École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

Master 2 Réseaux, Université Pierre et Marie Curie

Rapport de stage final

en vue de l'obtention du
diplôme d'ingénieur ENSIIE
et du Master Réseau UPMC
Tuteur Académique : Yacine Ghamri
Maitre de Stage : Timur Friedman
Encadrant technique : Jordan Augé

Preparing the next-generation traceroute

Inês Ribeiro

UPMC Sorbonne Universités & CNRS,
du 1 Avril au 30 Septembre 2011

Remerciements

Je souhaiterais remercier Timur Friedman et Jordan Augé pour leur soutien et leurs indications précieuses qui m'ont permis de réaliser ce stage dans les meilleures conditions. Je remercie également Yacine Ghamri pour sa patience et pour la souplesse dont il a fait preuve à mon égard. Je remercie enfin toute l'équipe NPA du LIP6 pour leur accueil, en particulier Sebastien Tixeuil et Serge Fdida, ainsi que Marguerite Sos pour la charge de travail supplémentaire qu'elle a dû gérer pour moi.

Abstract

Traceroute is an uniquitous tool available in all Linux distribution, and in Windows under the name "tracert". So when we need to find the topology of a network in research experiments, we naturally use it. But traceroute has a problem when load-balancers appear in the scanned network. It entailed the creation of a new version : paris-traceroute. On the other hand, algorithms have been found to reduce redundancy and improve accuracy when exploring a network. Those algorithms will slightly or considerably change the way traceroute has to be used. They will for example need it to begin at the end, going from destination to source. But traceroute or paris-traceroute are one-purpose tools, which makes it difficult to reuse them in particular ways, like changing the number of probes sent depending on the round. The algorithms users/testers will then need to write their own probing traceroute-like tool whereas the basic principle is exactly the same : sending a probe to a certain node, then get an answer from it that gives us its IP. It is to prevent this useless work of rewriting the probing system that we decided to write a traceroute library that will make the realisation of new algorithms using the traceroute principle much simplier. This library will make available basic commands to easily send a probe in the network and get the answer. With this library, write a new algorithm will just entail to write it at an high level, by managing the use of probes and the way they have to be send. All the network part is now written for good. We then decided to use our library mesure the topology of the PlanetLab testbeds to discover and analyse the load-balancers present in it. PlanetLab is a network composed of many sources made available by many laboratories or schools or firms, that are connected over the internet. So the topology resulting of it is quite unknown and so the use of this network in experiments will influence the results, some of which could be explained by the topology. But if the user does not know the topology, he cannot be sure that a particular result is notable or is just a result of the network configuration. The topology of PlanetLab is measured and monitored by a tool named TopHat, which currently consistd of traceroute launches. We will use our library to add the MDA algorithm (see description of MDA in the context part of this document) and thus be able to generate exhaustive mesurements of the load-balancers. TopHat will then be able to give this useful measurements and results to the users. They will for example be able to choose paths with no load-balancers. We then realised some analysis of the PlanetLab topology and present our main findings in relation to the presence of load balancers and their caracterisation.

Résumé

Traceroute est l'outil de référence lorsqu'il s'agit d'explorer les chemins qui composent un réseau. Il travaille avec différents protocoles tels que UDP et TCP, et fournit en sortie les différentes IP traversées ou des "*" s'il n'obtient pas de réponse. Son intégration aux distributions Linux le rend disponible sur la plupart des machines, ce qui facilite la réalisation de mesures distribuées l'utilisant et le recoupement des données obtenues. Cependant, cet outil n'est pas forcément adapté à l'utilisation qui en est faite, et diverses améliorations ont été proposées. Pourtant, celles-ci restent confinées par leur spécificité et le problème précis qu'elles traitent, alors qu'il serait plus judicieux de pouvoir en quelque sorte les additionner. Elles s'inspirent de plus du principe de traceroute mais ne peuvent pas le réutiliser car il n'est pas assez paramétrable. Chacune de ces améliorations a donc nécessité l'écriture d'un nouvel outil. Le principe de traceroute étant d'envoyer une probe à un certain nœud puis obtenir son IP grâce à la réponse, il serait plus judicieux d'écrire un programme de base permettant de faire cela, tout en pouvant par dessus ajouter les améliorations de son choix. Il serait donc souhaitable que cet outil soit proposé sous forme de bibliothèque, qui permettrait comme le fait traceroute de proposer à tous la même chose, l'envoi de probe, facilitant la distribution de mesures, tout en supportant les modifications personnelles et en les rendant elles-mêmes distribuables. C'est dans ce but qu'il a été décidé de concevoir une bibliothèque traceroute. Notre bibliothèque intègre les différentes améliorations qui ont été proposées selon les différents usages de traceroute et est très paramétrable, ce qui facilite l'écriture et l'ajout d'améliorations, ou de nouveaux algorithme utilisant le principe d'envoi de probe de traceroute. Nous avons ensuite utilisé notre bibliothèque sur le réseau PlanetLab, qui est un réseau constitué de nombreux nœuds mis à disposition par diverses entités, telles que des universités ou des laboratoires, et reliées à travers internet. Ce réseau étant relié par Internet, la topologie qui en résulte est inconnue. Nous utilisons donc la bibliothèque, et l'algorithme MDA que nous lui avons ajouté, afin de rechercher la topologie et plus particulièrement caractériser les load-balancers qui composent PlanetLab. Ces analyses et notre bibliothèque avec MDA seront mis a disposition sur TopHat, l'outil de mesure de PlanetLab, afin de permettre aux utilisateurs de connaître la topologie et de pouvoir agir ou conclure sur leurs résultats en conséquence.

Nous verrons dans un premier temps le contexte dans lequel s'inscrit notre bibliothèque, vis-à-vis de traceroute et de la mesure de topologie. Nous verrons ensuite l'état de l'art en ce qui concerne les mesures de topologies. Nous nous intéresserons alors à notre bibliothèque et à la façon dont elle répond aux besoins énoncés, tant au niveau des spécifications que lors de l'implémentation. Enfin nous utilisons notre bibliothèque et l'algorithme MDA pour sonder le réseau PlanetLab. Nous analyserons les résultats obtenus pour caractériser les load-balancers présents dans ce réseau. Nous conclurons sur les perspectives de notre bibliothèque.

Table des matières

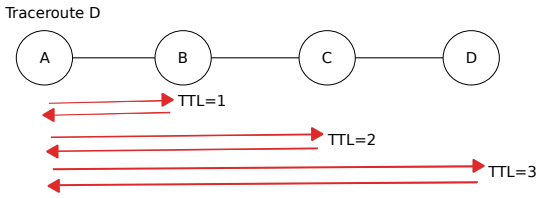

FIGURE 1 – exemple de découverte d'un chemin avec traceroute

1 Contexte

1.1 Bref descriptif de Traceroute

1.1.1 Principe de fonctionnement

Traceroute est le programme de base permettant de connaître le chemin parcouru entre deux nœuds (moniteurs) d'un réseau. Il utilise pour cela le champ TTL du protocole IP. Le TTL est le champ du paquet qui lui indique en pratique combien de nœuds il peut encore traverser avant d'être détruit. Un paquet envoyé vers une destination D avec un TTL de valeur i permettra donc de connaître le ième nœud du chemin entre la source et D. arrivé à ce ième nœud, le TTL arrivera à 0, ce qui provoquera une erreur, la destruction du paquet et une réponse du ième nœud vers l'emetteur du paquet. Ce paquet-réponse contient les 64 premiers octets du paquet envoyé, à partir desquels on pourra associer cette réponse au paquet envoyé. Traceroute utilise principalement le port destination de la sous-couche d'IP (UDP ou TCP) pour reconnaître à laquelle de ses probes le paquet répond. Il peut également utiliser ICMP avec le paquet de type 'ECHO request', ce qui est le cas de la version Windows de traceroute. Le choix du protocole est important car il va influencer les réponses obtenues par le réseau mesuré [9]. Selon le protocole utilisé, les nœuds sondés vont avoir plus ou moins tendance à répondre. Ainsi, c'est en utilisant ICMP qu'on a le plus de chance d'obtenir une réponse. Pour chaque TTL, traceroute va envoyer trois probes par défaut. Cette valeur semble être arbitraire et ne suffit pas pour sonder correctement le réseau [1] [17], comme expliqué paragraphe suivant. Ces valeurs peuvent être paramétrées lors de l'appel à traceroute, mais l'algorithme lui-même n'est pas altérable. On peut voir figure 1 un exemple de découverte de chemin par traceroute.

1.1.2 Exemple de limites de traceroute

Traceroute présente des erreurs de mesure qui ont conduit à l'élaboration de nouveaux outils. Ces erreurs sont liées à la présence dans le réseau de load-balancers. Sur certains réseaux, on cherche à décongestionner le trafic, ou en assurer la pérennité, en doublant les chemins. Le load-balancer est le nœud qui se trouve à la bifurcation entrante et sera chargé de répartir le flux entrant.

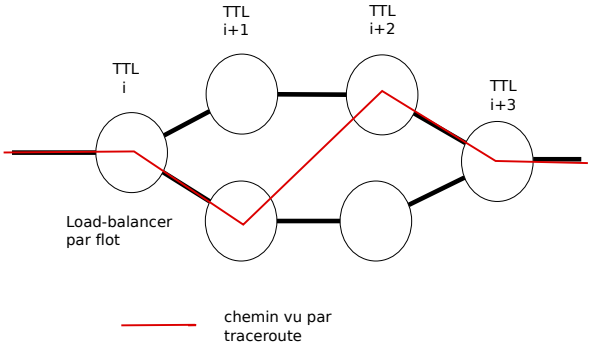

FIGURE 2 – erreur d'estimation du chemin à cause d'un load balancer

Selon le type de load-balancer, il se basera soit sur l'adresse IP destination, c'est donc un load-balancer par destination, soit sur un flot (adresse source, adresse destination, port source, port destination, protocole encapsulé par IP), c'est donc un load-balancer par application/flot, soit répartira aléatoirement, c'est donc un load-balancer par paquet. La plupart des load-balancers utilisent le flot ou la destination [2] [4].

Traceroute reconnaît les réponses à ses paquets grâce au port destination, et en change pour chaque probe/TTL. Cela induit donc que pour un load-balancer par flot, deux TTL différentes risquent de prendre deux chemins differents. Traceroute se trompe alors en inférant le lien à partir de ces deux probes. Un exemple d'erreur fait par traceroute est donné figure 2.

Pour le TTL i, les trois probes envoyées par traceroute vont bien atteindre le nœud visé. Traceroute va donc ensuite envoyer trois probes de TTL i+1 dont il va par exemple changer le port destination, pour chacune des trois. Ces probes vont être load-balancées par le nœud d'entrée, qui va les envoyer vers le chemin du bas. Au TTL i+2, traceroute va de nouveau changer les ports destination et envoyer les probes sur le réseau. Le load-balancer va cette fois-ci calculer que ce type de paquet doit être envoyé sur le chemin du haut. Le paquet va donc atteindre le deuxième nœud du chemin du haut et ce nœud va répondre à traceroute. En recevant cette réponse, traceroute va naturellement penser que le nœud se trouve directement à la suite du nœud trouvé au TTL i+1 et ne voir qu'un seul chemin qui de plus n'existe pas.

1.2 Les enjeux de la mesure de la topologie

La mesure de la topologie consiste simplement à sonder un réseau pour tenter d'en établir une carte. Pour cela, on établit une liste de moniteurs qui vont tenter d'établir la topologie en lançant une série de traceroute qui va permettre de réaliser un maillage complet.

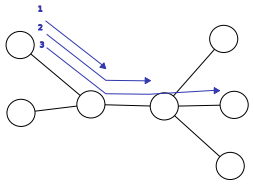

FIGURE 3 – traceroute : découverte d'un premier chemin

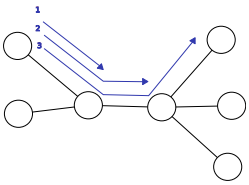

FIGURE 4 – traceroute : découverte d'un deuxième chemin et redondance

1.2.1 Paramètres et challenges de la mesure de topologie

La mesure de la topologie doit être faite à grande échelle si elle veut être le plus précis et exact possible [15] . On va chercher à développer des algorithmes ou heuristiques qui vont permettre de rendre l'exploration des chemins la plus efficace et la moins dérangeante possible pour le réseau. Les algorithmes pourront être mis en place sur des plateformes de mesure qui chercheront à sonder le réseau régulièrement. Leur efficacité doit être maximisée, ce qui conduit au developpement de nouveaux algorithmes de parcours des chemins dans un réseau. On va donc par exemple vouloir corriger les erreurs de load-balancing induites par traceroute. On veut aussi maximiser la complétude du réseau cartographié pour être au plus près du graphe réel. Il faut pour cela pouvoir examiner exhaustivement les voisins de chaque nœud. Un autre paramètre important à prendre en compte lorsque l'on mesure la topologie d'un réseau est que le réseau mesuré est dynamique. Cela signifie qu'il va changer, et possiblement pendant une mesure. On doit donc faire plusieurs sets de mesures pour pouvoir comparer et constater ces changements, et essayer d'avoir des mesures rapides pour que le réseau change le moins possible pendant la mesure. Les mesures ne doivent également pas empiéter sur le but premier du réseau qui est de transmettre des paquets contenant de l'information. Il ne faut donc pas surcharger le réseau par l'envoi d'un trop grand nombre de probes et donc éviter au maximum la redondance. Cela n'est pas le cas avec traceroute, qui va redécouvrir systématiquement le chemin en entier, y compris les nœuds déjà connus. On voit par exemple figure 3 la découverte d'un premier chemin par traceroute puis d'un deuxième figure 4. On voit que ce deuxième traceroute a conduit à redécouvrir tout le début du chemin une deuxème fois, ce qui est redondant et crée de la charge sur le réseau.

1.2.2 Mesure de la topologie au LIP6

Le LIP6 est le laboratoire d'informatique de Paris 6 (Jussieu). Il est sous la tutelle de l'université Pierre et Marie Curie et du CNRS.

OneLab [1] est un projet dont l'objectif est de réunir des ressources fournies par différents laboratoires d'Europe afin de fournir des testbeds à grande échelle pour expérimentation. PlanetLab [2] est un de ces testbeds. L'UPMC est en charge de gérer la coordination des réseaux mis en communs. Dans le cas de Planet-

1. http ://www.onelab.eu
2. http ://www.planet-lab.eu et http ://www.planet-lab.org

Lab, ces différentes ressources sont interconnectées via internet et la topologie résultant de leur mise en commun est inconnue. Il serait pourtant important de la connaître car elle peut influencer les résultats des expérimentations qui se dérouleraient sur ces réseaux, comme par exemple le déploiement d'application overlay ou sur du pair-à-pair. Connaître la topologie pourrait aussi expliquer certains résultats.

Dans ce cadre, l'UPMC a développpé pour OneLab un outil, TopHat, dont le but est de "photographier" régulièrement la topologie du réseau. Pour cela, l'outil a été installé sur les différents nœuds du réseau PlanetLab. À intervalles réguliers (5min), un snapshot du réseau est effectué grâce au lancement de nombreux paris-traceroutes. Mais cette technique est coûteuse en ressources (paquets envoyés) et présente beaucoup de redondance. Il serait donc intéressant de pouvoir utiliser des algorithmes améliorés de découverte de chemin afin de minimiser la charge sur le réseau, et de pouvoir ensuite mettre en oeuvre simplement ces algorithmes.

1.3 Une bibliothèque pour répondre à ces enjeux

On peut voir que l'on a d'un côté le programme traceroute qui n'est pas adapté à la mesure de topologie, d'une part à cause de son erreur sur les load-balancings et de l'autre à cause de son manque d'adaptabilité. On a de l'autre côté des mesures de topologie qui se veulent le plus léger, précis et complet possible, tout en étant evidemment aussi correctes que possible. Des propositions ont été faites pour améliorer les algorithmes de découverte de chemin, présentées dans la section 2, mais elles ne peuvent pas réutiliser traceroute et nécessitent toutes un système d'envoi de probe et association des réponses. Chacun des algorithmes proposés doit donc réécrire un système de gestion des probes afin de pouvoir être utilisé.

C'est pour répondre à ce problème que nous avons décidé d'écrire une bibliothèque en C qui reprendra le principe d'envoi de probe de traceroute tout en le rendant entièrement modulable. Cette bibliothèque sera organisée de façon à pouvoir paramétrer ou ajouter facilement de nouveaux éléments. Elle permettra un grand contrôle sur les paquets eux-mêmes ou à plus haut niveau sur le déroulement de l'algorithme. Il sera aisé d'ajouter un algorithme, et il n'y aura pas besoin de réécrire le principe d'envoi de probe pour cela. Nous avons spécifié puis réalisé cette bibliothèque, lui avons intégré les algorithmes paris-traceroute et MDA. Nous pouvons ensuite la déployer sur PlanetLab afin de réaliser une mesure de ce réseau et en particulier de caractériser les load-balancers qui s'y trouvent. Nous pourrons ensuite l'intégrer à TopHat pour faciliter les études futures sur le réseau.

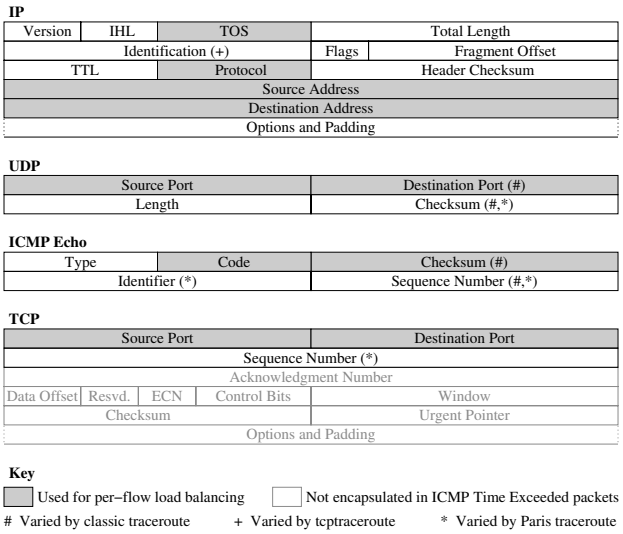

FIGURE 5 – champs retenus pour associer les probes selon traceroute ou paris-traceroute, source [1]

2 État de l'art

Divers algorithmes ont été proposés afin d'améliorer l'efficacité des découvertes de chemins dans un réseau. Tous reposent sur un principe d'envoi de probe pour découvrir des nœuds et des liens dans le graphe de la topologie, comme dans traceroute.

2.1 Correction des erreurs de load-balancing avec Paris-traceroute

Pour pallier les erreurs de load-balancing que génère traceroute décrites section 1, il a fallu repenser la façon dont étaient reconnues les probes et leurs réponses. On a pu voir que l'utilisation des ports destination pour reconnaître une probe n'est pas une bonne idée si l'on veut une topologie correcte du réseau. Or, seuls les 8 premiers octets du protocole inclus dans IP sont disponibles dans le paquet réponse que reçoit la source émettrice. Il fallait donc choisir un champ de l'en-tête IP ou du protocole encapsulé qui soit dans ces 8 octets pour pouvoir associer la réponse à une probe. C'est ce que fait paris-traceroute [1]. Il corrige les erreurs de load-balancing qui affectent traceroute en utilisant le même port pour des flots identiques. La reconnaissance de la probe à laquelle est associée une réponse est faite grâce à un autre champ du protocole encapsulé, par exemple "checksum" pour UDP et TCP. La figure 5 issue de [1] montre quels champs sont retenus par traceroute, tcptraceroute ou paris-traceroute pour identifier les réponses.

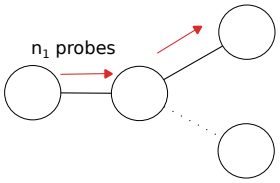

FIGURE 6 – MDA : initialement, un seul voisin connu

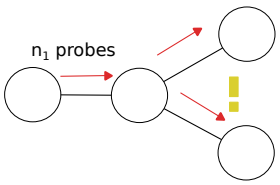

FIGURE 7 – MDA : un deuxième voisin apparaît

Les champs gris sont ceux qui affectent le load-balancing. les champs semi-transparents sont ceux qui ne sont pas transmis dans la réponse. Ce sont les champs marqués * qui sont retenus par paris-traceroute pour reconnaître les probes. On voit que ces champs n'entrent pas en compte pour le load-balancing, donc que les probes pourront avoir un flot constant. Traceroute n'étant pas modulaire, paris-traceroute a été entièrement codé, une première fois en C++ dans le cadre de la thèse de Brice Augustin, puis commencé en C.

2.2 Un algorithme exhaustif : MDA

Une autre amélioration a été proposée à partir de paris-traceroute : l'algorithme "multipath detection algorithm", MDA [3]. Le but de cet algorithme est de découvrir les interfaces de chaque nœud d'un graphe de manière exhaustive. Pour ce faire, on a estimé pour un nœud et un nombre de voisins connus, combien de probes étaient nécessaires pour trouver tous les voisins avec une probabilité d'échec *beta*. Cela signifie qu'il y a *beta* probabilité que l'on se soit trompé en rejetant l'hypothèse selon laquelle il y aurait encore un voisin supplémentaire à découvrir, c'est-à-dire que l'on ait raté au moins un voisin. Le principe est expliqué figures 6 à 8. Initialement, l'envoi de la première probe conduit à la découverte d'un premier voisin figure 6. Il est donc prévu d'envoyer n_1 probes pour s'assurer que c'est le seul voisin. L'une de ces probes conduit à la découverte d'un deuxième voisin comme illustré à la figure 7. Le nombre de probes est alors réajusté à n_2 figure 8. n_2 correspond aux nombre de probes à envoyer sachant que l'on a découvert deux voisins pour avoir une certaine probabilité de découvrir le troisième s'il existe. Le tableau figure 9 extrait de [3] montre pour une probabilité d'échec β le nombre de probes nécessaires à un nœud pour découvrir tous ses K voisins. Lorsqu'un nouveau voisin est découvert, les valeurs de probes à envoyer sont réajustées pour tenter de vérifier si ce voisin était le dernier que l'on ne connaissait pas, donc en visant le nombre de probes nécessaires pour en trouver un de plus avec la probabilité d'échec inférieure à β.

L'algorithme travaille indépendemment à chaque nœud en deux étapes. La

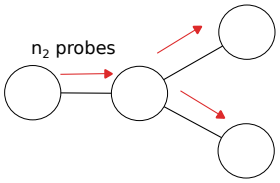

FIGURE 8 – MDA : réajustement du nombre de probes à envoyer

NUMBER OF PROBES TO SEND (k) VERSUS NUMBER OF EXPECTED INTERFACES (n), FOR A 95% DEGREE OF CONFIDENCE

n	2	3	4	5	6	7	8	9	10
k	6	11	16	21	27	33	38	44	51
n	11	12	13	14	15	16	17		
k	57	63	70	76	83	90	96		

FIGURE 9 – nombre de probes k nécessaires pour découvrir les n successeurs avec une probabilité d'échec inférieure à β, source [3]

première étape consiste à envoyer n_k probes pour découvrir les voisins, puis la deuxième étape consiste à envoyer un certain type de probe pour qualifier le load-balancer. La première version [3] de MDA reconnaissait les load-balancer par flot ou par paquet en faisant varier uniquement le port destination. Une deuxième version [2] tient compte du load-balancing par destination et cherche donc également à estimer si un load-balancer est de ce type. Pour la première étape, un certain nombre de probes est envoyé en faisant varier l'adresse destination suivant un certain masque (entre /29 et /24) afin de trouver tous les voisins d'un nœud. Pour la deuxième étape, on cherche d'abord à estimer s'il s'agit d'un load-balancer par paquet ou non. Pour vérifier si un load-balancer est de type load-balancer par paquet, six probes vont être envoyées avec un flot identique, et si elles passent par la même interface, on valide l'hypothèse selon laquelle il ne s'agit pas d'un load-balancer par paquet. Si ce n'est pas un load-balancer par paquet, on va ensuite envoyer six probes dont seul le port destination (ou source) change. Si les paquets subissent du load-balancing, alors il s'agit d'un load-balancer par flot. Sinon il s'agit d'un load-balancer par destination. une troisième amélioration [17] de MDA a été proposée afin d'assurer statistiquement la découverte de tous les chemins d'un trajet. Cet algorithme permet d'obtenir un compromis entre le nombre de probes générées et la complétude des chemins que l'on va pouvoir établir.

2.3 Autres outils de parcours de chemins

D'autres outils d'exploration ou de découverte de topologie ont été réalisés. Mercator [8] a été l'un des premiers. Tracenet [16] permet par exemple de connaître les autres interfaces des sous-réseaux visités lors de l'établissement du chemin emprunté. tcptraceroute est la version de traceroute utilisant TCP. tracert est la version Windows de traceroute, utilisant les ICMP "echo replies". Scapy [5] est un programme écrit en Python qui permet de générer et envoyer des paquets, ainsi que de recevoir, associer et analyser les réponses associées à

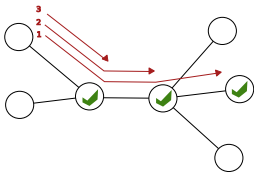

FIGURE 10 – tracetree : découverte d'un premier chemin

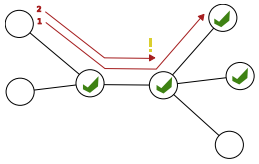

FIGURE 11 – tracetree : découverte d'un deuxième chemin et arrêt

ces paquets. Enfin, l'outil le plus proche de nos travaux est scamper [11], qui se présente comme nous sous forme de bibliothèque et est modularisé.

2.4 Des algorithmes d'optimisation des parcours de chemin

2.4.1 TraceTree

Une amélioration du principe de découverte de chemin a été proposée : Mercator [8], qui propose des heuristiques pour tenter d'explorer intelligemment le réseau. Cette amélioration a ensuite été reprise dans RADAR [10], sous le nom de TraceTree. Le constat de base était que traceroute redécouvrait systematiquement tout le trajet, alors que le début de celui-ci est en général constant, tel un arbre. L'idée a donc été de partir de la fin de l'arbre et de de baisser successivement le TTL, tout en mémorisant les nœuds rencontrés. Ainsi, une nouvelle mesure comparera à chaque itération le nœud rencontré et les nœuds présents dans sa table, et pourra donc s'arrêter lorsqu'elle rencontre un nœud déjà connu. La première mesure figure 10 s'effectue normalement (on notera l'ordre inversé par rapport à traceroute). Les nœuds cochés sont desormais connus. La deuxième mesure figure 11 va reconnaître le nœud et s'arrêter directement à la deuxième itération.

Cet algorithme optimise la mesure des chemins à partir d'une seule source, mais dans le cas où plusieurs sources effectuent les mesures, comme lorsque l'on souhaite mesurer une topologie, il y aura quand même de la redondance.

2.4.2 DoubleTree

Un algorithme a alors été proposé pour tenter de réduire la redondance à la fois du côté des sources et des destinations : Doubletree [7]. L'idée est ici de combiner deux arbres : un premier arbre d'un moniteur vers les sources où on voudra éviter la redondance, et un deuxième arbre qui correspond à la convergence des sources vers une destination commune. On devra donc établir un point de départ H des deux arbres qu'on essayera de placer optimalement sur le chemin. À partir de ce point seront déroulés les deux arbres. L'arbre remontant vers la source utilisera en quelque sorte le principe de tracetree : la source remontera progressivement le chemin en vérifiant à chaque itération qu'elle ne connaît pas le nouveau nœud rencontré. Si elle voit un nœud qu'elle

connaît, c'est qu'elle a déjà découvert le chemin jusqu'à ce nœud. Elle pourra donc s'arrêter.

Le second arbre part également de H et remontera jusqu'à la destination. Ici, les sources vont se partager les nœuds qu'elles rencontrent sous la forme de tuples (interface, destination). À chaque itération, le TTL sera augmenté de 1 comme pour un traceroute. Si le nouveau nœud découvert se trouve déjà dans la table partagée pour cette destination, c'est que le reste du chemin est déjà connu via un autre nœud. L'exploration pourra donc s'arrêter. Dans le cas contraire, l'exploration continue et le tuple (ip, destination) est partagé. La table contenant ces tuples pourra par exemple être une table de hashage distribuée, afin de réduire la taille prise par les données.

Par exemple, on voit d'abord dans la figure 12 le premier tour où S1 cherche à tracer le chemin jusque D1. Il sauve localement de S à H puis dans la table globale H-D1, H+1-D1 et ainsi de suite. Les encoches jaunes représentent les nœuds sauvés localement, et les vertes ceux sauvés et partagés. Lorsque S1 veut ensuite atteindre D2 comme sur la figure 13, il va regarder s'il y a H-D2 dans la table, et voyant que non, faire H+1-D2. Il regarde si H+1-D2 est dans la table et ainsi de suite jusqu'à finalement arriver a S2. On constate ici qu'il y a un peu de redondance dans la partie H-H+2. Lorsqu'il va vouloir remonter localement, il verra qu'il connaît déjà S1-H et s'arrêtera.

C'est au tour de S2 de chercher à établir le chemin jusque S1 comme sur la figure 14. Il va regarder dans la table et voir que H-D1 est déjà connu. Il peut donc s'arrêter là. Il cherche ensuite à remonter et va donc établir successivement le chemin jusque lui-même. S2 cherche ensuite à atteindre la nouvelle destination D3. Il doit donc tout redécouvrir car il n'y a aucune référence à D3 dans la table globale. Il découvrira donc ce chemin et l'ajoutera. Il connaît déjà d'autre part le trajet S2-H.

On voit ici que l'on a bien réduit la redondance, mais qu'il reste cependant encore des liens qui sont parcourus plusieurs fois inutilement.

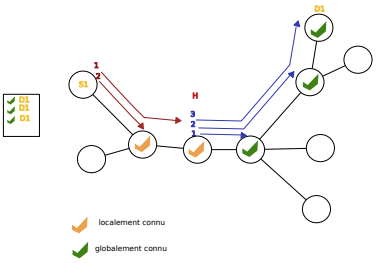

FIGURE 12 – doubletree : la source 1 trouve D1

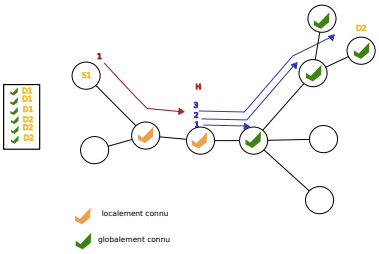

FIGURE 13 – doubletree : la source trouve D2 sans refaire S1-H

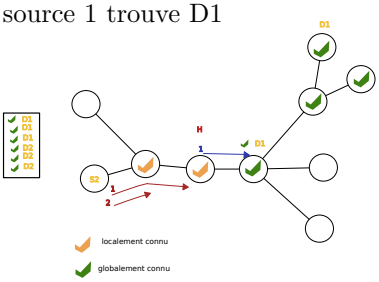

FIGURE 14 – doubletree : la source 2 trouve D1 sans refaire H-D1

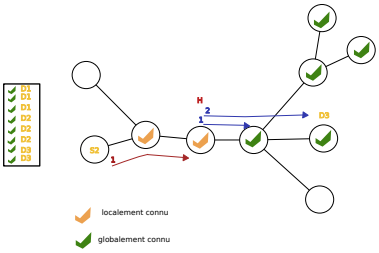

FIGURE 15 – doubletree : la source trouve D3 sans refaire S2-H

2.4.3 Limites de ces améliorations

Ces différents algorithmes ont permis de diminuer notablement la redondance lors de la recherche de chemin pour mesurer la topologie d'un réseau. Cependant on note que toute la redondance n'a pas disparu, particulièrement lors des "milieu de trajet" que traversent les moniteurs. L'idée générale pour arrêter la redondance serait de réussir à ne "découvrir" qu'une fois chaque association lien/nœud.

On remarque d'autre part que tous ces algorithmes ont une base commune qui consiste à envoyer une ou plusieurs probes à un nœud précis, dont on choisit certains paramètres. Il est dommage de réécrire entièrement une application lorsque l'on veut déployer un nouvel algorithme alors que le principe de l'envoi de probe suivant certains critères est chaque fois le même. C'est pour cela que notre bibliothèque s'inscrirait comme un outil pratique permettant de choisir à quel niveau on veut prendre la main sur ce que fait le programme, en décidant par exemple juste du contenu des probes ou au contraire en agissant au niveau des paquets. Les fonctions d'envoi et de réception seraient codées une bonne fois pour toutes et disponibles pour tout type d'action. Selon ce que voudrait l'utilisateur, la bibliothèque offre des fonctions de haut ou bas niveau. Notre bibliothèque intègre de plus nativement le principe Paris-traceroute d'association des probes, empêchant le problème lié aux load-balancers.

2.5 Autres outils de mesure de topologie

Il existe divers projets et programmes dont l'objectif est, tout comme TopHat, de réaliser des mesures de la topologie du réseau, tels que Archipelago [3]) ou DIMES [14]. Ces projets ont eux aussi choisi de réaliser leur propre programme pour mesurer la topologie du réseau. Iplane [13] (iplane.cs.washington.edu/) est également un outil de mesure des topologies. Certains outils comme DTrack [6] permettent de faire des prédictions afin de diminuer le trafic généré par les mesures. Il peut offrir un compromis entre l'utilisation brute de traceroute et la réécriture complète d'un outil qui utiliserait les améliorations proposées.

Ces outils utilisent généralement une serie d'appels à une implémentation de traceroute ou paris-traceroute, où il n'y a pas trop d'améliorations apportées si ce n'est sur le choix des destinations. Cela fait qu'un seul cycle de mesures dure longtemps (de l'ordre de plusieurs jours) avec une grande quantité d'informations redondantes. Il aurait été beaucoup plus efficace d'intégrer à ces outils un algorithme d'optimisation du parcours de chemins, diminuant ainsi à la fois le temps de parcours et la charge sur le réseau. Cette intégration est cependant rendue difficile par le codage souvent monolithique de ces outils.

3. http ://www.caida.org/projects/ark/

3 Notre objectif : concevoir une bibliothèque pour simplifier le déploiement de ces algorithmes

3.1 Conception de la bibliothèque

3.1.1 Objectifs et caractéristiques

Nous avons décidé de réaliser une bibliothèque de façon modulaire afin de faciliter son utilisation et son amélioration. Son utilisation sera simplifiée par une forte granularité, permettant à l'utilisateur de se placer au niveau le plus pratique pour ce qu'il souhaite faire. Son amélioration est facilitée car les éléments "ajoutables" sont en quelque sorte dotés d'une interface (au sens programmation objet du terme), c'est-à-dire qu'ils sont tous basés sur un modèle de données et fonctions à remplir. L'utilisateur remplit ce modèle comme il le souhaite et pourra l'inclure très simplement dans la bibliothèque. Nous avons modifié la modularisation telle qu'elle avait été proposée par la première ébauche de paris-traceroute pour la rendre la plus grande possible. Nous avons par exemple décidé que la sous partie créant les paquets devait être indépendante du reste de la bibliothèque. Il sera donc possible de réutiliser cette partie de la bibliothèque pour créer ses propres paquets indépendemment de traceroute. De même, les parties "ajoutables" telles que celle gérant l'algorithme sont indépendantes afin de pouvoir simplement ajouter la gestion d'un nouvel algorithme, sans être "conscient" de ce qui se passe dans les sous-couches. Cette modularité permet de conserver l'aspect abstrait des algorithmes sans être inquiété par les détails d'implémentation. La figure 16 schématise la modularité de la bibliothèque et montre les différentes granularités disponibles à l'utilisateur.

On veut aussi la rendre indépendante du fait qu'elle est écrite en C pour les appels extérieurs. On écrit par exemple un binding Python pour pouvoir utiliser la bibliothèque facilement sur TopHat. On veut également la rendre indépendante du système d'exploitation, ce qui est le cas pour la plupart des modules qui la composent, excepté la socket d'envoi et les structure des en-têtes.

Elle rend aussi les algorithmes parallélisables. Connaissant elle-même tous les algorithmes et se trouvant "au-dessus d'eux", elle peut simplement gérer une liste d'algorithmes qu'elle va lancer en parallèle. L'intérêt ici est de gagner du temps et d'optimiser l'utilisation de la bande passante et de la socket. Cela permet de plus de faire du multiplexage. Ces paramètres sont importants et l'on doit en tenir compte si on veut effectuer des mesures à très grande échelle tout en cherchant à être le moins intrusif possible.

3.1.2 Couche bas-niveau : gestion des paquets

Nous avons naturellement commencé par développer les fonctions bas-niveau de notre bibliothèque. Pour la réalisation des paquets, nous avons décidé d'utiliser les structures présentes dans netinet/, c'est-à-dire les structures C représentant tous les en-têtes des principaux protocoles telles que connues par défaut pour Linux. On peut se détacher simplement du fait que ce soit Linux en reprenant la structure au début de notre fichier. Il fallait que l'appel depuis l'extérieur

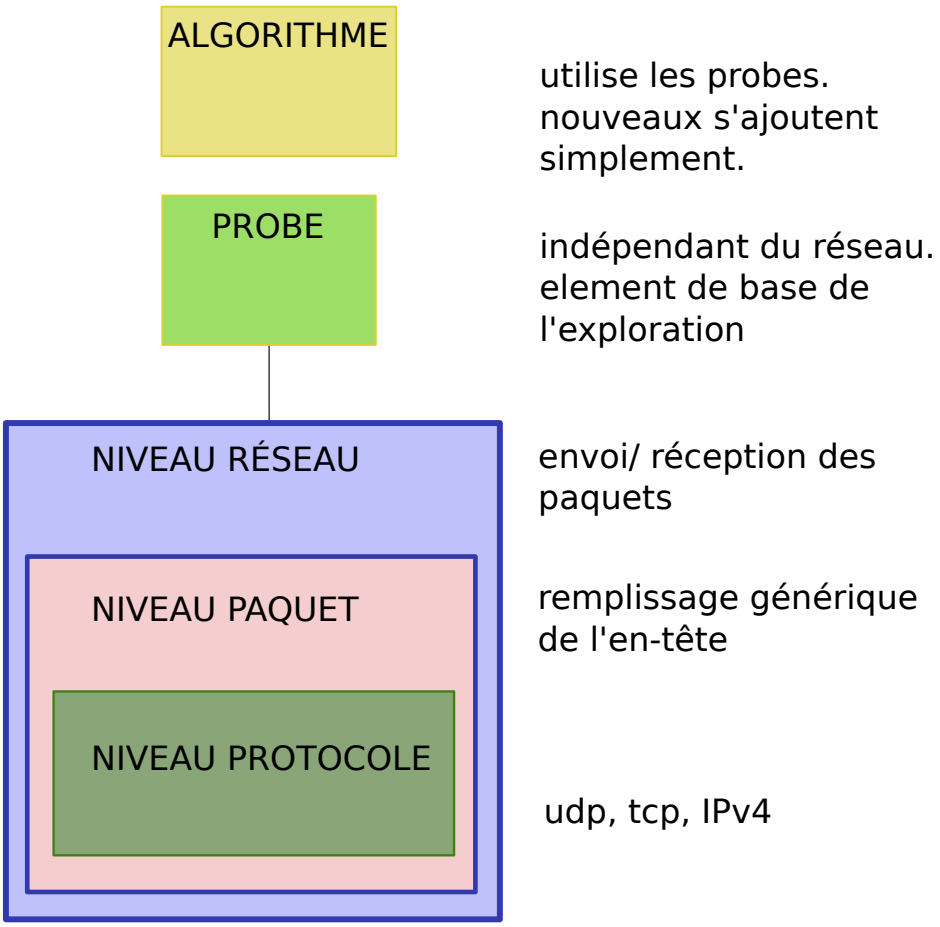

FIGURE 16 – Une bibliothèque modulaire

soit indépendant des protocoles choisis afin de rendre ceux-ci "ajoutables", c'est pourquoi il y a présence d'une partie protocole, qui sera connue via des fonctions génériques par le générateur de paquet. Cette généricité permettrait alors la totale liberté de choix des protocoles par l'utilisateur (par exemple IP sur IP s'il le souhaite). Nous avons aussi pris en compte l'interdépendance de certains protocoles, tel qu'IP qui nécessite de connaître le protocole encapsulé, ou les checksum UDP et tTCP qui nécessitent un pseudo-header de IP, en proposant des fonctions qui permettent à "paquet" de savoir qu'il doit les faire intéragir, sans pour autant perdre en généricité. Par exemple, faire une fonction de création d'un pseudo en-tête et une fonction de création d'en-tête qui accepte les pseudo en-têtes. Paquet appelle donc la fonction de pseudo en-tête si un niveau annonce en avoir besoin et ne fait rien sinon, sans avoir à connaître le protocole. L'idée ici était de fournir un équivalent de scapy, mais en C.

3.1.3 Couche intermédiaire : gestion des probes et du réseau

Au niveau intermédiaire, on veut créer des paquets simplement en leur passant les paramètres de notre choix puis pouvoir les envoyer tout aussi simplement sur le réseau. On veut également pouvoir recevoir la réponse le cas échéant. On va donc mettre en place une structure de champs qui contiendra génériquement les paramètres que l'on souhaite pour le paquet, et celui-ci se charge de les transmettre aux protocoles. On dispose aussi d'une fonction d'envoi des paquets et la possibilité de mettre en place une fonction de gestion des réponses.

On a créé à côté de cela une structure de probe qui permettra de simplifier la création des paquets et la gestion de leur envoi et réponse. Ainsi l'utilisateur peut se contenter de créer une probe, puis lui assigner les paramètres de son choix et l'envoyer. La partie "réseau" de la bibliothèque s'occupera alors d'envoyer le paquet correspondant à cette probe, puis à attendre la réponse via un sniffer et l'associer à la probe. L'utilisateur peut ensuite récupérer ces données et en faire ce qu'il souhaite.

3.1.4 Gestion haut-niveau des algorithmes et de leur appel

À haut niveau, l'utilisateur pourra se contenter de demander à la bibliothèque de dérouler un certain algorithme parmi ceux qu'elle connaît en lui spécifiant certains paramètres, ou écrire lui même son propre algorithme. L'utilisation sera simplifiée au maximum par la mise à disposition de fonctions de base pour envoyer les probes.

D'autre part, un module de mise en forme de la sortie de l'information a également été décidé, afin de pouvoir choisir simplement et à sa convenance l'affichage des résultats. De la même façon que pour algorithme et protocole, l'ajout sera possible par l'utilisateur de son propre module de formatage de la sortie.

La bibliothèque telle que vue de l'extérieur consistera donc au plus haut niveau à spécifier une série d'algorithmes à lancer avec certains paramètres et elle se chargera de tout orchestrer. Ces algorithmes pourront ainsi être parallélisés, ce qui présente un intéret pour l'optimisation du temps nécessaire à la mesure.

3.2 Implémentation et tests

3.2.1 Architecture et choix d'implémentation

Afin de suivre nos spécifications, nous avons modularisé le code comme montré sur la figure 17. Elle se compose des modules suivants :

Le module *Proto* expose une liste chainée offrant les fonctions génériques pour tous les protocoles qui ont été ajoutés. Il offre également des fonctions qui permettent d'appeler la fonction d'un algorithme en le passant en paramètre. Par exemple, la fonction qui va servir a l'extérieur à deviner si un en-tête est bien d'un certain protocole sera $guess\_protocol(char * protocol, char * *data)$, qui dans *Proto* trouvera la structure associée au protocole puis appellera la fonction *guess* de cette structure. Ainsi, *Packet* peut utiliser les fonctions de *Proto* sans avoir à connaître vraiment le protocole.

Le module *Packet* remplit une structure paquet à partir de protocoles spécifiés en entrée et à partir d'une liste de champs. C'est lui qui appelera les protocoles un par un puis calculera les checksums.

Network est le module principal de la bibliothèque. C'est lui qui va gérer à la fois l'appel aux algorithmes, la création des paquets à partir des probes et les envois sur le réseau. Il gère aussi la validation des probes restées sans réponse.

Le module $Sniffer$ est le module qui écoutera les réponses du réseau puis appelera la fonction d'association réponse-probe.

Le module *Queue* est comme son nom l'indique une queue pour l'envoi des paquets sur le réseau. On pourra y mettre des conditions telles qu'un débit ou un nombre maximum de paquets envoyés par seconde.

Algo est le module qui permet de gérer et lister les algorithmes connus par la bibliothèque. De la même façon que *Proto*, il expose une liste d'algorithmes qui pourront être choisis depuis l'extérieur. *Algo* propose aussi les fonctions de base pour explorer un nœud ou un chemin via *node_query* qui envoie une probe avec un certain flot et un certain TTL et *link_query* qui envoie deux probes avec le même flot afin d'être sûr que le lien existe. Ces fonctions permettent d'un côté de simplifier l'écriture des algorithmes qui pourront se contenter d'appeler *link_query*, et permettra de l'autre aux personnes qui souhaitent simplement envoyer une probe sans dérouler tout un algorithme de le faire.

Output est le module d'affichage. Il prend en paramètre une ou plusieurs probes et réalise l'action d'affichage prévue, qui peut être l'écriture dans un fichier ou l'affichage dans la sortie standard de l'IP réponse. Il se base sur le même principe de liste que *Proto* et *Algo*.

Probe est le module qui gère les probes. Il permet de créer une probe puis de lui associer les données de réponses. Cela permet de rendre abstrait le principe d'envoi de paquet sur le réseau et de paquet réponse.

Enfin, le module *Instance* permettra de transformer l'appel extérieur et appeler les fonctions adéquates. Il interprétera entre autres les options et la création de la liste d'algorithmes à lancer.

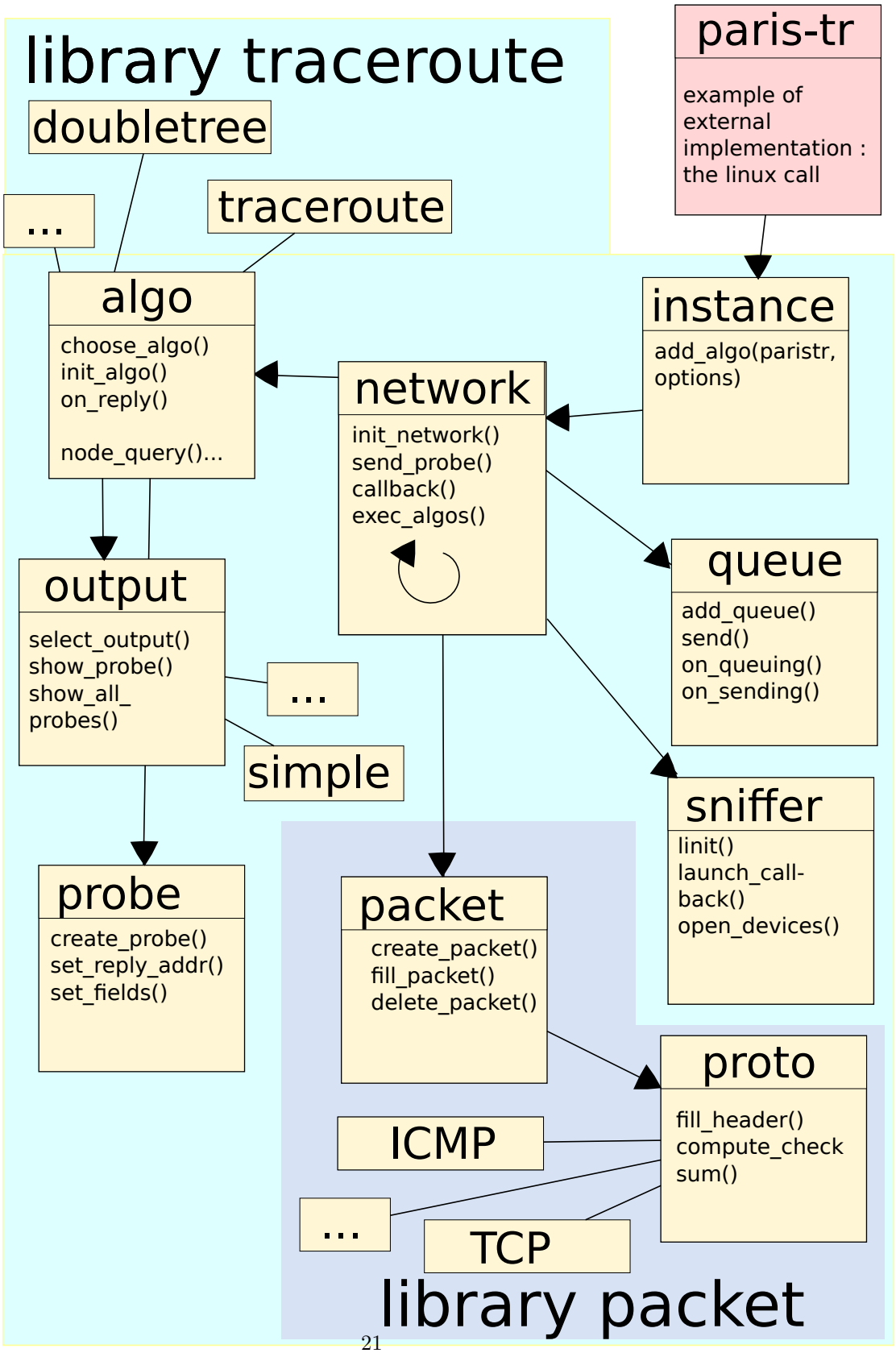

FIGURE 17 – Schéma de l'implémentation de la bibliothèque

Notre code est en C. On génère à partir du code une bibliothèque dynamique qui peut être utilisée pour forger des paquets ou appeler des algorithmes. Notre code est pour le moment compatible avec Linux uniquement mais l'ajout d'une couche reliant les paramètres Mac/Windows à ceux Linux devrait suffire. Nous avons fait attention à rendre le plus portable possible ce que nous avons écrit.

L'implémentation s'est faite du bas niveau vers le haut niveau, avec pour objectif d'obtenir d'abord une version très basique mais fonctionnelle. Nous avons implémenté les protocoles IPv4, UDP et en partie ICMP. Le choix des protocoles pour IP et ICMP se justifie par d'un coté la domination d'IPv4 dans les réseaux actuellement, et la nécessité de connaître ICMP pour traiter les réponses aux TTL expirés de traceroute. Le choix d'UDP quant à lui s'explique par sa simplicité d'une part relativement à TCP, et d'autre part le fait que les load-balancers appliquent principalement leur répartition sur UDP et TCP, et beaucoup moins sur ICMP [4]. TCP aurait donc également correspondu, mais la variation de port nécessaire à l'algorithme, utilisée via TCP, ressemble beaucoup à un scan de port ou réseau et peut provoquer l'absence de réponse du nœud scanné si celui-ci a une politique de sécurité forte. De plus, TCP est plus compliqué à implémenter.

Au niveau du réseau, nous avons décidé de n'implémenter pour les envois qu'une socket de type "raw" pour le moment. Nous envisageons de mettre à disposition dans *Proto* des sockets correspondant au type de protocole qui les crée. Du côté de l'écoute réseau, nous utilisons libpcap. Libpcap [4] est une bibliothèque permettant l'écoute du réseau avec apposition de filtre et appel de callback sur les paquets reçus. Une fonction de callback par défaut des réponses à été codée.

Au niveau des probes et algorithmes, nous avons développé un algorithme d'envoi d'une seule série de probes à un TTL donné, que nous avons appelé "simplehop", et l'algorithme MDA, que nous utilisons par la suite pour effectuer les mesures sur PlanetLab. Nous avons réalisé un module de sortie qui ressemble à celui de traceroute et qui affiche sur la sortie standard.

Pour le manuel qui doit être fourni avec notre bibliothèque, nous avons dans un premier temps commenté tous les prototypes des fonctions de manière à ce que l'on puisse extraire une documentation de type doxygen [5], par exemple en utilisant doxywizard ou en ligne de commande avec doxygen lui-même. Il est prévu de réaliser ultérieurement une page de manuel pour ajouter au système standard "man" de Linux.

3.2.2 Exemple de déploiement d'un algorithme

Pour mieux comprendre le fonctionnement de notre bibliothèque, nous allons observer son fonctionnement via le déroulement de l'algorithme traceroute tel que nous le connaissons. Pour simplifier la présentation, nous avons négligé certains aspects tels que "retenir le nombre de "*" pour vérifier qu'il y en a trois" ou "ajouter la probe en mémoire". En pseudo-code, l'algorithme se déroule de la façon suivante :

4. http ://www.tcpdump.org/
5. http ://www.stack.nl/ dimitri/doxygen/index.html

```
1  1/ initialisation
2  create_fill_infos_perso();
3  for(nb_probe_per_TTL){
4    node_query(destination_addr,destination_port, TTL=1);
5  }

6  2/ on_reply
7 probe_responded[TTL]= probe_responded[TTL]+1
8   if probe_responded[TTL]==nb_probe_per_TTL
9       if (reply=="*" or reply==dest_IP)
10         show_output(probe_list);
11         free_infos;//end of algo
12         exit();
13      else
14          TTL=TTL+1
15          for(nb_probe_per_TTL)
16               node_query(destination_addr,destination_port,TTL);
17  else
18      //wait next anwser
```

Deux fonctions principales composent l'algorithme : la fonction d'initialisation *init* et la fonction *on_reply*. ce sont ces fonctions que devra écrire l'utilisateur s'il souhaite ajouter son algorithme à notre bibliothèque. Pour dire à la bibliothèque lancer notre algorithme, on ajoutera une instance de celui-ci avec certains paramètres à la liste des algorithmes prévus. Elle va donc aller chercher la fonction *init* de notre bibliothèque et l'appeler. Dans notre cas, la fonction init va instancier une structure personnelle de données (ligne 2) que l'on considère nécessaires pour l'algorithme, comme par exemple le nombre de probes ayant déjà obtenu une réponse et la liste des probes. Elle va ensuite demander (lignes 3-4) $nombre\_de\_probe\_par\_TTL$ fois l'envoi d'une probe à un certain TTL grâce à la fonction disponible pour tous les algorithmes *node_query*. *node_query* permet d'envoyer une probe avec un certain TTL à une certaine destination, sans que l'utilisateur n'ait à gérer cette probe lui même. la fonction *init* peut ensuite rendre la main à la bibliothèque qui va pouvoir continuer à lancer les *init* de tous les algorithmes dont une instance a été prévue. Une fois ceci fait, la bibliothèque va, via $Network$, entrer dans un processus infini où elle va s'occuper de gérer les probes sans réponses. elle regarde la probe la plus vieille et compte un certain temps *timeout*. Si au bout de ce temps la probe n'a toujours pas de réponse, elle lui en ajoute une de type "*", la classe comme "ayant obtenu une réponse" et appelle la fonction *on_reply*.

Ces probes envoyées vont soit générer des réponses qui seront captées par le module chargé d'écouter le réseau, soit être traitées par network. Au recevoir d'une réponse de probe, le sniffer appelle la fonction de traitement, le callback. Cette fonction peut être codée par l'utilisateur, mais un callback par défaut a été codé par nos soins. Ce callback aura pour mission de vérifier de quel type de paquet il s'agit afin de savoir s'il s'agit bien d'une réponse à l'une de nos probes. Il se charge ensuite de retrouver de quelle probe il s'agit en suivant le principe de paris-traceroute de comparaison des checksums UDP. Ayant retrouvé la probe, le

callback pourra appeler la fonction de l'algorithme *on_reply* grâce à un pointeur vers la fonction conservée dans la probe.

La fonction *on_reply* de l'algorithme attend ici d'avoir obtenu les réponses à toutes les probes qu'il a envoyées pour le TTL en cours. En effet, comme dans traceroute, plusieurs probes sont envoyées pour un même TTL. L'algorithme commence donc par incrémenter son compteur de probes complètes (ligne 7). Si ce n'est pas la dernière réponse, notre algorithme ne fait rien (ligne 17-18). Si par contre il s'agit de la dernière réponse (ligne 8), notre algorithme va regarder si c'est une réponse de type "*" ou de type "ip_dest". Si oui, l'algorithme considère que le réseau ne répond pas ou que les envois sont terminés, affiche ses probes en mémoire et s'arrête. Sinon, il envoie *nombre_de_probe_par_TTL* fois une probe via *node_query* en augmentant le TTL de 1. La fonction d'affichage du résultat pour une liste de probe dépend de ce qu'on a associé à l'algorithme. On va ainsi obtenir avec notre sortie type traceroute, pour trois probes, pour chaque TTL :

```
targeting 1.1.1.1:32500 ; TTL 2, reply IP : 10.0.0.2 in 0.000264 sec
targeting 1.1.1.1:32500 ; TTL 2, reply IP : 10.0.0.2 in 0.000214 sec
targeting 1.1.1.1:32500 ; TTL 2, reply IP : 10.0.0.2 in 0.000165 sec
```

À plus bas niveau, chaque probe a engendré la création d'un paquet avec récupération des paramètres, mais tout ceci est transparent pour l'utilisateur.

On voit donc qu'il est simple de coder un algorithme et que des fonctions haut niveau sont fournies pour simplifier le processus. L'utilisateur n'a qu'à remplir les fonctions prévues. De la même façon, l'ajout d'un affichage se fera en remplissant les fonctions view_probe et view_all_probes. L'ajout d'un protocole nécessite de remplir environ une quinzaine de paramètres mais suit exactement le même principe et est donc tout aussi simple à mettre en place.

3.2.3 Tests de notre bibliothèque

Nous devons ensuite réaliser différents tests afin de vérifier le bon fonctionnement de notre bibliothèque mais également sa robustesse. Nous avons réalisé d'une part des tests unitaires sur les différentes parties de notre bibliothèque, et d'autre part un test de fonctionnement général en simulant un réseau grâce à un programme Python réalisé par Jordan Augé, *fakeroute*. *Fakeroute* va intercepter les probes à destination d'une adresse convenue d'avance, ici 1.1.1.1 et répondre en fonction des données du paquet envoyé. On peut ainsi simuler la présence d'un load balancer ou de n'importe quelle autre topologie particulière que l'on souhaite soumettre à notre bibliothèque. On peut aussi utiliser ce programme pour envoyer des données malformées ou fausses et voir comment réagit notre bibliothèque.

4 Application : mesure du déploiement des load-balancers au sein du réseau PlanetLab, grâce à notre bibliothèque

On va pouvoir utiliser notre bibliothèque pour faire une estimation de la topologie des ressources mises à disposition dans PlanetLab via l'algorithme MDA. Cependant, ces travaux sont encore en cours puisqu'il a d'abord fallu finir de développer la bibliothèque. Ceci explique le faible nombre de résultats présentés. Une version étendue de ces résultats sera présentée lors de la soutenance.

4.1 Objectifs

L'objectif de cette étude est double. D'un côté c'est un moyen de tester notre bibliothèque en conditions réelles, et de l'autre fournir une étude sur les load-balancers présents sur PlanetLab et et les caractériser. Pour les métriques retenues, cette étude s'inspire de [4]. La contribution de cette analyse est de fournir une étude beaucoup plus complète que ce qui avait été fait précédemment sur les load-balancers de PlanetLab, avec un nombre de sources et destinations beaucoup plus important. Cette étude va dans un premier temps localiser les load-balancers qui se trouvent sur PlanetLab (caractériser leur distance, leur AS, leur type de load-balancing) puis établir des caractéristiques sur leur taille, leur asymétrie et leur largeur. L'objectif à plus long terme est ensuite d'intégrer notre bibliothèque à TopHat, afin de lancer régulièrement ces mesures de load-balancing au sein du réseau et pouvoir les fournir aux utilisateurs de ce réseau.

4.2 Caractérisation des données du réseau analysé

Comme dit précédemment, l'algorithme utilisé pour parcourir le réseau sera l'algorithme MDA. Il sera utilisé entre 450 sources (vues comme actives au début de la mesure) et 998 destinations. On a cherché à identifier ces différents moniteurs. Pour cela, on a eu recours à diverses sources de données. On veut pouvoir caractériser les IP de nos load-balancers, on va donc utiliser divers outils permettant à partir d'une IP d'obtenir diverses informations. Ces outils sont :

- **TeamCymru IP-to-ASN mapping service** permet de récupérer des données sur l'AS auquel appartient une certaine IP.
  URL : http ://www.team-cymru.org/Services/ip-to-asn.html
- **MaxMind** permet de connaître des informations sur une IP telles que le pays d'origine, la région ou encore les coordonées.
  URL : http ://www.maxmind.com/app/geolitecity
- **Autonomous System Taxonomy Repository** classe les AS selon une liste bien définie, comme par exemple "grand ISP" ou "université"
  URL : http ://www.ece.gatech.edu/research/labs/MANIACS/as_taxonomy/

En plus des mesures obtenues par ses agents déployés sur les nœuds PlanetLab, le système TopHat propose les mesures et informations provenant de plusieurs autres systèmes de mesures ou d'informations partenaires interconnectés. C'est notamment le cas du service de TeamCymru, de MaxMind ou du jeu de données MANIACS. TopHat propose une API XMLRPC qu'il est possible

d'interroger pour obtenir ce genre d'informations d'une manière transparente et consistente, indépendemment du format de données de chaque service. Il est de plus possible d'utiliser un cache en interne pour accélérer les réponses.

On peut alors en déduire (grâce à TopHat)

- l'AS via team Cymru ip to asn mapping online tool
- type d'AS (Large ISP tier-1, Small ISP tier-2, university,...)
- pays via MaxMind geolocalization database
- continent

4.3 Méthodologie

Actuellement MDA ne fournit à l'utilisateur que l'ensemble des interfaces existantes à chaque hop, et pour chacune, l'ensemble des identifiants de probe les ayant traversées. Nous avons réalisé des scripts Python permettant le traitement de ces données (qui pourraient eux aussi être intégrés à la bibliothèque à terme) afin d'en déduire des informations sur les load-balancers.

4.3.1 Récupération des explorations exhaustives avec l'algorithme MDA

La première étape de notre étude a consisté à récupérer l'ensemble des mesures réalisées sur le réseau. Nous avons en grande partie réutilisé les fonctionnalités proposées par TopHat. TopHat est écrit de façon modulaire en Python, ce qui lui permet d'intégrer divers outils en réalisant simplement un wrapper Python. Il est également capable d'envoyer et recevoir des fichiers sur un serveur. Les mesures sont effectuées par des modules qui tournent en parallèle sur la machine (par exemple, un ensemble de traceroute simple effectué très régulièrement entre toutes les paires de nœuds PlanetLab). Nous avons écrit un module spécifique permettant de faire la campagne de mesure nécessaire à notre expérimentation. Nous avons également réutilisé l'ensemble des nœuds PlanetLab comme ensemble de destination (mis à disposition par l'agent TopHat après téléchargement auprès du serveur).

Le déploiement sur PlanetLab s'est effectué en créant une machine virtuelle Linux (Fedora) accessible en root sur l'ensemble des nœuds de PlanetLab. On a ensuite installé l'outil et les dépendances de manière automatisée grâce à des scripts d'installation. La prise des mesures a mis environ 5h pour les 425 sources et 998 destinations.

4.3.2 Reconstitution des liens à partir des informations de probe

La seconde étape était de récupérer les données précédemment générées afin d'en extraire les load-balancers et les caractéristiques de ces load-balancers qui nous intéressent pour nos analyses. On va donc parser l'ensemble des résultats obtenus par paris-traceroute.

Nous avons pour cela créé un premier script de nettoyage qui retire les données qui ne nous sont pas utiles dans la sortie de paris-traceroute. Les données qui nous intéressent ici sont de repérer quelles IP représentent des load-balancers, et calculer les métriques associées. Pour cela, on doit réussir à partir de ces données à retrouver les chemins de "sortie" de ces load-balancers.

Les load-balancers dans paris-traceroute sont signalés par la présence d'un signe "=" après l'IP s'il s'agit d'un load-balancer par flot, et d'un "<" s'il s'agit d'un load-balancer par paquet. Pour chaque IP du réseau, paris-traceroute lui associe soit rien si cette IP reçoit tous les flots ou dans le cas de load-balancing par packet, soit la liste des probes qui passent explicitement par ce paquet. Si un paquet est un load-balancer, on peut donc reconstituer le chemin sortant de ce nœud en suivant les probes qui le caractérisent. C'est ce que l'on a fait pour découvrir quels sont les vrais liens mesurés par paris-traceroute.

4.3.3 Reconstitution des load-balancers et calcul des métriques

La troisième étape, maintenant que l'on sait récupérer les chemins qui constituent la sortie du load-balancer, va consister à extraire des captures paris-traceroute toutes les informations que l'on souhaite sur le load-balancer. On va pour cela écrire un script Python qui réalisera à la fois l'étape deux et la reconstitution des métriques pour les load-balancers. Le script est écrit de sorte à balayer successivement les lignes de sortie de paris-traceroute pour en extraire les load-balancers. Lorsqu'un load-balancer est trouvé, on récupère directement certaines données comme l'IP du load-balancer, l'IP de la source ou l'IP de la destination. On va ensuite effectuer quelques calculs pour retrouver diverses métriques en s'inspirant de celles choisies par [4]. Finalement, les données retenues pour chaque load-balancer sont écrites sur une ligne et ajoutées à la fin d'un fichier de résultats. On peut ainsi remarquer que pour notre analyse, 624 153 apparitions de load-balancers ont eu lieu, correspondant à 5456 interfaces uniques. Il s'agit uniquement des interfaces et non des routeurs, qui n'entrent pas dans le cadre de l'étude et nécessiteraient des techniques d'associations des interfaces d'un routeur.

4.4 Métriques

Il est important de noter qu'ici la définition de diamant est "load-balancer ouvrant puis un certain nombre de hop puis un nœud fermant tel que toutes les probes passent par cette structure". Un load balancer est donc une ouverture de diamant s'il gère l'ensemble des données depuis la source. Les métriques retenues sont donc les suivantes, par load-balancer :

- nombre de load-balancers pour ce parcours source-destination
- IP du load-balancer
- IP de la source
- IP de la destination
- TTL du load-balancer
- TTL de la destination
- nombre d'interfaces de sortie de ce load-balancer

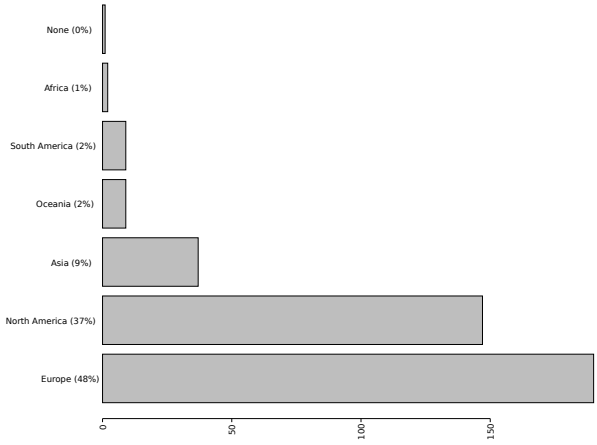

FIGURE 18 – Répartition des sources par continent

- le type de load-balancer (flot, destination, unknown)
- si c'est un diamant
- nombre total de chemins purement disjoints dans ce load-balancer
- nombre maximum de chemins possibles
- longueur maximum d'un chemin dans ce load-balancer
- largeur de l'asymétrie, i. e. écart de valeur entre le chemin le plus long et le plus court

4.5 Caractéristiques des données

4.5.1 Caractéristiques des sources

Les sources sont réparties sur 38 pays, mais principalements les États-Unis. Les répartitions sont principalement :

Pays	nombre de sources
"United States"	137
"Germany"	34
"France"	21
"Hungary"	16
"Italy"	15

Ce qui signifie qu'il y a 137 sources situées aux États-Unis, puis 34 en Allemagne et ainsi de suite. Ces sources proviennent des 7 continents, comme le montre la figure 18.

Les principaux types d'AS rencontrés sont des small ISP (tier-2), et correspondent principalement à des réseaux de recherche. Principaux AS rencontrés :

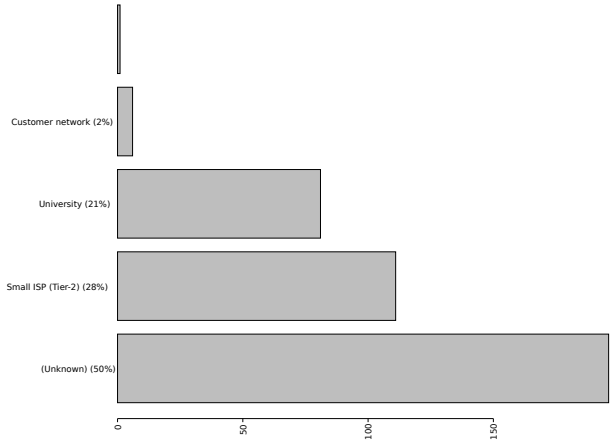

FIGURE 19 – Répartition des sources par type d'AS

numéro d'AS	nombre de sources
"DFN-IP service G-WiN"	28
"ASGARR GARR Italian academic and research network"	15
"HBONE-AS HUNGARNET"	14
"RCCN Rede Ciencia Tecnologia e Sociedade (RCTS)"	12
"JANET The JANET IP Service"	11
"FR-RENATER Reseau National de telecommunications pour la Technologie"	9
"SWITCH SWITCH, Swiss Education and Research Network"	8
"REDIRIS RedIRIS Autonomous System"	7
"ERX-CERNET-BKB China Education and Research Network Center"	6
"BELNET AS for BELNET, The Belgian National Research and Education Network"	5

Ce qui signifie que 28 des sources se trouvent dans l'AS nommé DFN-IP et ainsi de suite.

Les types d'AS sont répartis comme montré sur la figure 19

4.5.2 Caractéristiques des destinations

Les destinations sont quasiment tous les nœuds du réseau PlanetLab. On obtient globalement les mêmes résultats que pour les sources, si ce n'est plus de destinations aux États-Unis.

4.5.3 Caractérisation du réseau

Pendant le parcours du réseau, 20770 IPs uniques ont été rencontrées. Certaines de ces IP sont vues énormément de fois, ce qui montre qu'il y a de la redondance.

IP	nombre d'apparition dans nos traces
64.57.28.112	25002
64.57.28.96	25217
62.40.124.33	26635
62.40.112.9	27686
62.40.112.10	28815
64.57.28.6	30148
62.40.112.161	34083
62.40.125.18	43319

Cette redondance est principalement dans 3 réseaux (on prend les 100 IPs les plus rencontrées). Il serait souhaitable de pouvoir la minimiser car paris-traceroute utilise beaucoup de paquets pour établir la topologie.

AS	nombre d'apparitions
"GEANT The GEANT IP Service"	41
"ABILENE - Internet2"	13
"NLR - National LambdaRail"	10

En tout, 55 pays et 509 AS uniques sont traversés, de tous les types, avec en majorité des tier-1 et de tier-2 ce qui n'est pas surprenant étant donné que ce sont des AS de cœur et que nos nœuds sont très répartis à travers le monde.

type d'AS	nombre d'apparitions
"Large ISP (Tier-1)"	10319
"Small ISP (Tier-2)"	5558
"(Unknown)"	3852
"University"	1278
"Customer network"	131
"Internet Exchange Point (IXP)"	51
"Network Information Center"	14

4.5.4 Caractérisation des load-balancers

Notre liste des load-balancings vus possède 624 153 entrées, mais si l'on extrait les IP de ces load-balancers on obtient 5456 adresses différentes, soit 5456 interfaces de load-balancers différentes. L'analyse TopHat nous donne les données suivantes sur ces load-balancers :

Concernant le type d'AS :

type d'AS	nombre d'apparitions
"Large ISP (Tier-1)"	4013
"Small ISP (Tier-2)"	1037
"(Unknown)"	380
"University"	73
"Customer network"	11
"Internet Exchange Point (IXP)"	5
"Network Information Center"	2

On a donc quasiment 4/5 des load-balancers qui se trouvent dans les Larges ISP. Cela est cohérent puisque ce sont ces réseaux qui vont gérer d'enormes

trafics et ont donc tout intérêt à user des load-balancers.

Concernant la provenance :

Continent	nombre d'apparitions
"North America"	3582
"Europe"	1434
"Asia"	456
"South America"	29
"Oceania"	11
"None"	8
"Africa"	1

On constate une grande répresentation de l'amérique du nord dans les load-balancers.

Les numéros d'AS sont représentés comme suit (nous avons gardé ceux qui apparaissent le plus). On voit apparaître assez prévisiblement les AS de coœur en tête, tels que COGENT et Telia.

AS	nombre d'apparitions
"COGENT Cogent/PSI"	1799
"TELIANET TeliaNet Global Network"	480
"LEVEL3 Level 3 Communications"	432
"SPRINTLINK - Sprint"	392
"TELEFONICA Telefonica Backbone Autonomous System"	342
"SINGTEL-AS-AP Singapore Telecommunications Ltd"	155
"NTT-COMMUNICATIONS-2914 - NTT America, Inc."	135
"UUNET - MCI Communications Services, Inc. d/b/a Verizon Business"	134
"TINET-BACKBONE Tinet SpA"	102
"(Unknown)"	101
"GBLX Global Crossing Ltd."	98

Enfin, la répartition par pays indique ici aussi une prévalence des États-Unis dans les load-balancers. Le dénominatif "Europe" indique que le pays est inconnu, sa valeur a donc été remplacée par celle du continent où il se trouve.

pays	nombre d'apparitions
"United States"	3481
"Europe"	542
"Spain"	352
"Germany"	169
"Singapore"	159
"France"	154
"Canada"	96
"United Kingdom"	89
"Korea, Republic of"	70
"Japan"	67

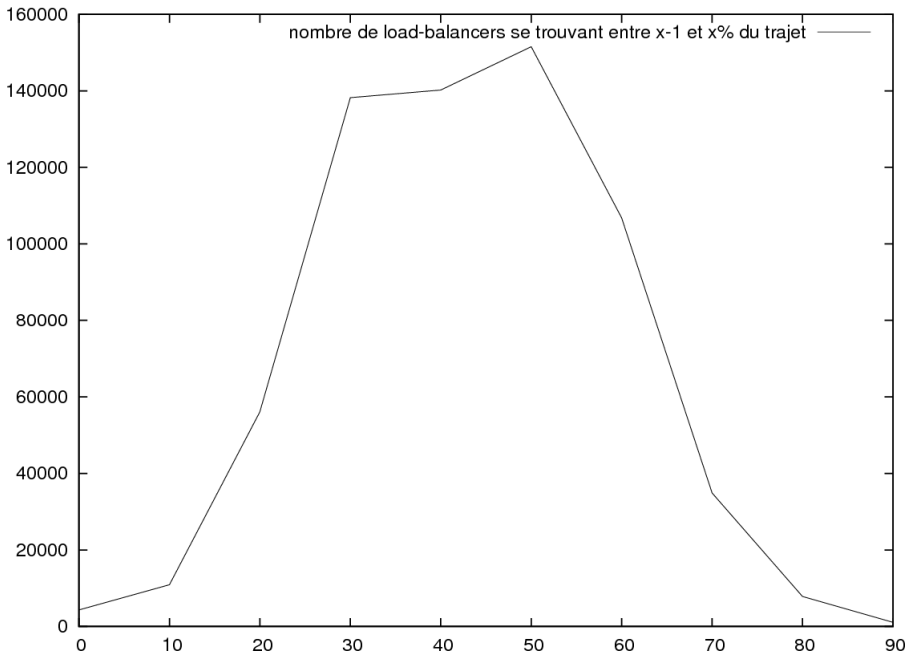

FIGURE 20 – Nombre d'interfaces vues pour un load-balancer en fonction de sa position le long du chemin

4.5.5 Présence des load-balancers

Nous avons cherché à estimer quel pourcentage des chemins parcourus traversaient au moins un type de load-balancer pendant leur parcours. On obtient que 15% des chemins parcourus passaient par au moins un load-balancer par flot, 0.23% par un load-balancer de type paquet et 0.98% de type indéterminé. Ce sont au total 16% des parcours de chemin qui ont rencontré au moins un load-balancer sur leur trajet. Ces données semblent aller dans le sens de celles réalisées par [4] qui expliquait que les faibles pourcentages de load-balancers s'expliquent par le fait que le réseau PlanetLab est principalement de type académique, et ce type de réseau utilise peu les load-balancers, alors que l'on trouverait une valeur beaucoup plus élevée pour un réseau commercial.

On estime d'autre part la position des load-balancers en divisant le TTL auquel ils se trouvent par le TTL de la destination et on fait une moyenne. On trouve ainsi 42% du trajet. Les load-balancers se trouvent donc plutot vers le milieu du trajet. La figure 20 montre la répartition en tranches de 10% des load-balancers sur les chemins parcourus

Parmi ces load-balancers, environ 16% sont des diamants, c'est-à-dire gèrent l'intégralité du flux entre la source et la destination.

La figure 21 montre la répartition des load-balancers en fonction de leur nombre d'interfaces. On constate que les load-balancers avec le plus de sorties se trouvent vers le milieu du trajet, ce qui semble indiquer le role décongestionnant de ces load-balancers pour le très gros trafic qui circule dans les AS de cœur.

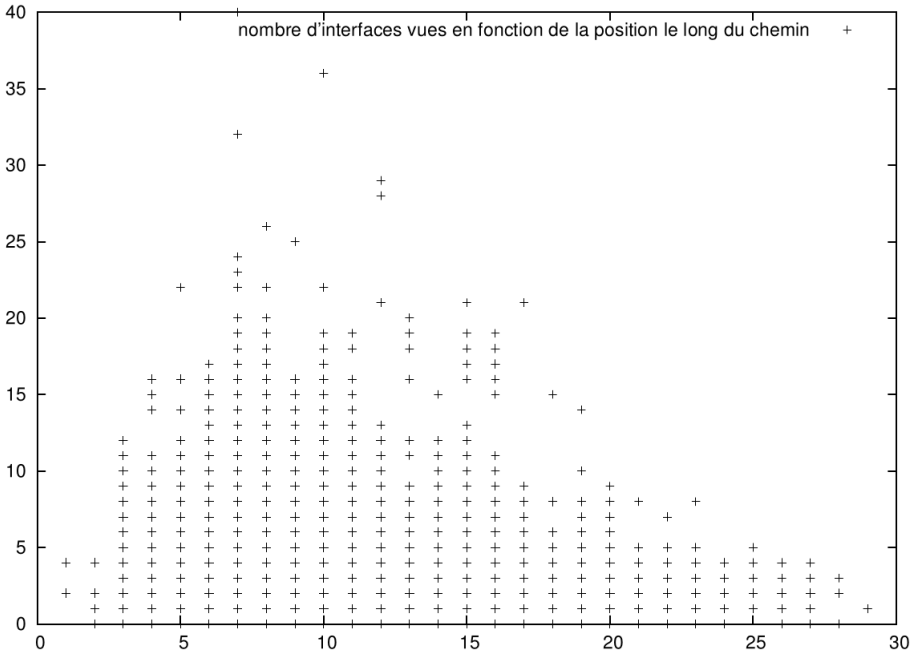

FIGURE 21 – Nombre d'interfaces vues pour un load-balancer en fonction de sa position le long du chemin

La figure 22 montre le nombre d'apparitions pour les 50 plus gros load-balancers dans nos mesures. On voit qu'un certain load-balancer a été traversé par près de 9000 explorations. On obtient pour ce load-balancer $'country' : 'Italy','continent' : 'Europe','asn' : 137','as\_types' : 'SmallISP(Tier-2)','as\_name' : 'ASGARR GARR Italian academic and research network'$ qui est pourtant un small ISP en Italie, ce qui est assez surprenant et nécessiterait une analyse plus poussée afin de comprendre ce résultat.

On observe pour ces 50 load-balancers les plus traversés les AS auxquels ils appartiennent. On obtient la liste suivante. On voit apparaître l'AS correspondant au load-balancer le plus fréquenté.

AS	nombre d'apparitions
"TELIANET TeliaNet Global Network"	18
"LEVEL3 Level 3 Communications"	12
"SPRINTLINK - Sprint"	6
"COGENT Cogent/PSI"	5
"ASGARR GARR Italian academic and research network"	2
"(Unknown)"	1
"MANDA Man-da.de GmbH"	1
"TINET-BACKBONE Tinet SpA"	1
"hetzner-as hetzner online ag rz-nuernberg"	1
"TPNET Telekomunikacja Polska S.A."	1
"ATT-INTERNET4 - AT&T Services Inc."	1
"Rede Nacional de Ensino e Pesquisa"	1

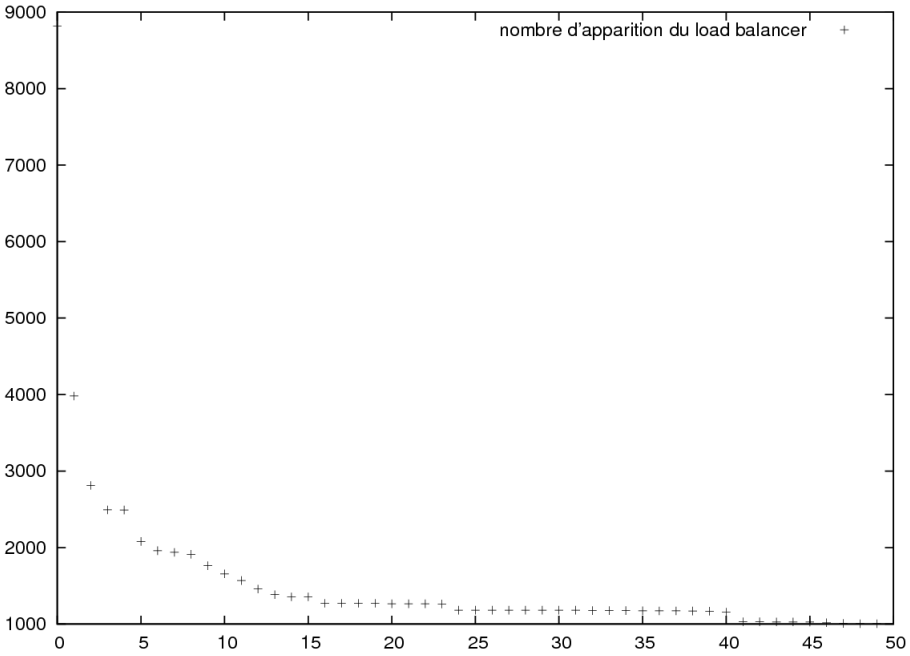

FIGURE 22 – Nombre d'apparitions pour les 50 plus gros load-balancers

Ils sont répartis entre les pays suivants.

Pays	nombre d'apparitions
"United States"	24
"Europe"	18
"Brazil"	2
"Italy"	2
"None"	1
"Poland"	1
"Germany"	1
"France"	1

Les types d'AS dans lesquels ils se trouvent sont de type IPS Large ou Small, ce qui est le résultat attendu.

type d'AS	nombre d'apparitions
"Large ISP (Tier-1)"	42
"Small ISP (Tier-2)"	6
"(Unknown)"	2

On observe également les cinquante plus gros load-balancers par paquet, afin de regarder où se trouvent ceux-ci dans le réseau. On constate qu'ils se trouvent majoritairement dans les small-ISP, mais qu'on en trouve également dans les larges ISP, ce qui peut être problématique par exemple pour TCP ou l'ordre des paquets est important. Le désordonnancement des paquets peut provoquer des problèmes de performance.

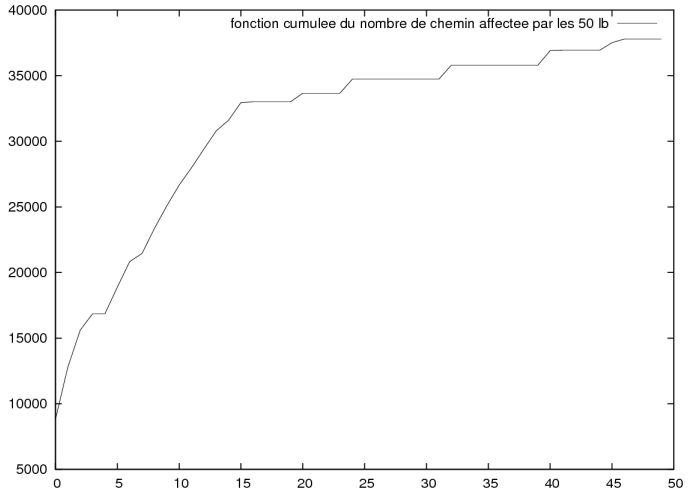

FIGURE 23 – fonction cumulée pour tous types de load-balancers

type d'AS	nombre d'apparitions
"Small ISP (Tier-2)"	28
"Large ISP (Tier-1)"	18
"(Unknown)"	3
"University"	1

Cela correspond principalement aux AS suivants :

AS	nombre d'apparitions
"TELEFONICA Telefonica Backbone Autonomous System"	22
"CW Cable and Wireless Worldwide plc"	6
"COGENT Cogent/PSI"	4
"TELIANET TeliaNet Global Network"	4

TELEFONICA est un tier-2 tandis que les trois autres sont des tier-1, qui ont également beaucoup de load-balancers par flot. De plus, étant donné la faible proportion des load-balancers par paquet qui a été vue, on peut penser qu'il s'agit d'un problème de configuration de leurs routeurs.

On cherche aussi à observer la fonction cumulée pour chaque type de load-balancer et pour tous confondu du nombre de chemins affectés par ces load-balancers. On obtient les figures 23 à 26. On va pour cela récupérer pour chaque type ou pour tous réunis les 50 load-balancers les plus présents. On va ensuite récupérer leurs apparitions sur chaque trace en ne comptant un load-balancer que si tous les load-balancers à plus fort taux de présence ne sont pas présents dans cette trace. Autrement dit, on ne considère à chaque fois que le load-balancer à plus fort taux de présence de la trace. On va ensuite cumuler les résultats obtenus successivement par les load-balancers du premier au 50ème.

On voit que certains load-balancers n'affectent pas beaucoup de chemins, en particulier pour tous types et flow. Toutes deux se ressemblent à cause du fort surnombre de ce type de load-balancer qui fait que lorsque l'on prend tous les load-balancers, on prend principalement des load-balancers par flot. Ces

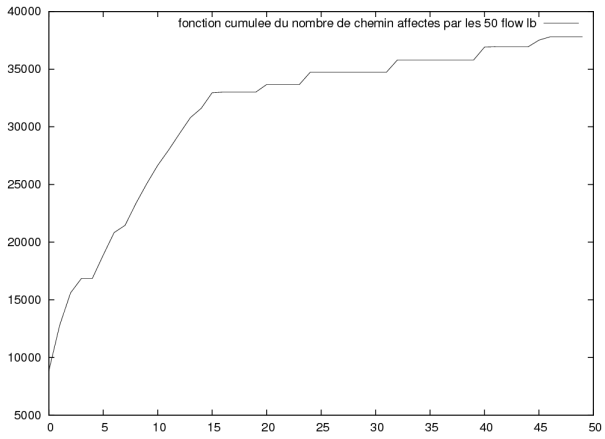

FIGURE 24 – fonction cumulée pour les load-balancers par flot

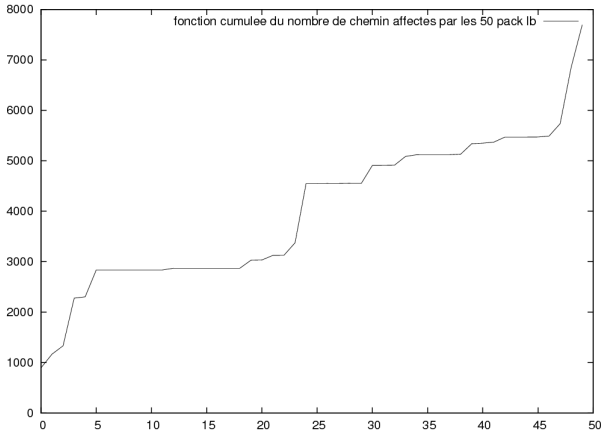

FIGURE 25 – fonction cumulée pour les load-balancers par paquet

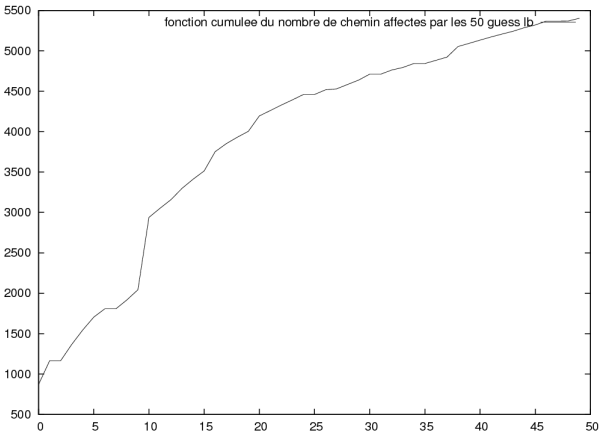

FIGURE 26 – fonction cumulée pour les load-balancers indéterminés

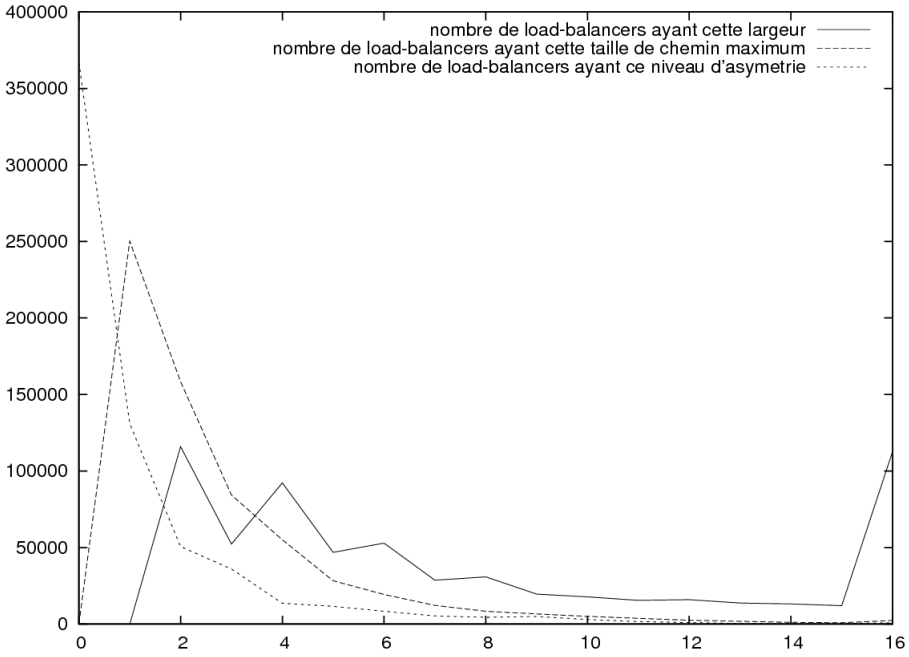

FIGURE 27 – nombre de load-balancers type flow vus ayant Largeur, longueur et asymétrie de valeur x

load-balancers ne font augmenter que peu la courbe, alors qu'au contraire certains load-balancers affectent un nombre plus conséquent, donnant à la courbe une forte pente. Ces load-balancers sont donc particulièrement interessants à connaître.

On voit que les load-balancers par paquet n'ont pas d'élément dominant, ce qui fait que la courbe a une progression plus linéaire.

4.5.6 Propriétés des load-balancers

Nous allons maintenant chercher à caractériser les propriétés pour chaque type de load-balancer. On va chercher à caractériser les données telles que la longueur maximale des chemins qui sortent de ce load-balancer, ou encore le nombre maximum de chemins qui existent dans ce load-balancer. Les tableaux présentées figures 27, 28 et 29 résument les données obtenues.

Chaque graphe représente un type de load-balancer. la courbe "largeur" indique pour chaque largeur x lue en abscisse, combien de load-balancers ayant cette largeur ont été croisés. Il est important de signaler que l'on compte vraiment le nombre d'apparitions, et donc que le même load-balancer peut avoir été vu plusieurs fois. L'étude avec unification est prévue ultérieurement. La courbe "taille de chemin maximum" indique pour la longueur x, le nombre de load-balancers ayant été rencontrés dont le chemin de sortie de taille maximum faisait cette taille x. Enfin, la courbe "niveau d'asymétrie" indique pour une valeur d'asymétrie x, combien de load-balancers ont obtenu cet écart. L'asymétrie

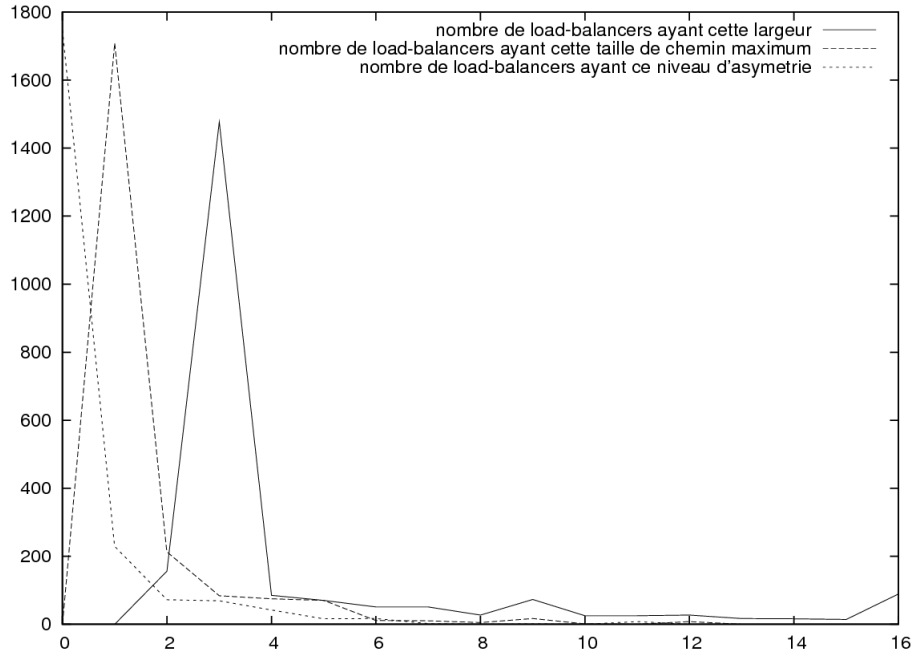

FIGURE 28 – nombre de load-balancers type packet vus ayant Largeur, longueur et asymétrie de valeur x

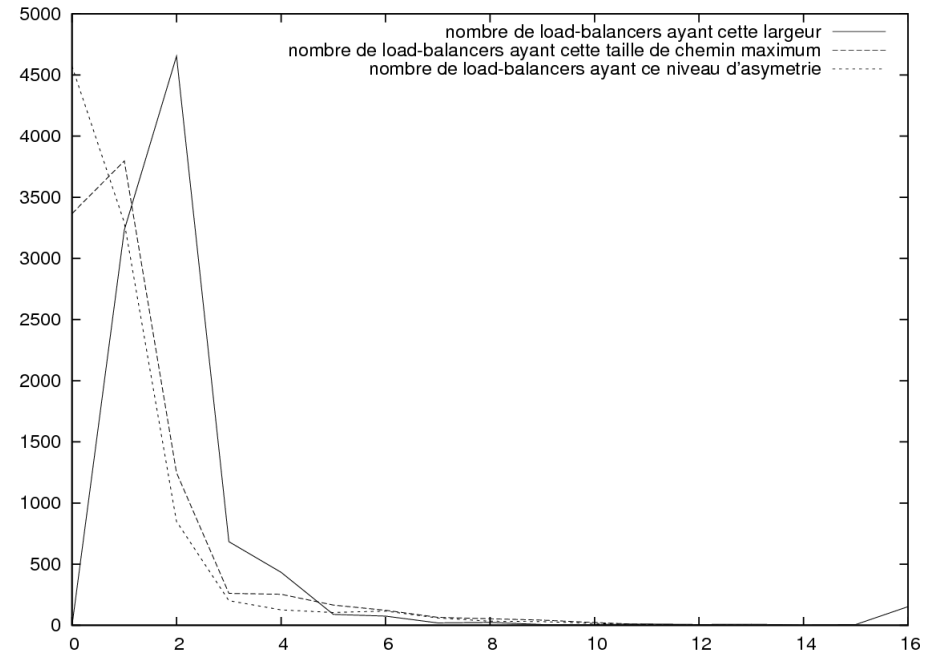

FIGURE 29 – nombre de load-balancers inconnu flow vus ayant Largeur, longueur et asymétrie de valeur x

représente ici l'écart, c'est-à-dire le nombre de nœuds de différence entre le plus grand et le plus petit chemin sortant du load-balancer.

En ce qui concerne la largeur, on remarque que les load-balancers des diférents types ont tous tendance à avoir entre deux et six interfaces, et qu'après cela la valeur décroît, sauf pour les load-balancers par flot où il semble qu'un nombre conséquent de nœuds ont un nombre de sorties supérieur à 16. Cela va de pair avec la figure 21 qui montrait que les load-balancers centraux ont tendance à avoir un grand nombre d'interfaces de sortie. En ce qui concerne la taille de chemin maximum, on observe que les load-balancers des trois types ont tendance à avoir un chemin maximum très court, indiquant un seul nœud de séparation avant reconvergence des données. Les load-balancers à deux sauts sont plus rares, mais encore assez présents. Au delà, cela devient vraiment plus rare, excepté pour les load-balancers par flot où la valeur est encore notable.

En ce qui concerne l'asymétrie, on constate que la plus grande valeur est de loin 0, signifiant que la plupart des load-balancers ont des chemins symétriques. On voit que les valeurs au delà de 4 sont négligeables.

4.6 Futurs travaux

4.6.1 Bibliothèque Paris-traceroute

Notre bibliothèque remplit pour l'instant les fonctions vitales pour le déploiement sur TopHat et l'utilisation de l'algorithme MDA. Cependant, nous voudrions généraliser son utilisation. Pour cela, nous allons rendre disponible sur un dépôt de type Git le contenu de notre bibliothèque, afin d'obtenir des retours et de l'améliorer. Nous voulons aussi créer des exécutables appelant notre bibliothèque qui effectuent les mêmes mesures que traceroute, tcptraceroute ou tracert (ICMP) mais avec le principe de paris-traceroute pour corriger les erreurs.

4.6.2 Analyse des load-balancers de PlanetLab

Les résultats de notre analyse sur TopHat permettront d'autre part de rendre disponible une mesure de la topologie de ce réseau dans laquelle apparaissent les load-balancers, ce qui pourra servir à ceux qui veulent utiliser, ou tout du moins connaître l'existence de ce phénomène pour leurs travaux sur PlanetLab avec TopHat. Il faudrait aussi réaliser plusieurs fois cette même analyse afin de vérifier si ce sont des données consistantes et stables.

Au niveau des analyses, il est prévu ultérieurement de regarder les propriétés par type de load-balancer mais en rendant unique leurs apparitions. Il est aussi prévu de regarder la disposition des load-balancers par rapport aux AS, et comparer les load-balancers Interdomaine avec les load-balancers Intradomaine. plusieurs points pourront être testés :

- tester avec d'autres protocoles que UDP
- prendre en compte MPLS
- tenir compte des étoiles (routeurs non responsifs)

- réussir à caractériser et repérer les load-balancers par destination
- impact de la dynamique (il serait nécessaire de comparer plusieurs jeux de données)
- destinations externes au testbed (volontaire)
- étude au niveau routeur (il peut y avoir plusieurs liens parallèles entre 2 mêmes routeurs, mais comme les interfaces ont des IP différentes, on ne le voit pas), voir [12]

5 Conclusion et bilan personnel

Nous avons donc cherché à concevoir une bibliothèque qui se veut modulaire et qui permet de créer les différents algorithmes que l'on souhaite et à les distribuer facilement, afin de faire des topologies efficaces d'internet ou du réseau de notre choix. Cette bibliothèque répondra ainsi aux différents besoins exprimés, tels que la généralisation de paris-traceroute et la distribution d'algorithmes optimisant la mesure de topologie. Elle évitera le phénomène de "réinvention de la roue" où tester les algorithmes ou les améliorations de traceroute nécessite le développement complet d'un outil. Elle facilitera d'autre part les futures expérimentations sur de nouveaux algorithmes et l'ajout de protocoles grâce à sa modularité. Du côté de l'analyse, nous avons établi une caractérisation du réseau PlanetLab, autant du côté des sources et des destinations qu'au niveau des load-balancers. On a ainsi pu établir le niveau d'influence qu'ils avaient sur le réseau et où ils se trouvent dans celui-ci. On a aussi pu caractériser ces load-balancers au niveau de leurs tailles et de leur symétrie. Cette expérience dans le monde de la recherche s'est révélé une expérience très enrichissante. La Recherche nécessite beaucoup d'autonomie tout en offrant beaucoup de liberté sur la façon dont le travail est traité. J'ai apprécié le fait d'être seule responsable et force de décision sur l'outil que je développais, tout en pouvant discuter des choix d'implémentation avec Jordan. Travailler dans la recherche m'a également permis de voir que les algorithmes ou outils que concoivent les gens sont le fruit d'un travail parfois long et dur, et qu'il faut garder cela en tête avant de critiquer leur travail trop facilement.

Glossaire

AS Un Autonomous System, abrégé en AS, ou Système Autonome, est un ensemble de réseaux informatiques IP intégrés à Internet et dont la politique de routage interne (routes à choisir en priorité, filtrage des annonces) est cohérente. Un AS est généralement sous le contrôle d'une entité unique, typiquement un fournisseur d'accès à Internet.(Wikipedia). 25, 28

binding programme/outil qui permet l'utilisation d'une bibliothèque logicielle dans un autre langage de programmation que celui avec lequel elle a été écrite. On parle alors binding de langage (Wikipedia). 17

callback fonction appelée lors de la réception d'un paquet par le sniffer de la bibliothèque libpcap. 22, 23

checksum ou somme de contrôle, nombre codé sur deux octets permettant d'assurer l'intégrité d'un en-tête, calculé en sommant tous les champs de celui-ci. 10

load-balancer Un load-balancer est un noeud dans un réseau possédant au moins deux interfaces de sortie, et qui va donc séparer le flux entrant selon certains critères entre ces deux sorties. 2, 6

moniteur Noeud du reseau qui va effectuer une mesure, en lançant un outil tel que traceroute ou autre. 6, 7

probe paquet qui va être envoyé pour faire de l'exploration sur le réseau (traduction : sonde). 2, 6

snapshot Capture de l'état du réseau à un moment donné par réalisation d'une multitude de mesures qui permettront par recoupement de générer une "carte". 9

socket descripteur de fichier permettant l'envoi et la réception de paquets entre le réseau et l'ordinateur local. Une Raw socket est une socket où c'est l'utilisateur qui remplit au préalable tous les en-têtes jusqu'au niveau IP inclus. 17, 22

testbed système physique permettant de mettre un produit en conditions d'utilisation paramétrables et contrôlées afin d'observer et mesurer son comportement. 2, 8

TTL Time To Live. c'est le nombre de nœuds que pourra encore parcourir un paquet avant d'être détruit (traduction "temps à vivre"). 6

wrapper Il permet de convertir l'interface d'une classe en une autre interface que le client attend. L' Adaptateur fait fonctionner un ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces(Wikipedia). 26

Références

[1] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien et R. Teixeira : Avoiding traceroute anomalies with paris traceroute. *In Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158. ACM, 2006.

[2] B. Augustin, T. Friedman et R. Teixeira : Measuring load-balanced paths in the internet. *In Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 149–160. ACM, 2007.

[3] B. Augustin, T. Friedman et R. Teixeira : Multipath tracing with paris traceroute. *In End-to-End Monitoring Techniques and Services, 2007. E2EMON'07. Workshop on*, pages 1–8. IEEE, 2007.

[4] B. Augustin, T. Friedman et R. Teixeira : Measuring multipath routing in the internet. *Networking, IEEE/ACM Transactions on*, (99):830–840, 2010.

[5] P. BIONDI : Network packet manipulation with scapy, 2005.

[6] Í. Cunha, R. Teixeira, D. Veitch et C. Diot : Predicting and tracking internet path changes. 2011.

[7] B. Donnet, P. Raoult, T. Friedman et M. Crovella : Efficient algorithms for large-scale topology discovery. *In ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 327–338. ACM, 2005.

[8] R. Govindan et H. Tangmunarunkit : Heuristics for internet map discovery. *In INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1371–1380. IEEE, 2000.

[9] M. Gunes et K. Sarac : Analyzing router responsiveness to active measurement probes. *Passive and Active Network Measurement*, pages 23–32, 2009.

[10] M. Latapy, C. Magnien et F. Ouédraogo : A radar for the internet. *In Data Mining Workshops, 2008. ICDMW'08. IEEE International Conference on*, pages 901–908. IEEE, 2008.

[11] M. Luckie : Scamper : a scalable and extensible packet prober for active measurement of the internet. *In Proceedings of the 10th annual conference on Internet measurement*, pages 239–245. ACM, 2010.

[12] M. Luckie, A. Dhamdhere *et al.* : Measured impact of crooked traceroute. *ACM SIGCOMM Computer Communication Review*, 41(1):14–21, 2011.

[13] H.V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy et A. Venkataramani : iplane : An information plane for distributed services. *In Proceedings of the 7th symposium on Operating systems design and implementation*, pages 367–380. USENIX Association, 2006.

[14] Y. Shavitt et E. Shir : Dimes : Let the internet measure itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, 2005.

[15] Y. Shavitt et U. Weinsberg : Quantifying the importance of vantage points distribution in internet topology measurements. *In INFOCOM 2009, IEEE*, pages 792–800. IEEE, 2009.

[16] M. Tozal et K. Sarac : Tracenet : an internet topology data collector. *In Proceedings of the 10th annual conference on Internet measurement*, pages 356–368. ACM, 2010.

[17] D. Veitch, B. Augustin, R. Teixeira et T. Friedman : Failure control in multipath route tracing. *In INFOCOM 2009, IEEE*, pages 1395–1403. IEEE, 2009.