaillerai le plan d'approche d'une fusion entre ces deux technologies et les choix qui ont dû être faits. Finalement, les résultats de simulations seront exposés suivis d'une étude dans une section Travail effectué. Puis une réflexion sur le travail effectué et une conclusion feront leur apparition.

3 Environnement de travail

Le laboratoire iCube a été créé en 2013 sous l'égide du CNRS, de l'Université de Strasbourg, de l'ENGEES et de l'INSA, il compte 750 membres dont font partie des maître·sse·s de conférence, des professeur·e·s des universités et des ingénieur·e·s de recherche entre autres. Mon tuteur de stage, Pascal Mérindol, est enseignant-chercheur pour l'équipe *Réseaux* du Département Informatique Recherche, son travail se concentre principalement sur les réseaux filaires. Fabrice Theoleyre, co-responsable avec Julien Montavont, du département quant à lui est directeur de recherche au CNRS. Jean-Romain Luttringer et Quentin Bramas sont tous deux membres permanents qui ont pris de leur temps pour m'aider durant mon travail.

Le laboratoire se situe sur le campus d'Illkirch-Graffenstaden, au sein du bâtiment du Pôle API. La construction du Pôle API s'est achevée en 1993 et il héberge deux écoles d'ingénieur·e·s, plusieurs laboratoires et des plateformes de transfert de technologies dont une partie du FIT IOT-Lab.

3.1 Présentation de l'équipe

L'équipe de recherche Réseaux d'iCube est composée de 11 membres permanents, deux post-doctorant·e·s et 5 doctorant·e·s. Comme le nom l'indique, l'équipe focalise ses recherches sur la conception d'algorithmes et protocoles de routage et de communication. Les principaux sujets sont :

- les réseaux de bordure : l'étude des connexions entre objets intelligents et leur environnement en termes de fiabilité, consommation d'énergie ou de durée de vie. Julien Montavont et Samir Si-Mohammed, par exemple, travaillent sur l'Internet des Objets [1][2] ;
- les réseaux de cœur : l'étude de l'acheminement des paquets dans les réseaux informatiques permettant les communications entre machines. Quentin Bramas, Jean-Romain Luttringer et Pascal Mérindol ont par exemple rédigé l'article Gofor[3] qui touche au *Segment Routing*, un outil d'ingénierie de trafic utilisé dans les réseaux d'opérateurs ;
- les algorithmes distribués : l'étude des systèmes distribués, leur complexité, contraintes et applications. Quentin Bramas et Anissa Lamani, en particulier, travaillent sur des problèmes portant sur des esseims de robots [4].

3.2 Conditions de travail

Durant les 2 mois de stage, j'ai effectué la plupart de mon travail aux bureaux du laboratoire, dans une salle dédiée aux stagiaires qui travaillaient sur des sujets légèrement similaires. En somme, nous travaillions sur le même *framework*. Cela nous permettait d'échanger à propos de nos sujets respectifs et de garder un peu de convivialité pendant nos heures de travail.

Afin de nous mettre à jour sur l'avancement du stage, Pascal Mérindol et moi faisions des réunion dès lors que le projet avançait ou subissait des changements de direction. Certaines de ces réunions étaient accompagnées de Jean-Romain Luttringer et Quentin Bramas car ils sont co-auteur·e·s d'un article dont j'utilisais le résultat. Certaines réunions se faisaient seulement avec Jean-Romain Luttringer car il était co-responsable de mon Travail d'Étude et de Recherche (TER). TER qui était un précurseur au sujet de stage étant donné qu'il introduit des solutions d'implémentation qui sont évaluées dans ce stage.

4 Contexte

Les réseaux modernes doivent être polyvalents pour supporter la gamme de services qui tournent sur lesdits réseaux. Les utilisateur·rice·s s'attendent à un certain niveau de performances, ce qui implique de surmonter les congestions et les pannes.

4.1 Équilibrage de charge classique / Équilibrage de charge à la source

Ce stage s'inscrit comme la continuation de mon TER. Le sujet de mon TER portait sur l'équilibrage de charge à la source. Pour expliquer ce qu'est l'équilibrage de charge à la source, il faut d'abord introduire l'équilibrage de charge "classique".

L'équilibrage de charge "classique", type ECMP, envoie sur différentes interfaces les paquets de différents flux dans le but d'éviter les congestions. En effet, si plusieurs chemins sont accessibles pour lier une source à une destination, il peut être préférable d'envoyer les paquets d'un flux sur l'un des chemins et les paquets de l'autre flux sur l'autre chemin.

ECMP ne peut distribuer la charge que sur les plus courts chemins. Ces plus courts chemins sont décidés par l'algorithme de routage (IS-IS ou OSPF par exemple) en assignant des poids aux liens et, par extension, aux chemins formés par ces liens.

Afin de distribuer la charge à l'aide d'ECMP, les routeurs calculent un *hash* à partir d'un quintuplet qui discerne les flux. Ce quintuplet est formé de l'adresse IP source, l'adresse IP de destination, le port source, le port destination et le protocole/service utilisé. Ce hash est utilisé pour choisir l'interface du routeur sur laquelle le paquet sera envoyé.

Le choix de l'interface est fait en fonction du flux auquel appartient le paquet et le nombre d'interfaces différentes capables d'envoyer le paquet sur un plus court chemin. Ces trois notions : la discrimination des paquets via leur appartenance à un flux, la notion de plus court chemin et le choix de l'interface de sortie sont les éléments qui ont été modifiés afin de rendre l'équilibrage de charge moins "classique".

4.1.1 Discrimination par flux / Discrimination par *flowlet*

Plutôt que de discriminer les paquets en fonction du flux auquel ils appartiennent, Sinha et al.[5] puis Vanini et al.[6] proposent de différencier les paquets en fonction de la *flowlet* auquel ils appartiennent. Cette notion s'appelle le *Flowlet Switching*. Sinha et al. décrivent une *flowlet* comme une rafale de paquets qui appartiennent à la même fenêtre d'émission TCP d'un même flux. Vanini et al. décrivent les *flowlets* comme étant des rafales de paquets d'un même flux suffisamment espacées dans le temps pour ne pas causer de déséquencement si elles sont envoyées sur des chemins différents.

Dans un cas hypothétique où deux larges flux, nécessitant beaucoup de bande passante, ont été assignés à la même interface de sortie via ECMP. Les deux flux se marcheraient dessus et créeraient de la congestion jusqu'à ce qu'un des deux se termine. Modifier la granularité de la discrimination permet, par exemple, de changer l'interface de sortie des *flowlets* d'un des deux flux pour, au final, éviter que les flux empiètent l'un sur l'autre.

Cette proposition est celle de Vanini et al., ils proposent de modifier l'interface de sortie de *flowlets* dès lors qu'on remarque une congestion (c'est-à-dire une augmentation significative du RTT). La détection

de congestion se fait en s'assurant que le temps passé entre le dernier paquet d'une *flowlet* et le premier paquet de la *flowlet* suivante ne dépasse pas une valeur de timeout nommée le "*Flowlet Timeout*". Dans le cas des deux larges flux, l'IR aurait remarqué des congestions dues aux croissances des fenêtres d'émission des flux qui auraient dépassé les capacités des routeurs qu'ils se partageaient et aurait modifié l'interface de sortie des *flowlet*s de l'un des flux.

Vanini et al. arrivent à ce résultat en ajoutant une 6^{e} valeur en entrée de la fonction de hash. Dès que le *Flowlet Timeout* est expiré, c'est-à-dire qu'aucun paquet appartenant au même flux n'a été envoyé depuis une longue période de temps, la 6^{e} valeur est modifiée et remplacée par une valeur aléatoire. Une fois que cette valeur est modifiée, le hash est également modifié et l'interface de sortie est possiblement modifiée également.

Il est important de noter que dans l'article de Vanini et al., la valeur de *timeout* est unique pour tous les flux. Ils peuvent se permettre cela car l'article se concentre aux réseaux de datacentres. Les réseaux de datacentres sont très différents des réseaux d'opérateurs de par la topologie en arbre et la présence de la source et de la destination dans le même réseau, ce qui n'est pas forcément le cas pour les réseaux d'opérateurs qui doivent envoyer les paquets dans des AS différents par moments.

L'équilibrage devient dynamique et intelligent. Il est capable d'évoluer au fil du temps et résout les problèmes de conflit de flux d'ECMP. La simplicité du *Flowlet Switching* fait qu'il est possible et intéressant de le modifier. Dans les paragraphes suivants nous explorons *Segment Routing*, pourquoi il est intéressant de l'utiliser et pourquoi il est intéressant de le combiner au *Flowlet Switching*.

4.1.2 Plus court chemin / Chemins non-dominés

On différencie le cas mono-métrique du cas multi-métrique selon si une ou plusieurs métriques sont assignées aux liens d'un réseau. Si une seule métrique (souvent poids IGP) est assignée, alors nous sommes dans un cas mono-métrique. Si plusieurs métriques (souvent poids IGP et délai) sont assignées, alors nous sommes dans le cas multi-métrique.

Dans le cas mono-métrique, il est possible de dire qu'un chemin est "le plus court chemin" étant donné que parmi tous les chemins qui séparent une source d'une destination, il existe forcément au moins un chemin dont le poids est minimal. [1]

Dans le cas multi-métrique, la notion de plus court chemin disparaît. Entre un chemin dont le poids IGP vaut 10 et le délai 5, et un autre dont le poids IGP vaut 5 et le délai vaut 10, il n'existe pas de minimum qui permettrait de départager un "plus court" chemin parmi les deux. On dit que ces chemins sont non-dominés. Les chemins non-dominés sont les chemins qui sont meilleurs que tous les autres sur au moins l'une des métriques. Dans l'exemple précédent, le premier chemin était meilleur car il minimisait le délai et le second chemin minimisait le poids IGP.

ECMP n'étant pas capable de supporter la répartition de paquets sur des chemins multi-métriques, il est nécessaire de trouver un outil qui en est capable.

4.1.3 Choix de l'interface de sortie

Les interfaces de sortie potentielles, dans ECMP, sont celles connectées aux liens faisant partie des plus courts chemins. On sait désormais que la notion de plus court chemin n'a plus de sens et qu'il faut retravailler l'association entre paquet et interface de sortie si l'on veut prendre en compte le caractère multi-métrique du réseau.

L'ingénierie de trafic consiste à appliquer des mesures pour contrôler le trafic afin d'offrir des garanties aux flux qui transitent et d'user, de manière efficace, des ressources du réseau. Il faut utiliser l'ingénierie de trafic si l'on veut utiliser les chemins non-dominés d'un réseau multi-métrique. Il faut non seulement calculer ces chemins mais trouver un moyen de router les paquets le long de ces chemins.

1. On suppose que le réseau peut être représenté par un multigraphe qui n'est pas dynamique et dont le poids des arêtes est strictement supérieur à 0. On suppose également qu'on souhaite minimiser toutes les métriques.

Segment Routing (SR) est une technologie utilisée pour assurer le bon cheminement des paquets dans un réseau multi-métrique. SR fonctionne en ajoutant une liste de segments aux paquets. Ces segments représentent des détours par lesquels les paquets doivent passer. Un détour représente soit un nœud soit un lien.

À la réception d'un paquet contenant une liste de segments, les routeurs interprètent le premier segment et envoient le paquet en direction de l'équipement représenté par le segment (appelé *segment-hop*). Le chemin emprunté pour atteindre le segment-hop est le (l'un des) plus court(s) chemin(s) selon l'algorithme de routage [2]. Si le premier segment représente le routeur qui vient de recevoir le paquet, le routeur retire le segment de la liste avant de lire le segment suivant.

Un exemple d'utilisation de SR est présent sur la figure 1. Dans le cas (1) la liste de segments force le paquet à passer par le chemin {IR, 1, 2} et non {IR, α, 2}. Dans le cas (2), on remarque qu'il existe plusieurs chemins possibles pour atteindre le segment-hop 2, le routeur 0 effectue un équilibrage ECMP entre segment-hops en utilisant les deux chemins.

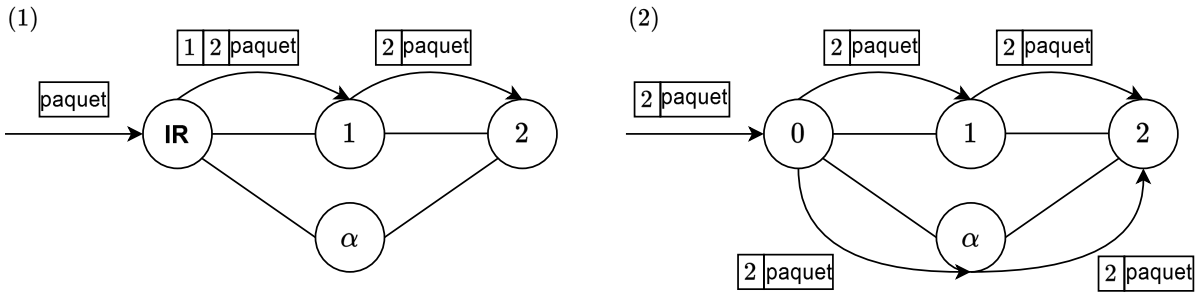

FIGURE 1 - Exemples de routage de paquets dans un réseau SR.

La liste de segments est ajoutée au paquet par l'ingress router (IR), dont on peut voir un exemple sur la figure 1. C'est donc cet IR qui devra choisir quelle liste de segments il faut ajouter au paquet et ensuite l'envoyer en direction du segment-hop qu'il aura lui même ajouté.

Il n'est plus question d'envoyer le paquet sur le meilleur chemin en direction de la destination mais sur le meilleur chemin en direction du segment-hop. C'est également de là que vient l'appellation "routage à la source" car c'est l'IR qui décide du chemin que prendra le paquet plutôt que de dépendre des décisions individuelles des routeurs. C'est la source qui décide du routage. Pour calculer la liste de segments à ajouter aux paquets, l'IR parcourt des DAGs qui résultent des algorithmes de recherche de chemins optimaux.

4.2 GOFOR et les types de DAGs

GOFOR [3] est l'outil que j'ai utilisé pour calculer les meilleurs chemins multi-métriques sous forme de meta-DAGs car c'est l'outil qui était au centre de mon TER. GOFOR est une augmentation des algorithmes de routage multi-métriques, il crée, pendant le parcours du graphe physique, les meta-DAGs. Pour pouvoir décrire le meta-DAG il faut décrire les DAGs physiques et logiques.

Les DAGs physiques sont les graphes renvoyés par les algorithmes de routage mono-métriques et représentent les arcs à prendre pour suivre les plus courts chemins. Chaque arc représente un câble reliant deux équipements physiques. Afin d'obtenir le plus court chemin d'une source à une destination, il suffit de parcourir le DAG physique enraciné à la source jusqu'à atteindre la destination. Peu importe les arcs que l'on emprunte on sait que l'on est sur un plus court chemin. Chaque nœud n'a besoin que d'un DAG physique pour pouvoir router, le DAG contient tous les autres nœuds ainsi que tous les arcs nécessaires pour les atteindre de manière optimale.

2. S'il existe plusieurs plus courts chemins pour atteindre le segment-hop, il est possible d'utiliser ECMP pour faire de l'équilibrage de charge entre segment-hops.

Les DAGs logiques sont le fruit des algorithmes de recherche des chemins multi-métriques non-dominés avec SR. L'arc d'un DAG logique représente un segment-hop, et seuls les segment-hops apparaissent en tant que nœud dans les DAGs logiques. Cela implique que tous les nœuds du graphe n'apparaissent pas dans un DAG logique et que les arcs peuvent cacher des équipements intermédiaires qui n'ont pas besoin d'apparaître car ils font partie d'un plus court chemin jusqu'au segment-hop. On rappelle que les segment-hops sont les équipements décrits par les segments dans la liste de segments par lesquels il faut passer avant de rejoindre la destination. Pour obtenir une liste de segments décrivant un (des) chemin(s) non-dominé(s) à l'aide d'un DAG logique il suffit de le parcourir et d'enfiler les segments au fur et à mesure. Contrairement aux DAGs physiques, il faut un DAG logique par point sur le front de Pareto parametré par les nœuds source et destination. C'est-à-dire que pour chaque couple (source, destination) il y a autant de DAGs logiques qu'il y a de combinaisons de valeurs de métriques ayant un chemin non-dominé pour les représenter (voir Annexe A).

On a un exemple de traduction de graphe physique en DAGs logiques dans la figure 2. On remarque 3 chemins non-dominés passant respectivement par les nœuds A, B et C qui lient une source s à une destination d. Le chemin passant par A est non-dominé car il minimise la 1re métrique, celui passant par C minimise la 2nde métrique. Le chemin passant par B est meilleur que le chemin A sur la 2nde métrique et est meilleur que le chemin C sur la 1re métrique. À partir du front de Pareto paramétré par s et d, on forme les DAGs logiques qui respectent les valeurs des métriques. Au final on obtient trois DAGs logiques qu'il suffit de parcourir pour obtenir les listes de segments à ajouter aux paquets. On choisirait le DAG logique passant par A si l'on souhaite minimiser la 1re métrique, on choisirait le DAG logique passant par B si l'on souhaite avoir un entre-deux.

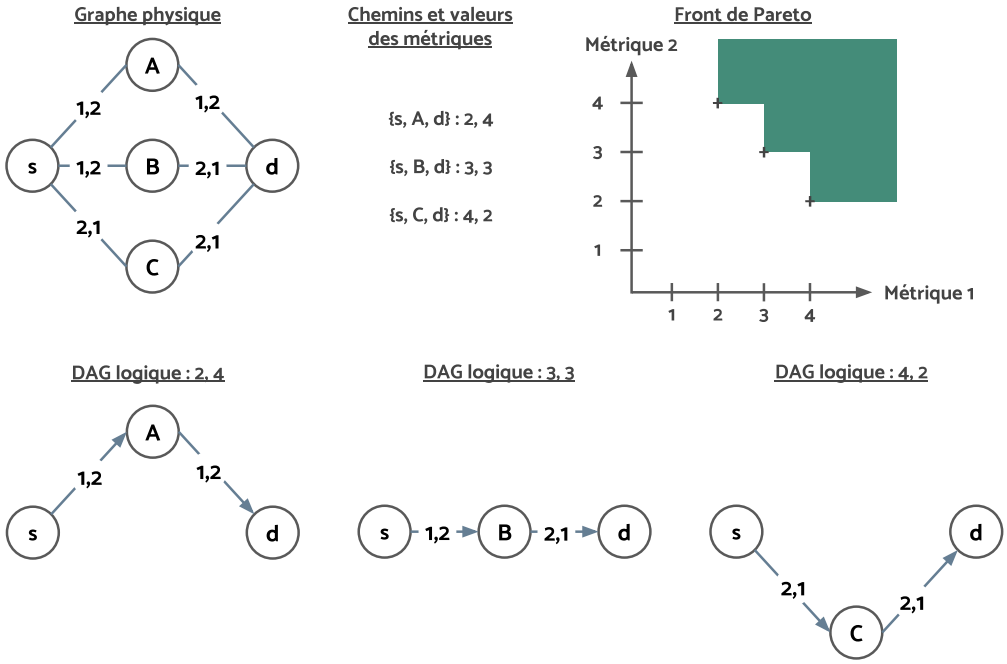

FIGURE 2 - Exemple de processus de création de DAGs logiques.

Les meta-DAGs sont une solution à un problème que subissent les DAGs logiques. Il est possible que plusieurs chemins aient les mêmes valeurs de métriques au total et qu'ils apparaissent donc sur le même DAG logique mais que ces chemins empruntent un même nœud via des chemins différents en valeurs de métriques. Auquel cas il faut s'assurer que les différents chemins n'utilisent pas les mêmes arcs sinon on risque de récupérer une fausse liste de segments au moment du parcours. Les meta-DAG résolvent ce problème en ajoutant des différenciations de chemins à chaque fois qu'il y a plusieurs chemins ayant des valeurs de métriques différentes lorsqu'ils passent par un même nœud intermédiaire.

L'exemple de la figure 3 expose un cas où un meta-DAG est nécessaire. Il y a 4 combinaisons de valeurs

de métriques ayant au moins un chemin non-dominé pour les représenter, nous nous intéressons au cas où les métriques valent 3, 3. Dans ce cas, il y a 2 chemins possibles, le premier passant par le lien séparant les nœuds s et i qui a les métriques 1, 2 puis par le lien ayant les métriques 2, 1 ; le second chemin fait l'inverse en passant d'abord par le lien 2, 1 puis 1, 2. Si on se contentait d'un DAG logique, il ne serait pas possible de différencier les deux chemins dès l'arrivée dans le nœud i lors du parcours du DAG. Il se peut qu'on fasse le choix d'envoyer un paquet sur le premier arc 2, 1 ainsi que le 2ème arc 2, 1 ce qui ne respecterait pas les valeurs de métrique 3, 3 voulues, on obtiendrait 4, 2.

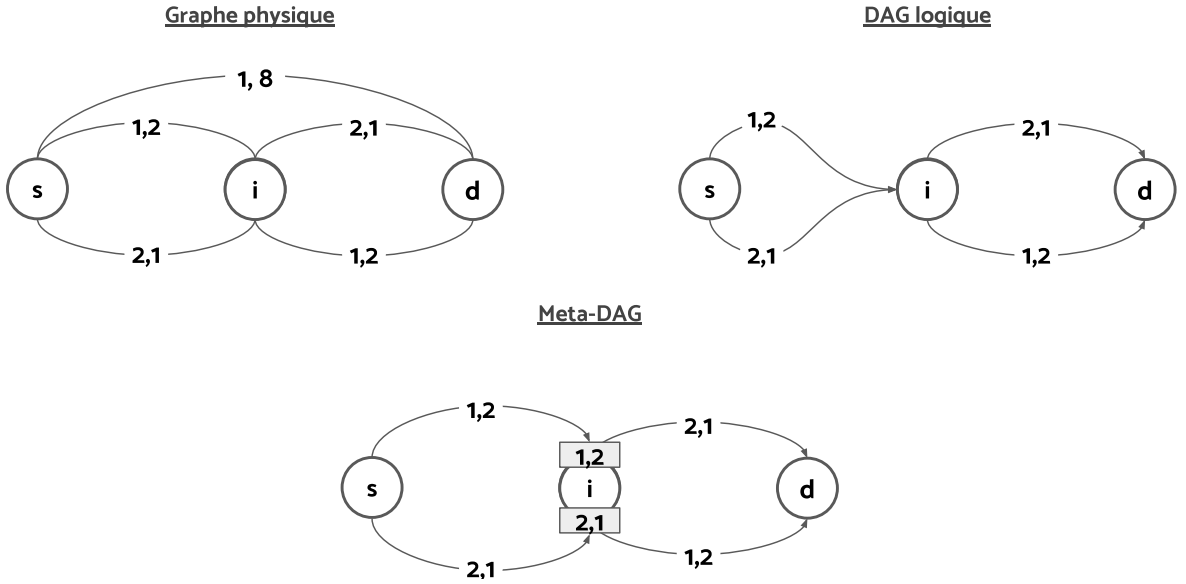

FIGURE 3 - Exemple de graphe physique, de DAG logique et meta-DAG pour des valeurs de métriques 3, 3.

Pour mon TER qui se concentrait sur GOFOR, j'ai dû trouver un moyen d'implémenter de l'équilibrage de charge à la source, c'est-à-dire augmenter la notion de routage à la source pour ajouter des listes de segments différentes aux paquets qui appartiennent à des flux différents. Les solutions que j'ai trouvées n'étaient que naïves, c'est pourquoi nous avons décidé d'essayer d'implémenter un équilibrage intelligent, notamment le *Flowlet Switching*, la solution proposée par Vanini et al. combinée à SR.

5 Missions

Ma mission durant ce stage était d'évaluer les performances d'une solution d'équilibrage de charge combinant les listes de segments du *Segment Routing* aux *flowlet*s du *Flowlet Switching*. C'est-à-dire profiter de la polyvalence de l'outil d'ingénierie de trafic SR au dynamisme et l'intelligence du *Flowlet Switching*.

L'évaluation de performances est intéressante car plusieurs facteurs peuvent influencer la qualité de l'équilibrage, en particulier :

- **La valeur de *Flowlet Timeout*** : l'article Let It Flow de Vanini et al. décrit un protocole avec une valeur de timeout unique et statique pour toutes les *flowlet*s. Cela fait sens dans un réseau interne d'un datacenter, le réseau utilisé dans l'article, étant donné que la source et la destination sont toutes deux présentes dans le datacenter et que tous les RTT sont à quelques microsecondes les uns des autres ; ce qui n'est pas le cas de réseaux d'opérateurs où la destination peut se situer dans un autre AS et donc le RTT peut varier dû à des décisions qui ne sont pas les nôtres. De plus, le RTT peut varier de plusieurs millisecondes en fonction de la destination à atteindre, même en restant dans le même AS, il n'est pas réaliste de proposer une valeur unique de timeout pour des flux dont le RTT est considérablement différent.
- **Différence entre graphe logique et graphe physique** : ECMP et le *Flowlet Switching* de Vanini et al. travaillent sur un graphe physique, c'est-à-dire que chaque nœud représente un équipement et chaque lien représente un câble reliant des équipements. Tandis que, pour SR, les DAGs logiques et les meta-DAGs ne respectent pas ces contraintes. Dans les DAGs logiques et les meta-DAGs, un arc peut représenter plusieurs câbles et équipements.

C'est pourquoi nous nous situons dans un réseau d'opérateur fermé, par exemple l'interconnexion de datacenters ou un réseau d'entreprise confiné à un réseau d'opérateur via tunnel VPN BGP MPLS. Ce type de réseau justifie l'utilisation de SR et donc de DAGs logiques et meta-DAGs, tout en rendant l'utilisation du *Flowlet Switching* possible.

5.1 Outils utilisés

Pour travailler j'utilisais P4-Utils [7], un *framework* développé par l'ETH Zurich, qui ajoute des routeurs P4 à Mininet.

Mininet est un logiciel de simulation réseau open-source. Il simule des switches, des routeurs, des contrôleurs et des hôtes via des environnements définis par logiciel. Tous ces équipements créent des interfaces réseau virtuelles sur la machine qui héberge Mininet et c'est sur ces interfaces que passeront les paquets.

P4 est un langage de programmation conçu pour être déployé sur le plan de données des équipements réseau. Il permet de définir le traitement des paquets contrairement aux routeurs traditionnels qui ne permettent pas la définition de protocoles, seulement leur configuration. Cette capacité à définir des protocoles facilite le développement de nouvelles fonctionnalités, par exemple la combinaison de listes de segments et le *Flowlet Switching*.

P4 n'est pas exempt de limitations, pour pouvoir avoir une vitesse de traitement des paquets comparable aux routeurs traditionnels, des choix ont été faits notamment :

- **pipeline de traitement** : le paquet doit obligatoirement passer par les différentes étapes du pipeline dans l'ordre. Le code P4 ne permet que de décrire ce qu'il se passe dans ces étapes. Chaque architecture possède ses propres caractéristiques, l'architecture v1model qui est utilisée par les simple switch BMV2 de P4-Utils est formé d'un *parser*, d'une *ingress pipeline*, d'un *traffic*

manager, d'une *egress pipeline* et d'un *deparser*.
— **linéarité et conclusion...** : P4 interdit la présence de boucles pour s'assurer de l'absence de boucles infinies et donc s'assurer que le programme se termine bien [1].
— **...en un temps fini** : Si le compilateur calcule que le temps maximal qu'un paquet peut passer dans la pipeline est trop élevé, il renverra une erreur.

5.2 Choix d'implémentation

Le principal choix a été de définir comment les listes de segments allaient être ajoutées aux paquets à l'aide de P4. Durant mon TER, j'avais défini deux solutions que je n'avais pas implémentées, la distribution et le *random walk*.

5.2.1 Distribution

Dans la plus simple des deux solutions, le plan de contrôle calcule les listes de segments en parcourant les meta-DAGs renvoyés par GOFOR et les distribue au plan de données. Lorsque le plan de données reçoit un paquet, il crée un hash qui lui permet de choisir parmi les différentes listes de segments.

Cette simple solution va nous permettre d'explorer, à travers un exemple, plus en détail le fonctionnement de P4, en particulier l'architecture v1model et son *parser* et son *ingress pipeline*.

Le *parser* est un automate déterministe qui lit les premiers octets d'un paquet et remplit les structures de données définies dans le code. Les structures de données sont appelées header dans le code. Si un header a été rempli, un flag de validité est mis à 1 pour indiquer à la suite du programme quels en-têtes on été lus et remplis.

Typiquement, l'*ingress pipeline* d'un programme P4 va démarrer par un test de validité d'un en-tête. Pour notre cas, on peut tester la présence ou non de segments dans l'en-tête, en ajouter si aucun n'a été trouvé ou procéder au routage en direction du segment-hop sinon.

Suite à quoi, l'*ingress pipeline* va faire appel à des actions et des *match action tables*, ou MAT. Les actions sont très similaire à des fonctions telles qu'on pourrait en voir en C. Elles prennent en argument des champs qu'elles souhaitent lire ou modifier, et sont exécutées dès lors qu'elles sont appelées. Elles peuvent être appelées directement dans le code, ou indirectement à travers les MATs. Ces MATs associent des clés à des actions pré-paramétrées. Les clés sont soient des headers lus par le *parser* ou des méta-données calculées entre-temps, par des actions ou d'autres MATs par exemple.

Un exemple simple de MAT est la commutation ethernet. Il suffit de créer une table qui a pour clé : l'adresse ethernet de destination et comme action : une action qui définit sur quelle interface il faut commuter le paquet.

Un exemple moins simple est celui de la figure 4, où l'on voit un réseau de quatre nœuds. On s'intéressera aux chemins qui séparent la source A de la destination D, en particulier comment ces chemins apparaissent dans le code P4 du routeur A. Comme expliqué précédemment, le *parser* remplit les headers Ethernet, IPv4 et TCP, le header SR n'est rempli que si des segments sont présents dans l'en-tête. Si aucun segment n'est présent, le test de validité du header SR renverra faux, auquel cas le routeur fera passer le paquet dans la MAT IPV4 LPM qui associe les adresses d'un préfixe IP à une action qui garde en mémoire le segment destination. Après avoir calculé un hash, le segment destination et la valeur de hash sont utilisées pour choisir une liste de segments via la MAT LST SEG. Si le hash calculé vaut 0 alors l'application de la table écrira les segments B et D, sinon l'action écrira les segments C et D.

Cette solution est très couteuse en mémoire, en effet le nombre de listes de segments explose rapidement avec la taille du réseau. En se rappelant qu'il y a autant de meta-DAGs qu'il y a de points sur le front de Pareto paramétré par une source et une destination, et que chaque meta-DAG peut héberger plusieurs

1. Étant donné que Alan Turing[8] avait montré qu'il est impossible de prédire si un programme se termine ou non dans une machine de Turing. Autant ne pas prendre de risques et retirer la possibilité même d'une boucle.

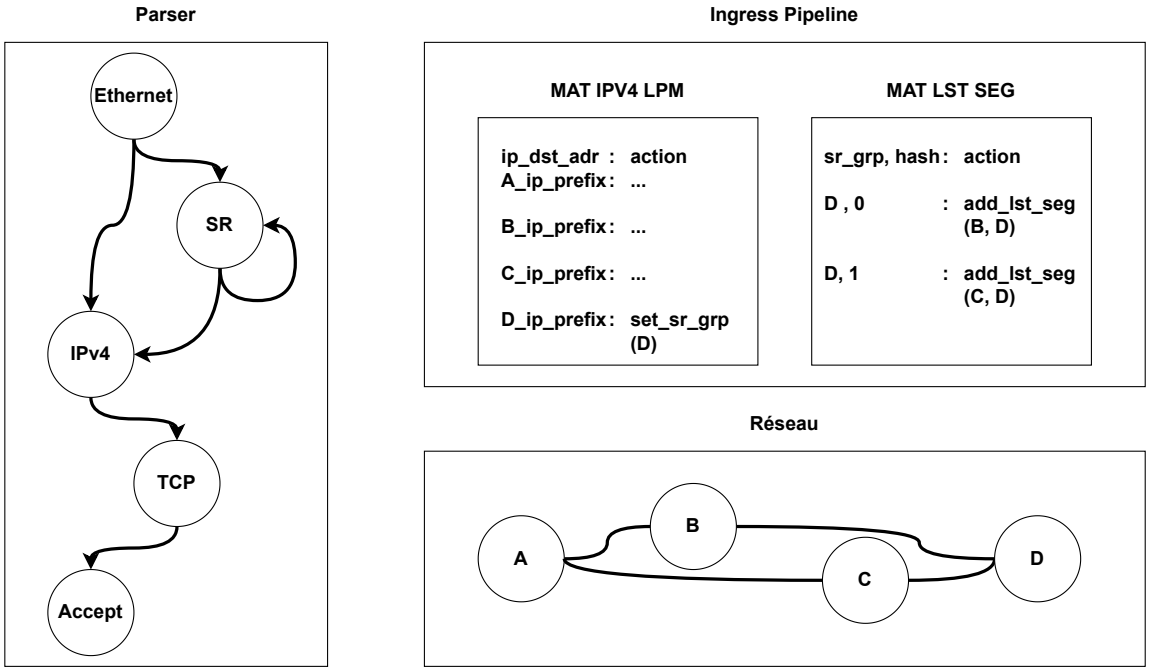

FIGURE 4 - Représentation d'un graphe physique et du *parser* et *ingress pipeline* du routeur A écrivant des listes de segments.

listes de segments, on se rend rapidement compte que cette solution n'est pas viable pour les grands réseaux.

De plus, en fonction de la topologie, il se peut que la solution de distribution favorise certains liens plutôt que d'autres. Dans l'exemple de la figure 5, qui représente un meta-DAG, on remarque deux familles de chemins. La famille composée de l'unique chemin 1, 2, 7 et les autres chemins passant par le nœud 3. Étant donné l'hypothèse que les liens ont tous des capacités égales, idéalement, il faudrait envoyer la moitié du trafic sur le chemin 1, 2, 7 et l'autre moitié sur les chemins passant par le nœud 3. Pourtant, en termes de liste de segments, le chemin 1, 2, 7 n'est représenté que par une seule liste alors que les autres chemins sont représentés par 3 listes de segments différentes. Cela implique que, lors du choix aléatoire d'une liste de segments, il y a 3 chances sur 4 qu'une liste de segments représentant l'un des chemins passant par le nœud 3 soit choisie. Donc que 3/4 du trafic sera envoyé sur ces chemins plutôt que l'idéal 1/2.

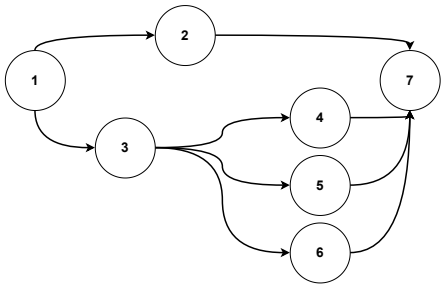

FIGURE 5 - Exemple de meta-DAG antagoniste à la solution de distribution.

5.2.2 Random walk

La solution *random walk* consiste à stocker, dans la mémoire du plan de données, les meta-DAGs et à les parcourir, avec une marche aléatoire, pour obtenir la liste de segments à ajouter aux paquets qui doivent être routés. Le principal frein à cette solution est que le plan de données n'est pas fait pour stocker des structures complexes comme les meta-DAGs. Il est compliqué, pour nous en tant qu'humain, de trouver une traduction élégante, et il est compliqué, en termes de complexité spacio-temporelle pour les programmes qui tournent, de faire fonctionner des algorithmes sur des structures de données non triviales comme les meta-DAGs.

La version du *random walk*, telle qu'elle apparaît dans mon rapport de TER[9], a été modifiée afin d'être plus propre et générique. Plutôt que de créer une MAT par nœud dans le graphe pour représenter le meta-DAG, on représente les meta-DAGs en fonction du degré de profondeur des nœuds. C'est-à-dire que, peu importe le meta-DAG, si un segment-hop est accessible en 3 sauts depuis la source, alors ce segment-hop ira dans la MAT des segment-hops qui sont à distance 3 de la source. Les meta-DAGs sont mélangés dans les MATs mais seront différenciés lors de l'exécution grâce aux clés.

S'il ne fallait que stocker un seul DAG logique dans la mémoire d'un routeur P4 on aurait pu, pour chaque profondeur de DAG, créer une table qui associe le segment courant et une valeur de hash aux différents segment-hops accessibles par le segment courant. Sans oublier de calculer une nouvelle valeur de hash en fonction du nombre de segment-hops possibles pour le prochain saut.

L'exemple de MATs de la figure 6, qui réutilise le graphe physique de la figure 4 avec les nœuds A, B, C et D où A est la source (et donc aussi l'IR) et D la destination, illustre ces propos. Le parcours commence dans le nœud A, on initialise donc le segment courant à A pour démarrer le parcours du DAG logique. Une valeur de hash est calculée et est rapportée modulo le nombre d'arcs sortant de A. À partir de ces valeurs on peut démarrer le parcours. La MAT DEPTH 1 associe le segment courant, pour l'instant A, et une valeur de hash, disons 0, à une action qui ajoute le segment B et calcule une nouvelle valeur de hash modulo le nombre d'arcs sortant du nœud B, ici 1. De ces nouvelles valeurs, c'est-à-dire un segment courant valant B et un hash valant 0, la MAT DEPTH 2 ajoute le dernier segment : D et le parcours est fini, la destination est atteinte.

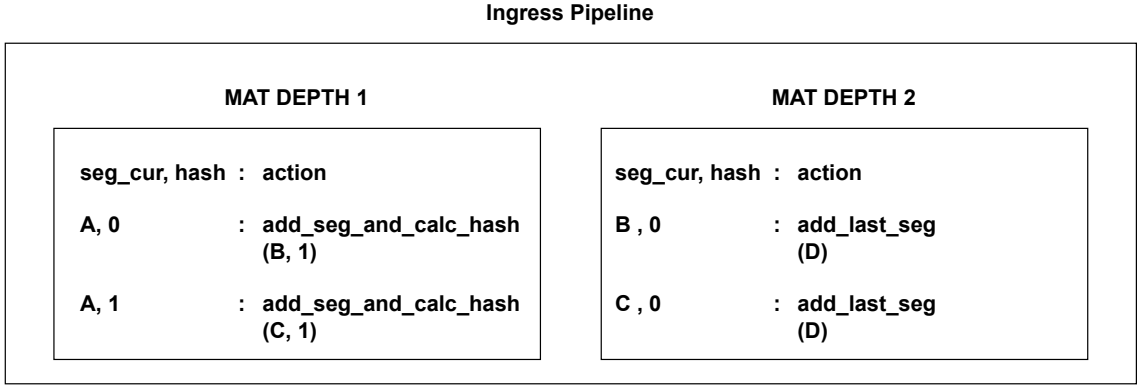

FIGURE 6 - *Ingress Pipeline* d'un routeur utilisant le *random walk* pour ajouter des listes de segments.

Maintenant il faut se rappeler qu'on travaille sur des meta-DAGs et non des DAG logiques, on rajoute donc des différenciateurs de chemins dans les clés et les actions des MATs. Il faut également se rappeler qu'on souhaite router sur l'intégralité du réseau et non juste sur un nœud en particulier. On ajoute alors aux clés des MATs : le segment destination que l'on souhaite atteindre. Au final, cette nouvelle solution *random walk* nécessite des MATs qui associent un quadruplet (segment courant, segment destination, valeur de hash, différenciateur de chemins) à une action qui met à jour le segment courant, qui remplace

la valeur de hash et qui met à jour le différenciateur de chemin si besoin. [2]

On a réussi à réduire l'espace mémoire occupé comparé à la solution de distribution mais cette solution se heurte à une autre limitation des équipements. Le problème étant que le nombre de MATs que le paquet peut traverser est limité et que cette solution à elle seule occupe autant de MATs qu'il y a de segments à ajouter.

La solution *random walk* est également susceptible aux problèmes de répartitions inégales en fonction de la topologie, similairemenet à la solution de distribution. Si l'on inverse les arcs du meta-DAG de la figure 5, et que l'on essaye donc d'envoyer les paquets d'une source 7 à une destination 1, on remarque que les chemins passant par le nœud 3 ont plus de chance d'être choisis. En effet, dès la première étape du *random walk*, il y a 3 chances sur 4 que le segment-hop choisis fasse partie des chemins passant par le nœud 3. On n'atteindra donc pas la répartition 1/2, 1/2.

Maintenant que l'on sait comment on peut ajouter les segments aux paquets, on peut évaluer les performances d'une fusion entre SR et le *Flowlet Switching*.

2. En théorie, il faudrait une 5^{e} clé pour différencier les meta-DAGs qui peuvent séparer une source d'une destination, mais en usant du différenciateur de chemin, qui décrit les valeurs des métriques du chemin déjà parcouru, il est possible de savoir, implicitement, dans quels meta-DAGs le parcours est en train de se faire.

6 Travail effectué

Le dépôt git [1] contient :
- Le code source de GOFOR (écrit par les auteurs de l'article) qui traduit les fichiers décrivant une topologie en meta-DAGs. (voir gofor_source/)
- Le code Python pour Mininet qui initialise les simulations, fait les appels à GOFOR, fait office de plan de contrôle pour les routeurs P4 et récupère les données pour évaluer les performances. (voir network.py, get_dag.py, controller_distribution.py, controller_random_walk.py)
- Le code P4 qui définit le traitement des paquets au sein des routeurs P4. (voir distribution.p4, *randomwalk*.p4, include/)
- Le code permettant de lancer, et traiter les données, des simulations. (voir fetch_data.sh, stats.py)

La solution de distribution et *random walk* ont été implémentées en P4 avec succès. J'ai également réussi à ajouter le *Flowlet Switching* aux deux, ce qui n'a pas été une balade de santé étant donné que P4 est un langage très éloigné des paradigmes que nous avons l'habitude de voir. Le code P4 est générique c'est à dire que toutes les topologies sont supportées. Le code Mininet que j'ai écrit peut être modifié pour représenter toutes les topologies tant que le graphe physique possède moins de 10 nœuds (limité par le code source de GOFOR) et tant qu'il n'est pas un multigraphe (les multigraphes ne sont pas supportés par P4-Utils).

Le programme crée tout d'abord la topologie avec des poids IGP et des délais assignés aux liens ainsi que des routeurs P4 et des hôtes. Cette topologie est transmise au code source de GOFOR qui produira les meta-DAGs. Ensuite, en fonction du type de la solution (distribution/*random walk*), les meta-DAGs seront manipulés avant d'être envoyés au routeurs afin de mettre à jour leurs MAT. Dans les deux solutions, les meta-DAGs sont parcourus. Dans la solution de distribution ils sont parcourus de bout-en-bout afin de produire les listes de segments directement. Dans la solution *random walk*, à chaque étape du parcours une entrée est ajoutée aux MATs afin de reproduire le meta-DAG au sein du routeur.

Pour effectuer une simulation, des commandes sont envoyées aux hôtes. Par exemple, on peut envoyer à certains hôtes la commande `time iperf -c 10.0.0.1 -n 100M` qui renvoie le temps qu'il a fallu pour envoyer 100M à l'hôte ayant l'adresse 10.0.0.1.

Pascal MÉRINDOL m'a proposé d'évaluer les performances sur une topologie qui favoriserait la solution *random walk* afin de voir si les résultats obtenus sont similaires aux résultats attendus. Une version très similaire de la topologie nécessaire pour obtenir le meta-DAG de la figure 5 a été choisie.

L'étude des performances a été effectuée en mesurant le Flow Completion Time (FCT), c'est-à-dire le temps écoulé entre l'envoi du premier paquet d'un flux et la réception du dernier acquittement dudit flux. Les paramètres étudiés sont :
- le type de solution pour écrire les listes de segments : distribution, *random walk*.
- le nombre d'hôtes : les hôtes émetteurs sont connectés à l'équivalent du routeur 1 de la figure 5 et l'hôte serveur est connecté au routeur 7. J'ai fait le choix d'utiliser 3, 5 ou 7 hôtes émetteurs.
- la taille des flux : chacun des hôtes envoie 1, 10 ou 100 Megaoctets.
- le *Flowlet Timeout* : comme expliqué précédemment, on ne peut reproduire à l'indentique la version du *Flowlet Switching* de l'article de Vanini et al., il faut modifier la valeur du *Flowlet Timeout* pour chaque liste de segments. En faisant l'hypothèse d'avoir connaissance des délais entre les liens, typiquement en étant dans un réseau faisant des calculs de chemins optimaux multi-métriques, on peut associer une valeur de RTT aux listes de segments. J'ai fait le choix d'associer aux listes de segments une valeur de *Flowlet Timeout* valant le RTT + Δ où Δ représente

1. `https://filesender.renater.fr/?s=download&token=36b57077-45a6-4adc-aa44-8ed610d108ab`

l'indulgence au retard. J'ai choisi Δ valant 10%, 20%, 30%, 40% et 50% du RTT.
Malheureusement, aucun avantage notable n'a été relevée pour la solution *random walk* (voir figure 7) alors que notre but était de la favoriser. Aucune différence notable n'a été observée lorsqu'on modifiait les différents paramètres cités plus haut. On remarque tout de même que les solutions introduisant le *Flowlet Switching* performent mieux que les solutions naïves où les listes de segments ont été attribuées ad vitam æternam aux flux, ce qui est une bonne nouvelle, tout ce travail n'a pas été complètement vain.

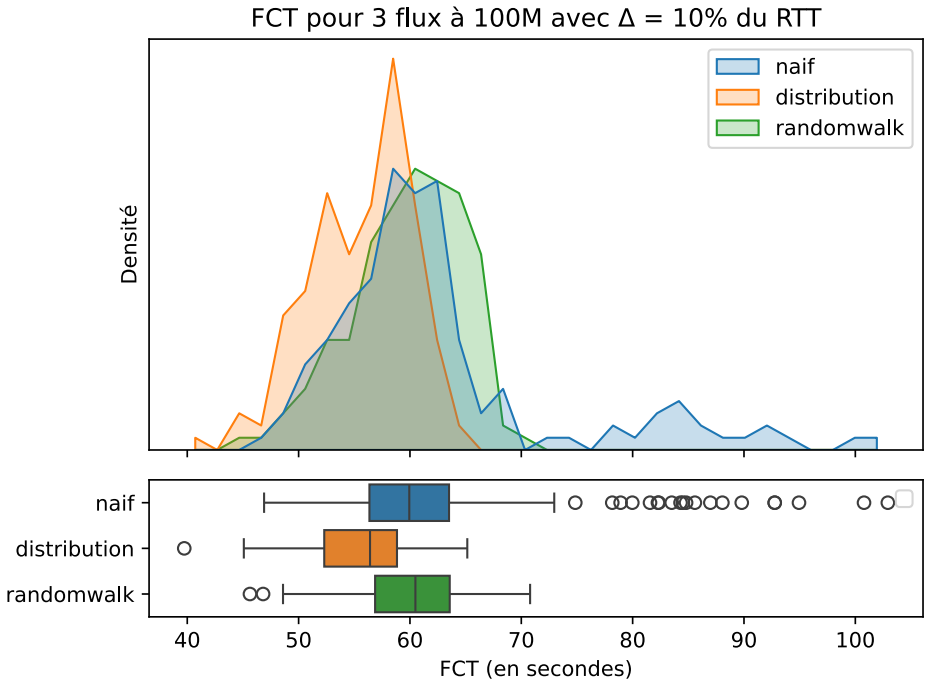

FIGURE 7 - Évaluation des performances des solutions de distribution et *random walk* comparées à une solution naïve (50 simulations par solution).

Comme on peut le voir dans la figure 7, l'avantage que l'on a voulu procurer à la solution *random walk* n'a pas porté ses fruits. Malgré cela, on remarque que le *Flowlet Switching* réduit le FCT. Pour la solution de distribution, on le remarque car la médiane (environ 56s) se situe où le 1er quartile est pour la solution naïve. Pour la solution *random walk*, on le remarque par l'absence de valeurs aberrantes au delà de 70s, qui sont très présentes pour la solution naïve. Pour référence, si l'on fait tourner un algorithme de maximisation des flux sur la topologie, qu'on effectue les simulations en forçant les flux à suivre les chemins décrits par l'algorithme, le FCT est de 27 secondes.

Après avoir cherché pourquoi la topologie n'avait pas favorisé la solution *random walk* j'ai trouvé plusieurs explications possibles par exemple : la topologie avait bel et bien favorisé la solution *random walk* mais la solution *random walk* est significativement plus lente que la solution de distribution. Cela peut être dû à un temps de traitement des paquets plus long. Il s'avère que les deux solutions produisent un temps de traitement similaire à quelques microsecondes près, ce n'est donc pas cette raison là.

J'ai tout de même lancé des simulations sur une topologie différente qui s'approche plus de ce que l'on peut croiser dans un réseau d'opérateur. Les résultats que l'on voit sur la figure 9 sont ceux de simulations lancées sur la topologie de la figure 8. Cette nouvelle topologie distribue les hôtes et serveurs un peu partout dans le réseau afin d'éviter les goulets d'étranglement au niveau des équipements connectés à plusieurs hôtes ou serveurs.

Les chemins optimaux obtenus par un algorithme de maximisation des flux ont été dessinés sur la figure, le FCT moyen, pour un flux à 100Mo, obtenu sur une simulation où l'on force les paquets à suivre ces chemins est 11.6 secondes. Ce qui est quasiment deux fois moins que le minimum de la solution

de distribution qui est la meilleure des solutions en termes d'extrema. Étrangement, la médiane de la solution *random walk* est moindre que que celle de distribution. Cela implique que la topologie sensée favoriser la solution *random walk* l'a en fait défavorisé.

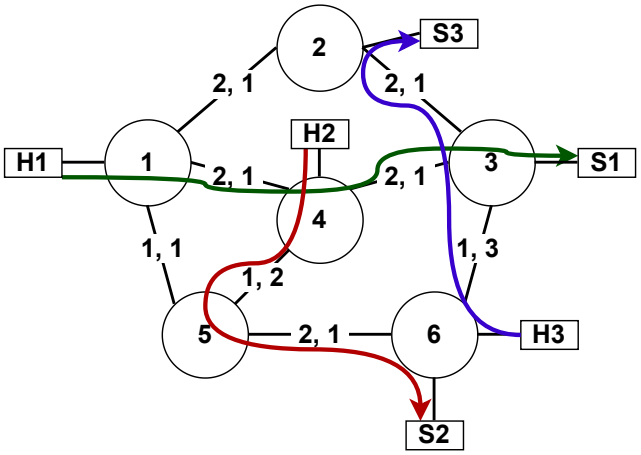

FIGURE 8 - Topologie et meilleurs chemins selon un algorithme de maximisation des flux séparant hôtes notés H et serveurs notés S.

Au final, on remarque que l'introduction du *Flowlet Switching* améliore bel et bien le FCT dans un réseau multimétrique mais qu'il reste encore beaucoup de marge avant de s'approcher de la perfection.

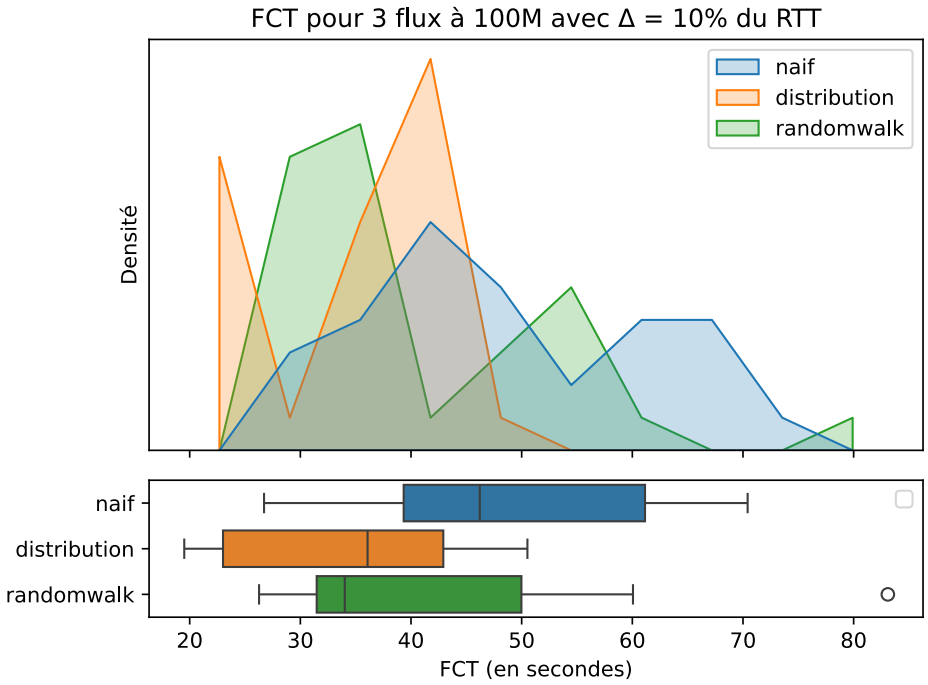

FIGURE 9 - Évaluation des performances sur la nouvelle topologie (20 simulations par solution).

7 Réflexions a posteriori

Après avoir discuté avec mes encadrants concernant les simulations, ils m'ont dit que, d'habitude, pour simuler des flux il aurait fallu suivre des lois exponentielles. En particulier des chaînes de Markov parametrées par des lois exponentielles (pour le nombre et la taille des flux) et normales (pour le nombre d'hôtes émetteurs).

De plus il aurait été intéressant de recréer les simulations de l'article Vanini et al. afin de voir si les modifications apportées au *Flowlet Timeout* et aux DAGs parcourus changent les résultats. Typiquement, on aurait pu étudier la vitesse de convergence vers la répartition optimale des flux ou comparer le *Flowlet Switching* à d'autres méthodes comme ECMP ou CONGA.

Il aurait également été intéressant d'utiliser des machines Tofino (gamme P4-programmable de routeurs Intel) plutôt que l'environnement simulé P4-Utils afin d'être au plus proche des équipements utilisés dans les réseaux d'opérateurs. D'autant plus que les équipements Tofino n'utilisent pas l'architecture v1model utilisée dans les Simple Switch de P4-Utils, il aurait sûrement été possible d'utiliser de nouvelles fonctionnalités.

Le code que j'ai écrit est loin d'être parfait. Il est fait pour, et uniquement pour, supporter des simulations P4-Utils sur des graphes (qui ne sont pas des multigraphes) ayant moins de 10 nœuds, dont les métriques sont forcément le poids IGP et le délai. Si on introduisait une 3^{e} métrique, le code ne fonctionnerait pas correctement. Il aurait été intéressant de rendre la solution plus générique, afin d'être retravaillée facilement par de futur·e·s utilisateur·rice·s pour proposer des améliorations ou simplement simuler l'utilisation de SR et du *Flowlet Switching*.

8 Conclusion

8.1 Bilan des travaux et résultats

Après avoir travaillé sur des solutions permettant l'ajout de listes de segments aux paquets durant mon TER, j'ai pu utiliser ces solutions pour évaluer les performances d'une fusion entre *Segment Routing* et *Flowlet Switching* dans un *framework* Mininet nommé P4-Utils.

J'ai dû écrire le code des simulations, c'est-à-dire la création de la topologie et la gestion du plan de contrôle et de données en conséquence. Après avoir modifié les valeurs de *Flowlet Timeout* pour supporter une variété plus grande de cas d'usages, les simulations ont montré qu'il vallait la peine d'utiliser le *Flowlet Switching* même dans un réseau multimétrique faisant tourner SR.

Les résultats ne sont pourtant pas parfaits, il reste encore beaucoup de marge pour améliorer l'équilibrage de charge à la source.

8.2 Bilan des apports du stage

Ce stage, avec mon TER, étaient mes premiers essais dans le monde de la recherche, monde que j'avais hâte de découvrir étant donné que j'aurais aimé possiblement le rejoindre. Malheureusement, même en ayant apprécié étudier SR, *Flowlet Switching*, P4 et Mininet, j'ai produit pour une deuxième fois un résultat en deçà de mes attentes.

Concernant les apports pour le laboratoire, en ayant approfondi l'article GOFOR, j'ai découvert une erreur dans le draft de l'article. Après leur en avoir fait part, ils m'ont dit qu'ils allaient le corriger.

Bibliographie

[1] F. Veisi Goshtasb, J. Montavont, and F. Theoleyre, "Energy efficient and reliable maintenance for sdn-based scheduled wireless networks," in *IEEE Consumer Communications amp ;amp ; Networking Conference (CCNC)*, IEE, Jan 2024.

[2] S. Si-Mohammed, T. Begin, I. Guérin Lassous, and P. Vicat-Blanc, "Hints : A methodology for iot network technology and configuration decision," *Internet of Things*, vol. 22, p. 100678, 2023.

[3] Q. Bramas, J.-R. Luttringer, and P. Mérindol, "A simple and general operational framework to deploy optimal routes with source routing," 2023.

[4] Q. Bramas, A. Lamani, and S. Tixeuil, "The agreement power of disagreement," *Theoretical Computer Science*, vol. 954, p. 113772, 2023.

[5] S. Sinha, S. Kandula, and D. Katabi, "Harnessing tcp's burstiness with flowlet switching," in *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, Citeseer, 2004.

[6] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow : Resilient asymmetric load balancing with flowlet switching," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 407–420, USENIX Association, Mar. 2017. `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini`.

[7] ETHZ, "P4-utils github repository." `https://github.com/nsg-ethz/p4-utils`.

[8] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[9] F. Hardy, "Rapport de ter." `https://filesender.renater.fr/?s=download&token=36b57077-45a6-4adc-aa44-8ed610d108ab`.

A Précisions sur les DAGs logiques

Il est nécessaire d'avoir au moins autant de DAGs logiques qu'il y a de couples (source, destination) étant donné que seuls les segment-hops apparaissent en tant que nœuds. Cela implique que, contrairement aux DAGs physiques, il n'est pas toujours possible de "réutiliser" le résultat d'une recherche de chemin optimal entre deux nœuds pour atteindre une autre destination. Dans l'exemple de la figure 10, on remarque que le DAG logique de la source s à la destination A n'est pas réutilisable pour le DAG logique de la source s à la destination d car le nœud A fait partie du plus court chemin entre s et d et est n'a donc pas besoin d'apparaître dans le DAG logique. S'il apparaissait, lors du parcours du DAG logique on ajouterait un segment supplémentaire qui n'est pas nécessaire. Il faut donc avoir au moins un DAG logique par couple (source, destination).

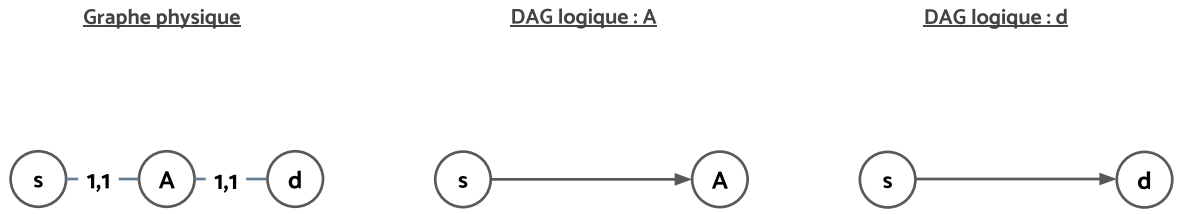

FIGURE 10 - Illustration de DAGs logiques différents pour des destinations différentes

Le même raisonnement peut être fait pour expliquer pourquoi il faut un DAG logique par point sur le front de Pareto paramétré par une source et une destination. Les DAGs pouvant être différents, il ne faut pas croiser des arcs qui servent à minimiser le délai et d'autres qui servent à minimiser le poids IGP, par exemple.